SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

AN APPLICATION OF BIN-PACKING TO MULTIPROCESSOR SCHEDULING*

E. G. COFFMAN, JR.,† M. R. GAREY‡ AND D. S. JOHNSON‡

Abstract. We consider one of the basic, well-studied problems of scheduling theory, that of nonpreemptively scheduling n independent tasks on m identical, parallel processors with the objective of minimizing the "makespan," i.e., the total timespan required to process all the given tasks. Because this problem is *NP*-complete and apparently intractable in general, much effort has been directed toward devising fast algorithms which find near-optimal schedules. The well-known LPT (Largest Processing Time first) algorithm always finds a schedule having makespan within $4/3 = 1.333\cdots$ of the minimum possible makespan, and this is the best such bound satisfied by any previously published fast algorithm. We describe a comparably fast algorithm, based on techniques from "bin-packing," which we prove satisfies a bound of 1.220. On the basis of exact upper bounds determined for each $m \leqq 7$, we conjecture that the best possible general bound for our algorithm is actually $20/17 = 1.176\cdots$.

Key words. bin packing, multiprocessor scheduling, approximation algorithms, worst-case analysis, performance bounds

1. Introduction. One of the fundamental problems of deterministic scheduling theory is that of scheduling independent tasks on a nonpreemptive multiprocessor system so as to minimize overall finishing time [1, Chap. 5], [2], [3]. Formally, we are given a set $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$ of *tasks*, each task T_i having *length* $l(T_i)$, and a set of $m \geqq 2$ identical *processors*. A *schedule* in this case can be thought of as a partition $\mathcal{P} = \langle P_1, P_2, \cdots, P_m \rangle$ of $\mathcal{T}$ into m disjoint sets, one for each processor. The ith processor, $1 \leqq i \leqq m$, executes the tasks in P_i. (Since the tasks are assumed to be independent, we may restrict our attention to schedules in which no idle time is inserted between consecutively executed tasks. For the same reason, the particular sequence in which tasks are executed on a processor is unconstrained and irrelevant for this problem.) The *finishing time* for the schedule $\mathcal{P}$ is then given by

$$f(\mathcal{P}) = \max_{1 \leqq i \leqq m} l(P_i)$$

where for any $X \subseteq \mathcal{T}$, $l(X)$ is defined to be $\sum_{T \in X} l(T)$.

An *optimum* m-processor schedule $\mathcal{P}^*$ is one that satisfies $f(\mathcal{P}^*) \leqq f(\mathcal{P})$ for all partitions $\mathcal{P}$ of $\mathcal{T}$ into m subsets. Since there are only a finite number of possible partitions, such an optimum schedule must exist, and we let $\mathcal{T}_m^* = f(\mathcal{P}^*)$ denote its finishing time.

The problem of finding an optimum schedule appears to be quite difficult in general. All known algorithms require computation time that grows exponentially with the number of tasks. This is not surprising, however, in light of the fact that

* Received by the editors September 28, 1976.

† Department of Computer Science, Pennsylvania State University, University Park, Pennsylvania 16802.

‡ Bell Laboratories, Murray Hill, New Jersey 07974.

this problem is known to be "*NP*-complete" and hence computationally "equivalent" to a host of other notoriously intractable problems [8]. These computational difficulties force us to lower our sights somewhat and seek instead reasonably efficient algorithms that find "near-optimal" schedules.

Let A be an algorithm that, when given $\mathcal{T}$ and m, constructs a partition $\mathcal{P}_A[\mathcal{T}, m]$ of $\mathcal{T}$ into m subsets. We shall use $F_A[\mathcal{T}, m]$ to denote the finishing time of $\mathcal{P}_A[\mathcal{T}, m]$, i.e., $F_A[\mathcal{T}, m] = f(\mathcal{P}_A[\mathcal{T}, m])$. The *m-processor performance ratio* for A is then defined by

$$R_m(A) = \sup\left\{\frac{F_A[\mathcal{T}, m]}{\mathcal{T}^*_m} : \text{all task sets } \mathcal{T}\right\}.$$

We would like to find an efficient algorithm A such that $R_m(A)$ is as close to 1 as possible, for all $m \geqq 2$ (the problem is trivial for $m = 1$).

In [2], [3], R. L. Graham describes a sequence $A_1, A_2, \cdots$ of algorithms such that $\lim_{i\to\infty} R_m(A_i) = 1$ for all $m \geqq 2$. Unfortunately these algorithms require computation time growing exponentially with m and become more and more like exhaustive search as the guaranteed accuracy improves. (Sahni presents similarly behaved algorithms in [7]). The best of the previously published *polynomial-time* algorithms, also in [2], [3], is the LPT algorithm, which satisfies

$$R_m(\text{LPT}) = \frac{4}{3} - \frac{1}{3m}.$$

In this paper we present a simple iterative algorithm, based on ideas from bin packing [5], [6], which substantially improves on this worst-case performance and also seems to outperform LPT on the average.[1]

In the next section we discuss the bin-packing problem and the well known first fit decreasing algorithm for it. We then describe the new results about this algorithm which have motivated the design of our scheduling algorithm and helped us produce upper bounds on its worst case behavior. The scheduling algorithm itself, called MULTIFIT, is then presented in § 3, along with our results about it. The remainder of the paper, §§ 4 and 5, are devoted to the proofs of the new bin-packing results.

2. Bin-packing. Our scheduling algorithm is based on the bin-packing algorithm first fit decreasing (abbreviated FFD). The bin-packing problem is in a sense the dual problem to the scheduling problem defined above. Given $\mathcal{T}$ as before, and a bound C, a *packing* is a partition $\mathcal{P} = \langle P_1, P_2, \cdots, P_m\rangle$ of $\mathcal{T}$ such that $l(P_i) \leqq C$, $1 \leqq i \leqq m$. The tasks T_i are here thought of as *items* with *size* $l(T_i)$, which are placed in *bins* of *capacity* C. Our goal is to minimize the number m of bins used in the packing. We let $OPT[\mathcal{T}, C]$ denote this minimum possible value of m. As before, the problem of finding an optimum packing appears to be intractable. The algorithm FFD is an attempt to find a near-optimum packing quickly. It constructs a packing $\mathcal{P}_{FFD}[\mathcal{T}, C]$ as follows.

[1] The algorithm described here is also an improvement, both in worst-case behavior and experimental average case behavior, over the earlier version described in [4].

First, the items in $\mathcal{T}$ are put in "decreasing order," that is, they are re-indexed so that $l(T_1) \geqq l(T_2) \geqq \cdots \geqq l(T_n)$. Then a packing is built up by treating each item in succession, and adding it to the lowest indexed bin into which it will fit without violating the capacity constraint. More formally, we might describe the packing procedure as follows:

1. Set $P_i \leftarrow \varphi$, $1 \leqq i \leqq n$, and $j \leftarrow 1$.
2. Set $k \leftarrow \min \{i \geqq 1: l(P_i) + l(T_j) \leqq C\}$.
3. Set $P_k \leftarrow P_k \cup \{T_j\}$, $j \leftarrow j+1$.
4. If $j \leqq n$, go to 2. Otherwise, halt.

Let us denote by $FFD[\mathcal{T}, C]$ the number m of nonempty bins in $\mathcal{P}_{FFD}[\mathcal{T}, C]$. It was proved in [5], [6] that for all $\mathcal{T}$ and C

$$FFD[\mathcal{T}, C] \leqq \frac{11}{9} OPT[\mathcal{T}, C] + 4.$$

In the following, however, we shall be interested in a different question about FFD: namely, "given $\mathcal{T}$ and m, how large does C have to be so that $FFD[\mathcal{T}, C] \leqq m$?"

This question will of course have many different answers, depending on $\mathcal{T}$ and m. We are interested, however, in finding an answer that works for *all* $\mathcal{T}$ and m. To obtain such an answer, we shall ask "how large does C have to be, *in terms of* $\mathcal{T}^*_m$, so that, for all $\mathcal{T}$ and m, $FFD[\mathcal{T}, C]$ is guaranteed to be m or less?"

Recalling our definition of $\mathcal{T}^*_m$ from § 1, we note that, in terms of the bin packing problem,

$$\mathcal{T}^*_m = \min \{C: OPT[\mathcal{T}, C] \leqq m\}.$$

Thus $\mathcal{T}^*_m$ is the smallest bin capacity which allows $\mathcal{T}$ to be packed into m or fewer bins. We now define

$$r_m = \inf \{r: \text{for all } \mathcal{T}, FFD[\mathcal{T}, r\mathcal{T}^*_m] \leqq m\}.$$

The intent of this definition is that r_m be the least "expansion factor" by which the optimum bin capacity should be enlarged to guarantee that FFD will use no more than m bins. However, the definition allows for the possibility that, while expansion factors arbitrarily close to r_m will work, the limiting value r_m itself does not. That this cannot happen (and hence that "inf" could have been replaced by "min") is proved as part of the following "monotonicity lemma."

LEMMA 2.1. *For every* $\mathcal{T}$ *and any* $r \geqq r_m$, $FFD[\mathcal{T}, r\mathcal{T}^*_m] \leqq m$.

Proof. We first show that $FFD[\mathcal{T}, r_m\mathcal{T}^*_m] \leqq m$ for every $\mathcal{T}$. Suppose, to the contrary, that $\mathcal{T}$ is a set for which $FFD[\mathcal{T}, r_m\mathcal{T}^*_m] > m$. Consider the application of the FFD algorithm to $\mathcal{T}$ with capacity $C = r_m\mathcal{T}^*_m$. Each time the algorithm places a particular item in a bin other than the first bin, this is because the size of that item would have caused the total size in each lower-indexed bin to exceed C by some positive amount. Let δ be the least such excess over all items and all such "unsuccessful" attempted placements. Then, for every capacity C', $C \leqq C' < C + \delta$, exactly the same packing will be obtained with bin capacity C' as with C, and hence $FFD[\mathcal{T}, C'] > m$. However, by the definition of r_m, we know that there exist ratios r arbitrarily close to r_m for which $FFD[\mathcal{T}, r\mathcal{T}^*_m] \leqq m$. Choosing such an

r with $r\mathcal{T}_m^* < C + \delta$, we obtain a contradiction to the assumed counter-example, proving that $FFD[\mathcal{T}, r_m\mathcal{T}_m^*] \leqq m$.

Now suppose that, for some $\mathcal{T}$ and $\alpha > 1$, $FFD[\mathcal{T}, \alpha r_m\mathcal{T}_m^*] > m$. We shall use $\mathcal{T}$ to construct a set $\bar{\mathcal{T}}$ for which $FFD[\bar{\mathcal{T}}, r_m\bar{\mathcal{T}}_m^*] > m$, contradicting what we have just proved. Consider any optimal packing of $\mathcal{T}$ into m bins of size $\mathcal{T}_m^*$. Enlarge the capacity of each optimal bin to $\alpha\mathcal{T}_m^*$ and augment $\mathcal{T}$ to form $\bar{\mathcal{T}}$ by adding new items, each smaller than the smallest item in $\mathcal{T}$, so that every enlarged bin becomes completely filled. Thus we have that $\bar{\mathcal{T}}_m^* = \alpha\mathcal{T}_m^*$. Furthermore $FFD[\bar{\mathcal{T}}, \alpha r_m\mathcal{T}_m^*] > m$ because the $(m+1)$st bin will be started before any of the new items are placed. But then we have $FFD[\bar{\mathcal{T}}, r_m\bar{\mathcal{T}}_m^*] > m$, which is the desired contradiction. The lemma follows. □

Thus r_m is indeed the desired minimum "expansion factor" and every capacity greater than $r_m\mathcal{T}_m^*$ will also work, so that we need not be concerned about possible anomalies. By determining the values of r_m, $m \geqq 2$, we will therefore be obtaining general answers to the question "how large does C have to be, in terms of $\mathcal{T}_m^*$?"

Table 1 gives the best upper and lower bounds we have discovered for r_m. As can be seen, we know the exact value of r_m for $2 \leqq m \leqq 7$, and our upper and lower bounds are quite close for $m \geqq 8$. Since $r_4 = r_5 = r_6 = r_7 = \frac{20}{17}$, and the best lower bound we have been able to find for $m \geqq 8$ is also $\frac{20}{17}$, we conjecture that in fact $r_m = \frac{20}{17}$ for all $m \geqq 4$.

The proofs of the lower bounds cited in Table 1 are straightforward, since all that is required is an example $\mathcal{T}$ such that $FFD(\mathcal{T}, C) > m$ whenever $C <$ (lower bound) $\cdot\, \mathcal{T}_m^*$. These lower bound examples for $m = 2$, $m = 3$, and $m \geqq 4$ are illustrated in Figs. 1, 2, and 3. In each case the items are represented by rectangles which are labeled by their sizes, and the bins by stacks of items.

The proofs of the upper bounds for $m \leqq 7$ involve case analyses which grow more and more complicated as m increases. These proofs have been omitted at the suggestion of the referees, but a full version of this paper containing them is available from the authors. We shall limit ourselves here to proving the general 1.220 upper bound, which in fact holds for all $m \geqq 2$. The proof will be postponed until §§ 4 and 5, however, so that we may first see how we incorporate FFD into a scheduling algorithm, and how we use the values of r_m to bound the worst-case behavior of that algorithm.

TABLE 1
Bounds on r_m

m	2	3	4, 5, 6, 7	8 or more
Upper bound on r_m	$\frac{8}{7}$	$\frac{15}{13}$	$\frac{20}{17}$	1.220
Lower bound on r_m	$\frac{8}{7}$	$\frac{15}{13}$	$\frac{20}{17}$	$\frac{20}{17} = 1.176\cdots$

3. The algorithm MULTIFIT. In the light of the results described in the preceding section, one might propose the following scheduling algorithm: Given $\mathcal{T}$ and m, set $C = r_m \cdot \mathcal{T}_m^*$ and run FFD on $\mathcal{T}$ and C. By definition of r_m,

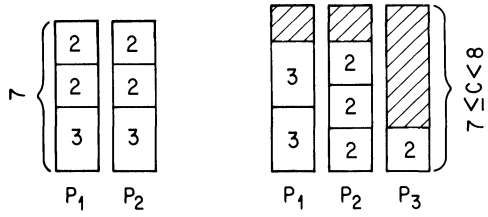


FIG. 1. *Example showing* $r_2 \geqq \frac{8}{7}$

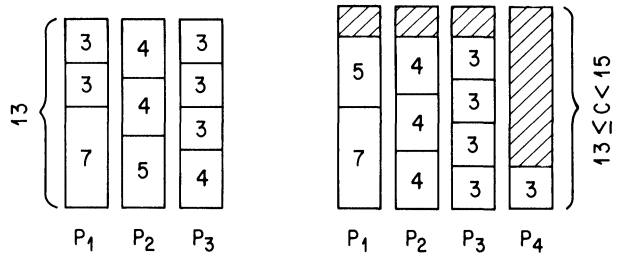


FIG. 2. *Example showing* $r_3 \geqq \frac{15}{13}$

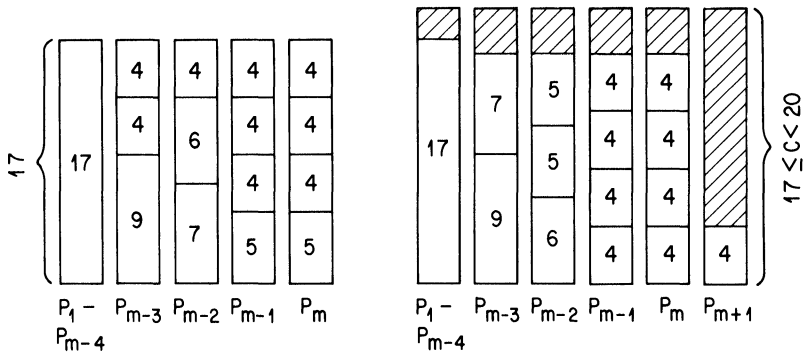


FIG. 3. *Example showing* $r_m \geqq \frac{20}{17}$, *for all* $m \geqq 4$

$\mathcal{P}_{FFD}[\mathcal{T}, C]$ will be made up of m or fewer sets, and hence will correspond to a valid schedule for $\mathcal{T}$ and m with finishing time $C = r_m \cdot \mathcal{T}^*_m$ or less, which would certainly be "near-optimal" in light of our bounds on r_m.

Unfortunately this approach depends on knowing the value of $\mathcal{T}^*_m$ in advance, and it is just as difficult to determine $\mathcal{T}^*_m$ as to find an optimum schedule. A second idea would be to make repeated trials with FFD and different values of C until we find the least C for which $FFD[\mathcal{T}, C] \leqq m$, and take the schedule resulting from this bin capacity. If we assume that all tasks have integer lengths, there are two natural ways to attempt this. One would be to try each integer C in turn, starting from some obvious lower bound on $\mathcal{T}^*_m$, until we found one for which $FFD(\mathcal{T}, C) \leqq m$; this would be the desired minimum value of C. Unfortunately, this procedure might require a large number of repeated trials of FFD and would be very costly in terms of running time. A common way of reducing the number of trials in such a search is to use *binary search*: Start with known upper and lower bounds on C, and at each step run FFD for a value of C midway between the current upper and lower bounds. If $FFD[\mathcal{T}, C] > m$, C becomes the new lower bound and we continue; if $FFD[\mathcal{T}, C] \leqq m$, C becomes the new upper bound. At each step we thus halve the potential range and so we should narrow in on the minimum value of C quite rapidly.

Unfortunately, binary search will only be guaranteed to do this if for every $C_1 < C_2$, $FFD[\mathcal{T}, C_1] \leqq m$ implies $FFD[\mathcal{T}, C_2] \leqq m$. Although this general monotonicity property does hold for $m = 2$, it fails for $m \geqq 3$. Figure 4 demonstrates that there exists a list $\mathcal{T}$ such that $FFD[\mathcal{T}, 60] = 3$ but $FFD[\mathcal{T}, 61] = 4$. Thus binary search is *not* guaranteed to find the least C such that $FFD[\mathcal{T}, C] \leqq m$. However, recall that Lemma 2.1 tells us that binary search *will* be guaranteed to narrow in on a capacity $C \leqq r_m \mathcal{T}^*_m$. Therefore, binary search can still be used to find a near-optimum schedule, and this is the approach taken in our algorithm MULTIFIT.

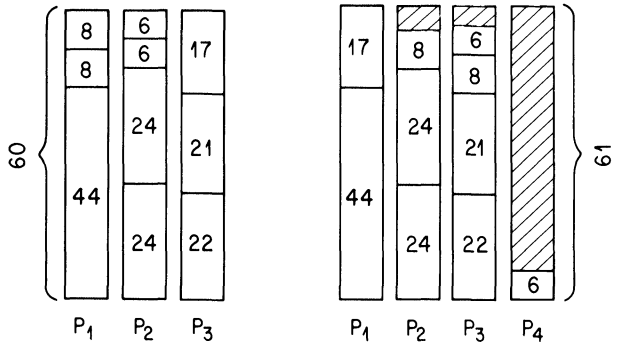


FIG. 4. $\mathcal{T}$ *such that* $FFD[\mathcal{T}, 61] > FFD[\mathcal{T}, 60]$

We begin our description of MULTIFIT by specifying the initial lower and upper bounds, C_L and C_U, used in the binary search. These are given in the following two lemmas.

LEMMA 3.1. *Let* $C_L[\mathcal{T}, m] = \max\{(1/m)l(\mathcal{T}), \max_i\{l(T_i)\}\}$. *Then for all* $C < C_L[\mathcal{T}, m]$, $FFD[\mathcal{T}, C] > m$.

Proof. This follows from the obvious fact that $\mathcal{T}^*_m \geqq C_L[\mathcal{T}, m]$.

LEMMA 3.2. *Let* $C_U[\mathcal{T}, m] = \max\{(2/m)l(\mathcal{T}), \max_i\{l(T_i)\}\}$. *Then for all* $C \geqq C_U[\mathcal{T}, m]$, $FFD[\mathcal{T}, C] \leqq m$.

Proof. Suppose $C \geqq C_U[\mathcal{T}, m]$ and $FFD[\mathcal{T}, C] > m$, and assume without loss of generality that $l(T_1) \geqq l(T_2) \geqq \cdots \geqq l(T_n)$, that is, $\mathcal{T}$ is indexed in nonincreasing order by item size. Let T_j be the first item to go in bin P_{m+1} under FFD. Clearly we must have $j \geqq m+1$. If $l(T_j) > C/2$, then since $\mathcal{T}$ is indexed in nonincreasing order $l(T_i) > C/2$, $1 \leqq i \leqq m+1$. This implies

$$l(\mathcal{T}) > \frac{mC}{2} \geqq \frac{mC_U[\mathcal{T}, m]}{2} \geqq l(\mathcal{T}),$$

a contradiction. If on the other hand $l(T_j) \leqq C/2$, then by the fact that T_j did not fit in any of the first m bins, each must have contained items whose total size exceeded $C/2$, and we again have $l(\mathcal{T}) > mC/2$ and a contradiction. □

We are now prepared to give a precise description of MULTIFIT. MULTIFIT takes as input a task set $\mathcal{T}$, a number of processors m, and a bound k on the desired number of iterations. Its first step is to put $\mathcal{T}$ into nonincreasing order, so that all subsequent applications of FFD will not have to reorder $\mathcal{T}$ themselves. It then proceeds by binary search for the desired number of iterations as follows:

1. Set $CL(0) \leftarrow C_L[\mathcal{T}, m]$;
   $CU(0) \leftarrow C_U[\mathcal{T}, m]$;
   $I \leftarrow 1$.
2. If $I > k$, halt.
   Otherwise, set $C \leftarrow [CL(I-1) + CU(I-1)]/2$.
3. If $FFD[\mathcal{T}, C] \leqq m$, set $CU(I) \leftarrow C$;
   $CL(I) \leftarrow CL(I-1)$;
   $I \leftarrow I+1$;
   and go to 2.
4. If $FFD[\mathcal{T}, C] > m$, set $CL(I) \leftarrow C$;
   $CU(I) \leftarrow CU(I-1)$;
   $I \leftarrow I+1$;
   and go to 2.

The final value $CU(k)$ gives the smallest value of C found for which $FFD[\mathcal{T}, C] \leqq m$. Either the corresponding schedule has been generated along the way (and could have been saved, storage space permitting), or else it has not yet been generated because $CU(k) = C_U[\mathcal{T}, m]$. In either case, the schedule can now be generated by a single additional application of FFD, and this schedule is the output of MULTIFIT.

Before beginning our analysis of the worst case behavior of MULTIFIT, let us first examine how long it takes to run.

We first note that the application of FFD referred to in steps 3 and 4 need not be run to completion if $FFD[\mathcal{T}, C] > m$, but can be halted as soon as the first item

enters bin P_{m+1}. This, plus the fact that $\mathcal{T}$ is already in decreasing order, means that each application of FFD need only take $O(nm)$ steps (and can actually be implemented to take $O(n \log m)$ steps using techniques described in [5]). Thus the total time for MULTIFIT, including the initial sorting of $\mathcal{T}$ by size and the k iterations of FFD is $O((n \log n)+knm)$ or $O((n \log n)+(kn \log m))$, depending on implementation. This is to be compared to the time required to run the LPT algorithm of [2], [3]. LPT involves the same initial sorting step, followed by a packing procedure which is quite comparable to one execution of FFD's packing procedure, and also can be implemented to run in either $O(nm)$ or $O(n \log m)$ time. Thus LPT does have a slight advantage over MULTIFIT in timing when $k>1$. However, as we shall see, there is apparently no reason to choose $k>7$ in applying MULTIFIT, and thus for large values of n, both algorithms will have execution times dominated by the time for the initial sort, and hence will be comparable. Of course, for small values of n both algorithms are so fast as to make matters of relative timing purely academic.

We now turn to the question of how the two algorithms compare as to the quality of the schedules they produce. We have already presented in § 1 the main result about the worst-case behavior of LPT. For MULTIFIT, our main result is presented in Theorem 3.1, where $MF(k)$ stands for the version of MULTIFIT in which k is the specified number of iterations.

THEOREM 3.1. *For all* $m \geq 2$, $k \geq 0$, $R_m(MF[k]) \leq r_m + (\frac{1}{2})^k$.

Proof. Suppose m and k are such that $R_m(MF[k])>r_m+(\frac{1}{2})^k$. Recall from the definition of $R_m(A)$ in § 1 that this means there exists a $\mathcal{T}$ such that when $MF[k]$ is applied to $\mathcal{T}$ and m, the resulting schedule has finishing time exceeding $(r_m+(\frac{1}{2})^k)\cdot \mathcal{T}^*_m$. Since this finishing time cannot exceed the final value $CU(k)$, we thus have

$$CU(k) \geq (r_m+(\tfrac{1}{2})^k)\cdot \mathcal{T}^*_m. \tag{3.1}$$

Consider the final value $CL(k)$. By the process of binary search, $CU(k)-CL(k)=(\frac{1}{2})^k\cdot(C_U[\mathcal{T}, m]-C_L[\mathcal{T}, m]) \leq (\frac{1}{2})^k\cdot \mathcal{T}^*_m$, since $C_U[\mathcal{T}, m] \leq 2C_L[\mathcal{T}, m]$ and $C_L[\mathcal{T}, m] \leq \mathcal{T}^*_m$. Thus we must have

$$CL(k) \geq r_m \cdot \mathcal{T}^*_m. \tag{3.2}$$

But then, since $r_m>1$ for $m \geq 2$, this means that $CL(k)>C_L[\mathcal{T}, m]$. This implies that FFD must have been performed with bin capacity $CL(k)$ at some point during the operation of the algorithm, and yielded $FFD[\mathcal{T}, CL(k)]>m$. However, this is impossible as, in light of (3.2), it violates Lemma 2.1. □

Combining Theorem 3.1 with the bounds on r_m presented in Table 1, we present in Table 2 a comparison of the worst case behavior of $MF[k]$ and LPT for various values of m and k. The lower bound on $R_m(MF[k])$ for all k which is presented in the last row of the table follows from the same examples (Figs. 1, 2, and 3) used to provide lower bounds on r_m.

From the table it can be seen that $MF[5]$ has better worst-case behavior than LPT for all $m \geq 3$ and $MF[6]$ and $MF[7]$ are better for all $m \geq 2$. Further improvement is possible by choosing larger values of k, but for $k=7$ the bound is already quite close to the limiting value, except for the cases where $m \geq 8$ and our bounds on r_m are not tight. If our conjecture that $r_m = \frac{20}{17}$ for all $m \geq 8$ is true, then

TABLE 2
A comparison of worst case bounds

$m=$	2	3	4	5	10	50
$R_m(\text{LPT})=$	1.167	1.222	1.250	1.267	1.300	1.327
$R_m(MF[5])\leqq$	1.174	1.185	1.208	1.208	1.251	1.251
$R_m(MF[6])\leqq$	1.158	1.169	1.192	1.192	1.236	1.236
$R_m(MF[7])\leqq$	1.151	1.162	1.184	1.184	1.228	1.228
$R_m(MF[k])\geqq$	1.143	1.154	1.176	1.176	1.176	1.176

the upper bounds on $R_m(MF[k])$ for $m = 10$ and $m = 50$ would be the same as those for $m = 4$ and $m = 5$.

We remind the reader that the figures in Table 2, attractive as they are, are only *worst-case* bounds. In practice the algorithms can be expected to do much better. To get a feel for how much improvement might be expected, and how the algorithms might compare as to average-case behavior , three limited simulation tests were run, each covering 10 task sets with $m = 10$. In Test 1, each task set consisted of 30 tasks with sizes chosen independently according to a uniform distribution between 0 and 1. In Test 2 each task set again consisted of 30 tasks, this time with sizes being the sum of 10 independent choices from the uniform distribution, to simulate a normal distribution. For both these tests $C_L[\mathcal{T}, m]$ was taken as an estimate of $\mathcal{T}^*_m$, and since this could well have been an under-estimate, the values for $F_A[\mathcal{T}, m]/\mathcal{T}^*_m$ might be somewhat inflated. To circumvent this difficulty, in Test 3 we generated our task sets so that $\mathcal{T}^*_m$ was known in advance. We started with 10 tasks of size 1, divided each independently into 2, 3, or 4 subtasks, whose relative sizes were also randomly chosen. $\mathcal{T}$ consisted of the approximately 30 subtasks so constructed, and we automatically had $\mathcal{T}^*_{10} = 1$. (As a matter of curiosity, we might note that in no case did any of the algorithms reconstruct an optimum schedule, though all came close.)

Our results are summarized in Table 3, which for each test gives the average value of $F_A[\mathcal{T}, m]/\mathcal{T}^*_m$(or our possibly inflated estimate of it) for LPT, $MF[7]$, and $MF[7]'$, where the last-named algorithm is the earlier and inferior version of MULTIFIT described in [4].

TABLE 3
Average values of $F_A[\mathcal{T}, m]/\mathcal{T}^*_m$

TEST	1	2	3
LPT	1.074	1.023	1.051
$MF[7]$	1.024	1.023	1.016
$MF[7]'$	1.033	1.065	1.021

Due to the limited and somewhat arbitrary nature of these simulations, one should not be prepared to draw far-reaching conclusions from them. However, we might note that, although $MF(7)$ and LPT came out in a dead heat in Test 2, the improvement provided by $MF[7]$ over LPT is clear in the other two tests, where in fact $MF[7]$ found a better schedule in 19 out of the 20 cases, the remaining case being a tie. It also might be noted that $MF[7]$ always found its ultimate schedule by the 6th iteration, so that $MF[6]$ could just as well have been used, for a slight saving in running time.

4. Upper bound proofs: Preliminaries. The problem of proving an upper bound on r_m is considerably simplified if we can focus our attention on a "normalized" situation. In this section we show how this can be done, and thus set up a framework for such proofs. We illustrate this framework by using it directly to prove an easy upper bound of $r_m \leqq 5/4 = 1.250$ for all $m \geqq 2$. In the next section we show how more sophisticated analysis can yield a 1.220 general bound. The same basic framework can also be used as a starting point when proving the exact upper bounds for $m \leqq 7$ summarized above in Table 1.

In what follows we shall be dealing primarily with sets $\mathcal{T}$ of items which are ordered by size. It is therefore convenient to define formally a *list* to be an ordered set $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$ such that $l(T_1) \geqq l(T_2) \geqq \cdots \geqq l(T_n)$. For any pair of integers $p \geqq q$, we define a *(p/q)-counterexample* to be a list $\mathcal{T}$ and number M of bins such that

$$FFD(\mathcal{T}, p) > M \geqq OPT(\mathcal{T}, q).$$

Thus, though it is possible to pack the items of $\mathcal{T}$ into M bins of capacity q, FFD is unable even to pack them into M bins of the larger capacity p. It is easy to see that $r_m > p/q$ implies the existence of a (p/q)-counterexample with $M = m$.

A *minimal (p/q)-counterexample* consists of a list $\mathcal{T}$ and number M of bins such that all of the following hold:

(a) $\mathcal{T}$ and M form a (p/q)-counterexample;

(b) For all lists $\mathcal{T}'$ satisfying (a), $|\mathcal{T}'| \geqq |\mathcal{T}|$;

(c) For all m, $1 \leqq m < M$, $r_m \leqq p/q$.

Thus a (p/q)-counterexample is minimal if there exists neither a (p/q)-counterexample for the same number of bins having fewer items nor a (p/q)-counterexample for a smaller number of bins. It is not difficult to see that the existence of a (p/q)-counterexample implies the existence of a minimal (p/q)-counterexample. It follows that if $r_m > p/q$, there must exist a minimal (p/q)-counterexample having $M \leqq m$.

In a normalized proof of an upper bound r on r_m we assume the existence of a minimal (p/q)-counterexample, for convenient p and q satisfying $r = p/q$, and derive a contradiction from that assumption. In such proofs we will be able to use many general properties which must be obeyed by such a minimal counterexample. We now turn to the task of deriving some of these properties.

We shall assume for the rest of this section that the list $\mathcal{T}$ and number m of processors provide a minimal (p/q)-counterexample. Our first observation is an immediate consequence of the definitions.

LEMMA 4.1. *$\mathcal{T}$ and m must satisfy the following*:

(a) *$FFD(\mathcal{T}, p) = m+1$ and $OPT(\mathcal{T}, q) = m$*;

(b) *All items on $\mathcal{T}$, except the last, are assigned to the first m bins by FFD.*

In all that follows, we shall let $\mathcal{P} = \langle P_1, P_2, \cdots, P_{m+1}\rangle$ be the FFD packing of $\mathcal{T}$ into bins of capacity p and $\mathcal{P}^* = \langle P_1^*, P_2^*, \cdots, P_m^*\rangle$ be the optimal packing of $\mathcal{T}$ into bins of capacity q. For subsets X and Y of $\mathcal{T}$, we shall say that *X dominates Y* if there is a 1–1 mapping f: $Y \to X$ such that $l(y) \leqq l(f(y))$ for all $y \in Y$. Using this concept and the following lemma, we can draw some interesting conclusions about $\mathcal{P}$ and $\mathcal{P}^*$.

LEMMA 4.2 (cancellation lemma). *Let I, $J \subseteq \{1, 2, \cdots, m+1\}$ be such that $|I| = |J| = k > 0$. Then the set $X = \bigcup_{i\in I} P_i$ cannot dominate the set $Y = \bigcup_{j\in J} P_j^*$.*

Proof. Suppose X dominates Y, and let f: $Y \to X$ be the mapping involved. Consider the list $\mathcal{T}' = \mathcal{T} - X$ obtained by deleting the items of X from $\mathcal{T}$ while retaining the same relative ordering of the remaining items. Since we have deleted *exactly* those items which were contained in the bins P_i, $i \in I$, the FFD packing of $\mathcal{T}'$ will be identical to that for $\mathcal{T}$ except that those bins will be missing. Thus $FFD(\mathcal{T}', p) = m - k + 1$. Now consider the packing $\mathcal{P}^*$, and construct a new packing $\mathcal{P}'$ by interchanging each item $y \in Y$ with its image $f(y)$. Since $l(y) \leq l(f(y))$, we thus must have $l(P_i') \leqq l(P_i^*)$ for all $i \notin J$, $1 \leqq i \leqq m$. Moreover, bins P_j', $j \in J$, contain only items of X (although they may have $l(\mathcal{P}_j') > q$). Thus by deleting all elements of X from $\mathcal{P}'$, we obtain a packing of $\mathcal{T}'$ into at most $m-k$ bins of capacity q. Hence $OPT(\mathcal{T}', q) \leqq m-k$, and since $m-k < m$, this contradicts the presumed minimality of $\mathcal{T}$ and m. □

LEMMA 4.3. $|\mathcal{P}_i^*| \geqq 3$, $1 \leqq i \leqq m$.

Proof. Suppose first that $P_i^* = \{x\}$. Let j be such that $x \in P_j$. Then P_j dominates P_i^*, contrary to Lemma 4.2. Suppose next that $P_i^* = \{x, y\}$, where x precedes y in the (decreasing) ordering of $\mathcal{T}$. Suppose $x \in P_j$ and $y \in P_k$ in the FFD packing. If $j = k$, then P_j dominates P_i^*, a contradiction. Suppose $j < k$. Then the fact that y went to the right of the bin containing x, even though $l(x) + l(y) \leqq q < p$, means that P_j must have contained a second item z, in addition to x, when y was assigned. Thus z preceded y in $\mathcal{T}$ and so $l(z) \geqq l(y)$, and P_j dominates P_i^*, again a contradiction. Finally, suppose $j > k$. Then, since y follows x in $\mathcal{T}$, it cannot be the first item to go in P_k; call that first item z. Since z is the first item in a bin to the left of the one containing x, z must have preceded x in $\mathcal{T}$ and hence $l(z) \geqq l(x)$. Therefore P_k dominates P_i^*, giving us our final contradiction. □

LEMMA 4.4. $|P_i| \geqq 2$, $1 \leqq i \leqq m$.

Proof. Suppose $P_i = \{x\}$, for some i, $1 \leqq i \leqq m$. Then we must have $l(x) + l(T_n) > p$, where T_n is the last, and hence smallest, item in $\mathcal{T}$. But this means that $l(x) + l(y) > q$ for all $y \in \mathcal{T}$, so if P_i^* is the optimal bin containing x, $|P_i^*| = 1$, contrary to Lemma 4.3. □

The next lemmas obtain bounds on the item sizes. Let $\alpha = l(T_n)$ denote the size of the smallest item.

LEMMA 4.5. $\alpha > (m/(m-1))(p-q)$.

Proof. For $1 \leqq i \leqq m$, since T_n did not fit in P_i, we have $l(P_i) + \alpha > p$. Hence

$$\sum_{i=1}^{m} l(P_i) + m\alpha > mp.$$

However, since all items are contained in m bins of capacity q in $\mathcal{P}^*$,

$$\sum_{i=1}^{m} l(P_i)+\alpha < mq.$$

The lemma follows. □

LEMMA 4.6. *For all* $T_i \in \mathcal{T}$, $l(T_i) \leqq q-2\alpha$.

Proof. This follows immediately from Lemma 4.3, which implies that T_i must be in a bin with at least two other items in $\mathcal{P}^*$. □

At this point, we can already illustrate the use of these lemmas by proving an easy upper bound on r_m.

THEOREM 4.1. *For all* $m \geqq 2$, $r_m \leqq 5/4$.

Proof. Suppose $\mathcal{T}$ and m provide a minimal $(5/4)$-counterexample. By Lemma 4.5, $l(T_n)=\alpha>1$. Thus no optimal bin can contain more than three items. By Lemma 4.3 this means that $|P_i^*|=3$, $1 \leqq i \leqq m$, and hence $|\mathcal{T}|=3m$. By Lemmas 4.1 and 4.4, this means that there must be a P_i, $1 \leqq i \leqq m$, with $|P_i|=2$. Let $P_i=\{x, y\}$. Since T_n did not go in P_i, we must have $l(x)+l(y)+\alpha>5$. Thus by Lemma 4.3 we must have

$$5<2(q-2\alpha)+\alpha=8-3\alpha.$$

This implies $\alpha<1$, a contradiction. □

In order to prove stronger results, we will need to take a more detailed look at the FFD packing $\mathcal{P}$ of our minimal (p/q)-counterexample. Let us label the items of $\mathcal{T}$ according to their assigned locations in $\mathcal{P}$ as follows: If $|P_i|=k$, then the elements of P_i are denoted by $P_i[j]$, $1 \leqq j \leqq k$, in the order in which they were assigned to P_i. $P_i[1]$ is the first item assigned (the earliest in the ordering of $\mathcal{T}$), and so on. A bin P_i, $1 \leqq i \leqq m$, is a *k-bin* if it contained exactly k items when first an item was assigned to a bin to its right (i.e., when $P_{i+1}[1]$ was assigned). This is in distinction to a *k-item bin*, which is merely a bin P_i, $1 \leqq i \leqq m$, with $|P_i|=k$. The *base level* $b(P_i)$ of a k-bin P_i is defined as $\sum_{j=1}^{k} l(P_i[j])$, whereas the *final level*, or simply *level*, is just $l(P_i)$. If P_i is a k-bin, we call the items $P_i[j]$, $1 \leqq j \leqq k$, *regular items*, and all $P_i[j]$, $j>k$, are called *fallback items*. A *fallback k-bin* is a k-bin P_i such that $|P_i|>k$, that is, one that contains fallback items. A *regular k-bin* is one which contains no fallback items. (Observe that none of these definitions applies to bin P_{m+1}, a bin which, being last, is atypical and will not enter very strongly into our arguments.)

The final lemma of this section gives a list of properties that follow from these definitions and the manner in which FFD operates.

LEMMA 4.7. *In* $\mathcal{P}=\langle P_1, P_2, \cdots, P_{m+1}\rangle$,

(a) *If* $P_i[j]$ *is a regular item, then it precedes in* $\mathcal{T}$ *all* $P_{i'}[j']$ *with* $i<i' \leqq m+1$ *and* $1 \leqq j' \leqq |P_{i'}|$, *and all* $P_i[j']$ *with* $j<j' \leqq |P_i|$.

(b) *If* $1 \leqq k<k'$, *all k-bins are to the left of all k'-bins.*

(c) *If* $\{P_i: s \leqq i \leqq t\}$ *is the set of k-bins, then* $b(P_s) \geqq b(P_{s+1}) \geqq \cdots \geqq b(P_t) > (k/(k+1)) \cdot p$.

(d) *For each* $k>1$, *all regular k-bins are to the left of all fallback k-bins.*

Proof. Parts (a), (b), and (c) are all straightforward consequences of the fact that $\mathcal{T}$ is ordered by decreasing item size. We derive (d) from (b) and (c) by contradiction. Suppose P_i is a fallback k-bin, P_j is a regular k-bin, and $j>i$. Then,

since T_n, the smallest item, went in a bin to the right of P_j, we have $b(P_j)+l(T_n)=l(P_j)+l(T_n)>p$. But we also have $l(P_i[k+1])\geqq l(T_n)$ and, by (c), $b(P_i)\geqq b(P_j)$. Thus $l(P_i)\geqq b(P_i)+l(P_i[k+1])\geqq b(P_j)+l(T_n)>p$, a contradiction. □

We conclude from Lemma 4.1, Lemma 4.4, and Lemma 4.7, that if $\mathscr{P}$ is the FFD packing for a minimal (p/q)-counterexample, it consists of a (possibly vacuous) sequence of fallback 1-bins, followed by a (possibly vacuous) sequence of regular 2-bins, followed by a (possibly vacuous) sequence of fallback 2-bins, followed by a (possibly vacuous) sequence of regular 3-bins, and so on, with the last nonempty bin P_{m+1} containing the single item T_n, which is the last and hence smallest item on $\mathscr{T}$.

In § 5 we expand on this picture, using information derived from the particular values of $p=122$ and $q=100$ to derive a general bound $r_m\leqq 1.220$. More specific arguments, using values of m as well as those of p and q, are required when proving the exact upper bounds of Table 1.

5. The general upper bound.

THEOREM 5.1. $R_M\leqq 1.22$ *for all* $m\geqq 2$.

Proof. Suppose the theorem is false. Then there exists a minimal (122/100)-counterexample, provided say by $\mathscr{T}=\{T_1, T_2, \cdots, T_n\}$ and m. We first derive some constraints on the sizes of the items in $\mathscr{T}$.

By Lemma 4.5, we know that for some $\Delta>0$,

$$l(T_n)=22+\Delta. \tag{5.1}$$

Since T_n is the smallest item, this means that all items are at least this large. Furthermore, by Lemma 4.6 we can conclude that

$$l(T_i)\leqq 56-2\Delta, \qquad 1\leqq i\leqq n. \tag{5.2}$$

Our final observation gives us an upper bound on Δ. Suppose $\Delta>4$. Then every item must have size exceeding 26, and so no bin in $\mathscr{P}^*$, with its capacity of 100, can contain more than 3 items. By Lemma 4.3, this means that all bins in $\mathscr{P}^*$ contain *exactly* 3 items, and $n=3m$. On the other hand, if a bin P_i in $\mathscr{P}$, $1\leqq i\leqq m$, contained two or fewer items, we would have $l(P_i)+l(T_n)\leqq 2(56-2\Delta)+(22+\Delta)=134-3\Delta<122$, which would violate the FFD packing rule. Thus every one of the first m bins of $\mathscr{P}$ contains at least 3 items and so $n\geqq 3m+1$, a contradiction. We thus conclude that

$$0<\Delta\leqq 4. \tag{5.3}$$

The remainder of the proof consists of a weighting argument in which each item is assigned a "weight" based on its size and where it was placed in the FFD packing $\mathscr{P}$. This weighting will have the property that each P_i, $1\leqq i\leqq m$, will contain items whose total weight is *at least* $100-\Delta$. In addition, except for a very limited number of bins, in the optimal packing $\mathscr{P}^*$ each bin will contain weight *no more* than $100-\Delta$. A conservation-of-total-weight argument will then allow us to contradict the assumption that we had a counterexample.

We group the items of $\mathscr{T}$ into seven classes based on the structure of $\mathscr{P}$ as it was described at the end of the previous section. Since all items have size less than 56 by (5.2), there are no fallback 1-bins in $\mathscr{P}$. Since all items exceed 22 in size by

(5.1), there are no bins in $\mathcal{P}$ which contain more than 5 items. We classify items as to which of the remaining possible bin types contain them, and as to size. The observations made concerning item sizes will follow from the fact that $\mathcal{P}$ is an FFD packing, and $l(P_i) > 122 - l(T_n) = 100 - \Delta$ for all i, $1 \leqq i \leqq m$.

The two items in each regular 2-bin, except the last (rightmost) such bin, both exceed (100-Δ)/2 in size, and are *type*-X_2. If both items in the last regular 2-bin exceed (100-Δ)/2 in size, they are also *type*-X_2; otherwise they are both *type*-Z.

The two regular items in each fallback 2-bin must total at least 2(122)/3 in size, by Lemma 4.7(c), and are *type*-Y_2.

The three items in each regular 3-bin, except possibly the last one, all exceed (100-Δ)/3 in size and are *type*-X_3. If all three items in the last regular 3-bin exceed (100-Δ)/3 in size, they are also *type*-X_3; otherwise all three are *type*-Z.

The three regular items in each fallback 3-bin must total at least 3(122)/4 in size, and are *type*-Y_3.

The four items in each regular 4-bin, except possibly the last one, all exceed (100-Δ)/4 in size and are *type*-X_4. If all four items in the last regular 4-bin exceed (100-Δ)/4 in size, they are also *type*-X_4; otherwise all four are again *type*-Z.

The remaining items are all of size at least $22+\Delta$ and are *type*-X_5. These consist of T_n, all the fallback items in fallback 2- and 3-bins, and all the items in 5-item bins.

Now we are prepared to define our weighting function. The weight of an item depends on Δ, the item's type, and the item's size. These dependencies are described in Table 4, where l denotes the size of the item.

TABLE 4
Item weights $w(T_i)$

Item type	$0 < \Delta \leqq 12/5$	$12/5 < \Delta \leqq 4$
X_2	$50 - \frac{\Delta}{2}$	$50 - \frac{\Delta}{2}$
Y_2	$l - \Delta$	$l - \Delta$
X_3	$\frac{100}{3} - \frac{\Delta}{3}$	$\frac{100}{3} - \frac{\Delta}{3}$
Y_3	$l - \Delta$	$l - \Delta$
X_4	$25 - \Delta/4$	$25 - \Delta/4$
X_5	22	$25 - \Delta/4$
Z	l	l

Observe from the table that

$$w(T_i) \leqq l(T_i), \quad \text{for all } T_i \in \mathcal{T}. \tag{5.4}$$

Moreover, for any fixed range of Δ, the weight of a *type*-X_{i+1} item never exceeds the weight of a *type*-X_i item, $2 \leqq i \leqq 4$, and all items have weight at least 22.

For any set S of items, let $w(S)=\sum_{T_i\in S} w(T_i)$. We first show that $w(P_i)\geqq 100-\Delta$, $1\leqq i\leqq m$. Clearly this holds for bins with two *type-X_2* items, three *type-X_3* items, or four *type-X_4* items, and for all 5 item bins. Also, the (at most) three bins composed solely of *type-Z* items have this property since the sum of the item sizes in such a bin, as with all the other P_i, $1\leqq i\leqq m$, must exceed 100-Δ, and each *type-Z* item has weight equal to its size.

This leaves just the fallback 2-bins and 3-bins to be accounted for. A fallback 3-bin contains three *type-Y_3* items and one *type-X_5* item, since the three *type-Y_3* items total at least 3(122)/4 in size. Thus the total weight of such a bin is at least

$$\frac{3(122)}{4}-3\Delta+22=113.5-3\Delta>100-\Delta,$$

since $\Delta\leqq 4$ by (5.3).

A fallback 2-bin contains two *type-Y_2* items whose total size is at least 2(122)/3 and hence one *type-X_5* item. If $0<\Delta\leqq 12/5$ the total weight of such a bin is at least

$$\frac{2(122)}{3}-2\Delta+22=\frac{310}{3}-2\Delta>100-\Delta.$$

On the other hand, if $12/5<\Delta\leqq 4$, the total weight is at least

$$\frac{2(122)}{3}-2\Delta+25-\frac{\Delta}{4}=\frac{319}{3}-\frac{9}{4}\Delta>100-\Delta.$$

Thus $w(P_i)\geqq 100-\Delta$, $1\leqq i\leqq m$, and in fact we can conclude that

$$w(\mathscr{T})\geqq(100-\Delta)m+w(T_n). \tag{5.5}$$

Next we consider the optimal packing $\mathscr{P}^*$. We first claim that no optimal bin containing a *type-Y_2* or *type-Y_3* item can exceed 100-Δ in total weight. This is clear since the weight of such an element is Δ less than its size, by (5.4) no other item weighs *more* than its size, and by definition $l(P_i^*)\leqq 100$, $1\leqq i\leqq m$.

Next we consider the possible optimal bins that contain only *type-X_2*, *-X_3*, *-X_4*, and *-X_5* items. Recall that the sizes of such items are at least $(100-\Delta)/2$, $(100-\Delta)/3$, $(100-\Delta)/4$, and $22+\Delta$, respectively. Clearly no optimal bin can contain more than four of them and, by Lemma 4.3, each optimal bin must contain at least three items. Table 5 presents a partial enumeration of the conceivable configurations, for each stating whether it is permitted by the size constraints, and, if so, the maximum possible weight for such a bin. A configuration is not listed if its total size clearly exceeds that for some listed configuration which is "impossible," due to the size constraints, or if its total weight is trivially dominated by that for some permissible configuration already listed. In the table, a "—" in a configuration stands for *any type-X_i* item, $2\leqq i\leqq 5$.

In no case does the maximum total weight for a permissible configuration exceed $100-\Delta$. Thus any optimal bin whose total weight exceeds $100-\Delta$ must contain at least one of the (at most) 9 *type-Z* items. By (5.4) the total weight of any such optimal bin is at most 100, giving an "excess" weight of at most Δ. Since there are at most 9 such bins, we thus have

$$w(\mathscr{T})\leqq(100-\Delta)m+9\Delta. \tag{5.6}$$

TABLE 5
Upper bounds on $w(P_i^*)$ for bins with all type-X_i items

Configuration	$0<\Delta\leqq 12/5$	$12/5<\Delta\leqq 4$
X_2X_3—	Impossible	Impossible
$X_2X_4X_4$	$100-\Delta$	$100-\Delta$
$X_3X_3X_3$	$100-\Delta$	$100-\Delta$
X_2———	Impossible	Impossible
X_3X_4——	Impossible	Impossible
$X_3X_5X_5X_5$	$100-\frac{\Delta+2}{3}(0<\Delta\leqq 1/4)$ Impossible $(1/4<\Delta\leqq 12/5)$	Impossible
$X_4X_4X_4X_4$	$100-\Delta$	$100-\Delta$

Combining (5.5) and (5.6), we obtain

$$w(T_n)\leqq 9\Delta. \tag{5.7}$$

For $0<\Delta\leqq 12/5$, (5.7) implies that

$$w(T_n)\leqq 9\Delta\leqq 108/5<22,$$

a contradiction. When $12/5<\Delta\leqq 4$, there can be only 5 *type-Z* items, as the four items in the last regular 4-bin must all exceed $22+\Delta\geqq 25-\Delta/4$ and hence are *type-X_4* rather than *type-Z*. Thus in this case we must have

$$w(T_n)\leqq 5\Delta\leqq 20<22,$$

again a contradiction.

Having obtained contradictions in all cases, it follows that a minimal (122/100)-counterexample cannot exist, proving the theorem. □

The reader may have noted that there is a certain amount of slack left in the arguments of this proof, suggesting that better bounds could be proved using much the same methods. This is indeed the case, but all that seems possible is a very slight lowering of the bound—not enough to reduce it to 1.21, for instance. Since we conjecture that the right answer is $20/17=1.176\cdots$, we have settled for the 1.22 bound, judging that the additional effort and complication that might be introduced by an attempt to obtain such a slight improvement would not be justified.

REFERENCES

[1] K. R. BAKER, *Introduction to Sequencing and Scheduling*, John Wiley, New York, 1974.
[2] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[3] ———, *Bounds on the performance of scheduling algorithms, Computer and Job/Shop Scheduling Theory*, E. G. Coffman, ed., John Wiley, New York, 1976, Chap. 5.

[4] D. S. JOHNSON, *Fast allocation algorithms*, Proceedings, 13th Annual IEEE Symposium on Switching and Automata Theory, IEEE, New York, 1972, pp. 144–154.

[5] ———, *Near-optimal bin-packing algorithms*, Ph.D. thesis, Mathematics Dept., Mass. Inst. of Tech., Cambridge, 1974.

[6] D. S. JOHNSON, A. DEMERS, J. D. ULLMAN, M. R. GAREY AND R. L. GRAHAM, *Worst-case performance bounds for simple one-dimensional packing algorithms*, this Journal, 3(1974), pp. 299–326.

[7] S. SAHNI, *Algorithms for scheduling independent tasks*, J. Assoc. Comput. Mach., 23 (1976), pp. 116–127.

[8] J. D. ULLMAN, *Complexity of sequencing problems*, Computer and Job/Shop Scheduling Theory, E. G. Coffman, ed., John Wiley, New York, 1976, Chap. 4.

SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

MINIMEAN MERGING AND SORTING: AN ALGORITHM*

R. MICHAEL TANNER†

Abstract. A simple partitioning algorithm for merging two disjoint linearly ordered sets is given, and an upper bound on the average number of comparisons required is established. The upper bound is $1.06 \log_2 \binom{n+m}{m}$, where n is the number of elements in the larger of the two sets, m the number of the smaller, and $\binom{n+m}{m} = (n+m)!/(m!\,n!)$. An immediate corollary is that any sorting problem can be done with an average number of comparisons within 6% of the information theoretic bound using repeated merges; it does not matter what the merging order used is. Although the provable bound is 6% over the lower bound, computations indicate that the algorithm will asymptotically make only 3% more comparisons than the lower bound. The algorithm is compared with the Hwang-Lin algorithm, and modifications to improve average efficiency of this well known algorithm are given.

Key words. merging, algorithm, sorting, average, bound, insertion, probability, information

Introduction. Given two disjoint linearly ordered sets or lists, A and B, with elements

$$A_1 < A_2 < \cdots < A_m,$$

$$B_1 < B_2 < \cdots < B_n,$$

we are interested in merging them into one ordered set with $m+n$ elements by making pairwise comparisons between an element of A and an element of B. Since such a comparison can yield at most 1 bit of information, the noiseless coding theorem of information theory gives $\log_2 \binom{n+m}{m}$ as a lower bound on the number of comparisons required to perform this task, assuming that a priori all possible combinations of the elements in the final set are equally likely. This model is appropriate for the merges of a merge sort of a random list. In this paper we give an algorithm, which we will refer to as fractile insertion, whose average number of comparisons or work, $W_f(m, n)$, is upper bounded by $1.06 \log_2 \binom{n+m}{m}$. The fractile used to achieve this is the median.

The noiseless coding theorem will be a central facet of our argument, and its relevance warrants some review and explanation. In its standard form [1, p. 37], it states that if a source emits symbols from an alphabet $S_1, \cdots, S_k$ with probabilities $p_1, \cdots, p_k$ respectively, then the average length of any binary code used to encode the source must be greater than or equal to the source entropy, $H(p_1, \cdots, p_k) = \sum_{i=1}^{k} p_i \log_2 (1/p_i)$. In the present context we can regard each possible permutation of the elements as being a symbol, and for each merging

* Received by the editors April 13, 1977.

† Board of Information Sciences, University of California, Santa Cruz, Santa Cruz, California 95064.

problem nature chooses one of the combinations with uniform probability $= \binom{n+m}{m}^{-1}$. Since the result of a comparison of two elements can be coded as a 0 or 1, the outcomes of the comparisons specified by any given algorithm constitute a binary code for the source; therefore, the entropy, $\log_2 \binom{n+m}{m}$, must be a lower bound. But more importantly, the additivity of the entropy function on conditionally independent distributions enables it to serve as an effective measure of the quality of a comparison. A perfect comparison gives one bit of information; the average number of comparisons will be close to the lower bound if each of the comparisons yields close to one bit. In particular, our algorithm calls for the insertion of one element in the smaller list into the larger list, thus decomposing the problem into two smaller problems. For example, suppose A_f is inserted into B. Let the probability that A_f is less than all B_i be p_0, that it is greater than B_1 but less than B_2 be p_1, and so forth. We could write a recursion relation for the average number of comparisons required by the algorithm as

$$W_f(m, n) = (\text{Average number of comparisons required to insert } A_f)$$

$$+ \sum_{i=0}^{n} p_i (W_f(f-1, i) + W_f(m-f, n-i)), \tag{1}$$

since after locating A_f between B_i and B_{i+1}, the conditionally independent problems of merging $f-1$ into i and $m-f$ into $n-i$ remain.

On the other hand, determining the correct merging can be regarded as the two step process of first encoding the location of A_f in B, and then encoding for the two remaining subproblems. This interpretation permits the grouping property of the entropy function to be used to give a recursion for the information theory bound on $W_f(m, n)$ which closely resembles (1) (see [1, p. 8]). Let q_j, $j = 1, \cdots, \binom{n+m}{m}$, be the probability of the ith merge order in an exhaustive list arranged such that $q_{r_i+1}, \cdots, q_{r_i+1}$ are associated with the merge orders which have A_f between B_i and B_{i+1}. Since all merging orders are assumed equally likely $q_j = 1 \Big/ \binom{n+m}{m}$ for all j. By definition of the p_i, $p_i = \sum_{j=r_i+1}^{r_{i+1}} q_j$, $\left(r_0 = 0, r_{n+1} = \binom{n+m}{m}\right)$. The group property then permits the entropy of the complete merge to be written as the entropy of the partitioning induced by insertion of A_f plus the weighted sum of the entropies remaining in each partition:

$$H(q_1, \cdots, q_{\binom{n+m}{m}}) = H(p_0, p_1, \cdots, p_n) + \sum_{i=0}^{n} p_i H\left(\frac{q_{r_i+1}}{p_i}, \cdots, \frac{q_{r_{i+1}}}{p_i}\right).$$

Let

$$I(m, n) = H\left(\frac{1}{\binom{n+m}{m}}, \cdots, \frac{1}{\binom{n+m}{m}}\right)$$

be the information theory lower bound on the number of comparison needed to merge m into n. The equation then can be rewritten as

$$I(m,n)=H(p_0,p_1,\cdots,p_n)+\sum_{i=0}^{n} p_i[I(f-1,i)+I(m-f,n-i)]. \tag{2}$$

In other words, the total entropy is sum of the entropy of the distribution of A_f in B plus the weighted sum of the entropies which remain, given the position of A_f.

This identity, (2), stands independent of any algorithm and, recalling that the definition of p_i, $i=0,\cdots,n$, depends on f, holds for any f. The proof of the bound relies heavily on the close similarity between the bound identity (2) and the recursion relation (1): the essential ingredient is showing that the insertion of a fractile element A_f can be done in very little more than $H(p_0,p_1,\cdots,p_n)$ comparisons on the average.

An algorithm: Fractile insertion. Without loss of generality, m is assumed less than n. An element A_f is compared with a sequence B_{k_i} of elements from B until A_f has been located in the B order. The algorithm is then applied to the two smaller problems of merging $A_1,\cdots,A_{f-1}$ into $B_1,\cdots,B_i$ and $A_{f+1},\cdots,A_m$ into $B_{i+1},\cdots,B_n$, where $B_i<A_f<B_{i+1}$. If $A_f>B_n$ then $i=n$ and only the first problem remains. If $A_f<B_1$, then $i=0$ and only the second remains. The sequence of subscripts k_i is given as follows:

1. Let $k_1=\lceil n\cdot f/(m+1)\rceil$. Let $\alpha=\lfloor \frac{1}{2}\log_2(n(1+n/m))-1.3\rfloor$. Let $\Delta=2^{\alpha}$.
2. If $A_f>B_{k_i}$ then continue to set $k_{i+1}=k_i+\Delta$ until either $k_{i+1}>n$ or $A_f<B_{k_{i+1}}$. If $k_{i+1}>n$, binary insert A_f in $B_{k_i+1},\cdots,B_n$. Otherwise binary insert A_f in $B_{k_i+1},\cdots,B_{k_{i+1}-1}$.
3. If $A_f<B_{k_i}$ then continue to set $k_{i+1}=k_i-\Delta$ until either $k_{i+1}<1$ or $A_f>B_{k_{i+1}}$. If $k_{i+1}<1$, binary insert A_f in $B_1,\cdots,B_{k_i-1}$. Otherwise binary insert A_f in $B_{k_{i+1}},\cdots,B_{k_i-1}$.

Briefly, the first test is at the proportional position of A_f in B. The test position makes hops of size Δ, either up or down, depending upon the outcome of the first test, until A_f's location is known within Δ or less. Finally A_f is binary inserted. (See Fig. 1.)

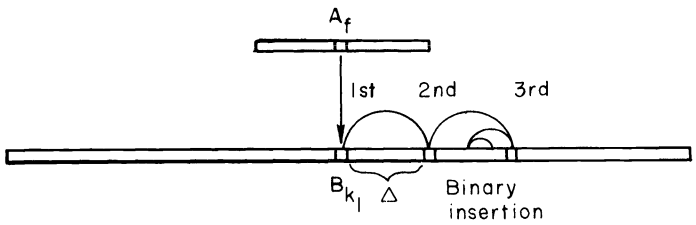


FIG. 1

When implemented as a recursive procedure, fractile insertion will require fewer than $\lceil X\rceil$ pointers to record the subdivisions, where X satisfies $m(f/(m+1))^X=1$. Note that for $f=\lceil m/2\rceil$, $X\approx\lceil\log_2 m\rceil$. Similarly, the amount of active store needed will depend on the particular fractile chosen.

As a simple example, suppose $f = \lceil m/2 \rceil$

$$A = \{1, 3, 8, 11\},$$

$$B = \{2, 4, 5, 6, 9, 10, 12, 16, 17, 19\}.$$

Then $k_1 = \lceil 10 \cdot \frac{2}{5} \rceil = 4$, $\alpha = \lfloor \frac{1}{2} \log_2 35 - 1.3 \rfloor = 1$, $\Delta = 2$. A_2 $(=3)$ is compared to B_4 $(=6)$; then B_2 $(=4)$. Since decreasing the B index by 2 again gives 0, A_2 is inserted by comparing with B_1 $(=2)$. The two subproblems which remain are $\{1\}$ into $\{2\}$, and $\{8, 11\}$ into $\{4, 5, 6, 9, 10, 12, 16, 17, 19\}$.

As heuristic motivation for the form of the algorithm consider the probability distribution of the location of A_f in the B list. To make the previous notation for the insertion probabilities more explicit, let $p_f(0; m, n)$ be the probability $A_f < B_1$, $p_f(n; m, n)$ the probability $A_f > B_n$, and $p_f(i; m, n)$ the probability $B_i < A_f < B_{i+1}$ for $0 < i < n$. Then

$$p_f(i; m, n) = \frac{\binom{i+f-1}{f-1}\binom{n-i+m-f}{m-f}}{\binom{n+m}{m}}, \qquad 0 \leqq i \leqq n. \tag{3}$$

By simple reorganization of the factorials, this can also be written as

$$p_f(i; m, n) = \frac{m}{m+n}\binom{m-1}{f-1}\frac{\binom{n}{i}}{\binom{m+n-1}{i+f-1}}. \tag{4}$$

For any fixed m, as $n \to \infty$, the distribution converges in law to a continuous Beta distribution of the form

$$p(X) = \frac{m!}{(f-1)!(m-f)!} X^{f-1}(1-X)^{m-f},$$

which can be seen most easily by observing that

$$\frac{(i+f-1)\cdots(i+1)(n-i+m-f)\cdots(n-i+1)}{(n+m)\cdots(n+1)} \sim \frac{1}{n+m/2}\binom{i+f/2}{n+m/2}\left\{\frac{n-i+[(m-f)/2]}{n+m/2}\right\}^{m-f}.$$

When $n \to \infty$ with m/n fixed, the distribution converges in law to a Gaussian with mean $((f-1)/(m-1)) \cdot n$, and variance $\frac{1}{4}n(1+n/(m-1))$. This can be derived either from the continuous Beta or directly from (3) using Gaussian approximations to the binomial distribution. In any event, the key to the success of the algorithm is that the search interval, Δ, is pegged to the standard deviation of the underlying distribution of $A_f(\Delta \approx \frac{3}{4}\sigma)$. Consequently, for large n, the initial comparisons are as efficient as on the smaller n cases, and the later comparisons are basically perfect binary insertions. As the latter begin to constitute the majority of the comparisons, over-all efficiency of each insertion becomes closer and closer to one.

Within this basic scheme there is some leeway in choosing the ratio of the search interval to the standard deviation. The term 1.3 in the expression for α was determined empirically to give the best balance between overshoot and undershoot for different n.

We will now show that when $f = \lceil m/2 \rceil$, representing the median, the average number of comparisons required by this algorithm is within 6% of the information theory bound. Hereafter, $p(i; m, n)$ denotes $p_{\lceil m/2 \rceil}(i; m, n)$. It is perhaps worth noting that, trivially, the optimum algorithm for merging one into two requires 5.2% more than the information theory bound. Thus, in a limited sense, no significantly tighter bound on all problems is possible for any algorithm. The actual worst case for the median insertion occurs at $m = 1$ and $n = 65$, where the average is 5.8% over the bound.

Since the proof is based on an intricate scaling argument, it is inconvenient to deal directly with the discrete distribution, (3). Instead we regard $p(i; m, n)$ as a function continuous in both i and n, but not m, by interpreting combinatorials of the form $\binom{x+j}{j}$ in (3), x real, j integer, as simply

$$\binom{x+j}{j} = \frac{(x+j)(x+j-1)\cdots(x+1)}{j!}.$$

By considering the number of comparisons the algorithm makes to insert $A_{\lceil m/2 \rceil}$ in position i to be defined for all real $i \in [-\frac{1}{2}, n+\frac{1}{2}]$, the average number of comparisons, $c(m, n)$, required by the algorithm to locate $A_{\lceil m/2 \rceil}$ can be viewed as a function continuous in n. The theorem is then proven based on the scaling properties of $c(m, n)$ and $h(m, n)$, the entropy of $p(i; m, n)$.

To validate the analytical proof for the actual discrete distribution, three types of potential error due to approximations implicit in this approach must be bounded. The first is approximation of the type

$$p(i; m, n) \approx \int_{i-\frac{1}{2}}^{i+\frac{1}{2}} p(i; m, n)\, di$$

which arises when a discrete sum over i is replaced by a continuous integral over i in calculating $c(m, n)$ and $h(m, n)$. The second is the possible interpolation error that occurs when the values of $c(m, n)$ and $h(m, n)$, computed only for integer n, are interpolated to define both for continuous n. Finally, there is the machine round-off and function error in the computation of the table for Region I (see proof of theorem). The first two sources of error are treated at the end of the proof. The third is deemed negligible, being of the order of 10^{-5} smaller than the margins allowed in the proof.

THEOREM. *Let $W_{\text{med}}(m, n)$ be the average number of comparisons required by the median insertion algorithm to merge m into n assuming all $\binom{n+m}{n}$ possible combinations are equally likely. Then $W_{\text{med}}(m, n) \leqq 1.06 \log_2 \binom{m+n}{m}$.*

Proof. The verification of the inequality has been carried out by dividing the table of values $W_{\text{med}}(m, n)$, $m \geqq 0$, $n \geqq 0$, into three regions. The first is a basis set

where the average number of comparisons and the efficiency of the insertion process can be evaluated numerically.

Region I: $m \leqq 40$, $n \leqq 100$. Using the above notation, (1) can be rewritten

$$W_{\text{med}}(m, n) = \sum_{i=0}^{n} p(i; m, n)(\text{comparisons made to locate } A_{\lceil m/2 \rceil} \text{ in position } i + W_{\text{med}}(\lceil m/2 \rceil - 1, i) + W_{\text{med}}(m - \lceil m/2 \rceil, n - i)).$$

This equation, along with $W_{\text{med}}(0, n) = 0$ for all $n \geqq 0$, defines $W_{\text{med}}(m, n)$. An ALGOL W program was written to calculate $W_{\text{med}}(m, n)$, as well as the entropy of the $p(i; m, n)$ distribution and the average number of comparisons required for the insertion. The results, when combined with the proofs used for Regions II and III, give the 1.06 $I(m, n)$ bound. For most m and n this region $W_{\text{med}}(m, n) \approx 1.03\, I(m, n)$.

Region II: $m \leqq 40$, $n > 100$. For fixed m we extend the bound by induction on n. Having calculated the efficiency of the remaining merges to be within the 1.06 bound, we need only show that the insertion efficiency satisfies

$$\frac{c(m, n')}{h(m, n')} < 1.06$$

for all $n' > 100$. The inductive strategy is to show for any n there exists an $n' \approx 2n$ such that efficiency for n implies efficiency for n'. Since we are allowing n to take on real values, this is equivalent to showing that for any n' there exists a corresponding n. Having computationally established the 6% bound for $40 \leqq n \leqq 100$, repeated use of this argument establishes the bound for all $n' > 100$.

We begin by comparing $p(i; m, n)$ with $p(i'; m, n')$ chosen to make the latter very closely the former scaled by a factor of 2.

Let $i' = ((f-1)/(m-1))n' + 2(i - ((f-1)/(m-1))n)$,

$$D_1 \triangleq \frac{d}{di} \ln \frac{p(i; m, n)}{p(i'; m, n')} = \sum_{k=1}^{f-1} \frac{1}{i+k} - \sum_{k=1}^{m-f} \frac{1}{n-i+k} - \sum_{k=1}^{f-1} \frac{1}{\frac{1}{2}\frac{f-1}{m-1}(n'-2n) + i + \frac{k}{2}} + \sum_{k=1}^{m-f} \frac{1}{\frac{1}{2}\frac{m-f}{m-1}(n'-2n) + n - i + \frac{k}{2}}$$

(5)

$$D_2 \triangleq \frac{d^2}{di^2} \ln \frac{p(i; m, n)}{p(i'; m, n')} = -\sum_{k=1}^{f-1} \frac{1}{(i+k)^2} - \sum_{k=1}^{m-f} \frac{1}{(n-i+k)^2} + \sum_{k=1}^{f-1} \frac{1}{\left(\frac{1}{2}\frac{f-1}{m-1}(n'-2n) + i + \frac{k}{2}\right)^2} + \sum_{k=1}^{m-f} \frac{1}{\left(\frac{1}{2}\frac{m-f}{m-1}(n'-2n) + n - i + \frac{k}{2}\right)^2}.$$

By making n' sufficiently large, it is clear that this second derivative, D_2, can be made negative. This will assure that the maximum of the ratio is achieved near the peaks of the distributions, where the first derivative, D_1, will be zero. The best comparison will be obtained by finding the smallest n' for which the second derivative is still negative for all i.

An upper bound to the second derivative can be obtained using an integral approximation to the sums. Specifically

$$\int_{i-\frac{1}{2}+\varepsilon}^{i+\frac{1}{2}+\varepsilon} \frac{1}{x^2}\,dx = \frac{-1}{i+\frac{1}{2}+\varepsilon} + \frac{1}{i-\frac{1}{2}+\varepsilon} = \frac{1}{i^2+2i\varepsilon+\varepsilon^2-\frac{1}{4}}.$$

If $\varepsilon > 1/(8i)$ the integral is less than $1/i^2$; if $\varepsilon = 0$ it is greater than $1/i^2$. Consequently

$$\sum_{k=1}^{f-1} \frac{1}{(i+k)^2} > \sum_{j=i+1}^{i+f-1} \int_{j=\frac{1}{2}+\varepsilon}^{j+\frac{1}{2}-\varepsilon} \frac{1}{x^2}\,dx = \int_{i=\frac{1}{2}+\varepsilon}^{i+f-\frac{1}{2}+\varepsilon} \frac{1}{x^2}\,dx = \frac{1}{(i+f-\frac{1}{2}+\varepsilon)(i+\frac{1}{2}+\varepsilon)}$$

for $\varepsilon > 1/(8(i+1))$, the largest value necessary for each of the parts. This bound can then be applied to the four sums in D_2.

$$-\sum_{k=1}^{f-1} \frac{1}{(i+k)^2} < -\frac{f-1}{\left(i+\frac{1}{2}+\frac{1}{8(i+1)}\right)\left(i+f-\frac{1}{2}+\frac{1}{i(i+1)}\right)} \triangleq B_1(i),$$

$$-\sum_{k=1}^{m-f} \frac{1}{(n-i+k)^2} < -\frac{m-f}{\left(n-i+\frac{1}{2}+\frac{1}{8(n-i+1)}\right)\left(n-i+m-f+\frac{1}{2}+\frac{1}{8(n-i+1)}\right)} \triangleq B_2(n-i),$$

(6)

$$\sum_{k=1}^{f-1} \frac{1}{\left(\frac{1}{2}\frac{f-1}{m-1}(n'-2n)+i+\frac{k}{2}\right)^2} < \frac{f-1}{\left(\frac{1}{2}\frac{f-1}{m-1}(n'-2n)+i+\frac{1}{4}\right)\left(\frac{1}{2}\frac{f-1}{m-1}(n'-2n)+i+\frac{f}{2}-\frac{1}{4}\right)} \triangleq B_3(i),$$

$$\sum_{k=1}^{m-f} \frac{1}{\left(\frac{1}{2}\frac{m-f}{m-1}(n'-2n)+n-i+k/2\right)^2} < \frac{m-f}{\left(\frac{1}{2}\frac{m-f}{m-1}(n'-2n)+n-i+\frac{1}{4}\right)\left(\frac{1}{2}\frac{m-f}{m-1}(n'-2n)+n-i+\frac{m-f}{2}+\frac{1}{4}\right)} \triangleq B_4(n-i),$$

$$D_2 < B_1(i)+B_2(n-i)+B_3(i)+B_4(n-i).$$

Leaving aside the cases $m = 1, 2$, and 3 for later attention, we now consider $m \geqq 4$. The sum of the four right-hand sides in (6) is a bound on D_2. The behavior of this bound is most easily discerned by grouping the $f-1$ terms, $B_1(i)$ and $B_3(i)$ and the $m-f$ terms, $B_2(n-i)+B_4(n-i)$. The form of $B_1(i)+B_3(i)$ is

$$B_1(i)+B_3(i)=\frac{ai+b}{c_1(i)}, \quad \text{where } c_1(i)\approx i^4.$$

(The reciprocal correction terms have been temporarily suppressed, since they do not affect the important behavior.) Similarly

$$B_2(n-i)+B_4(n-i)=\frac{c(n-i)+d}{c_2(n-i)}, \quad \text{where } c_2(n-i)\approx (n-i)^4.$$

Because of the $O(i^4)$ and $O((n-i)^4)$ terms in the denominators, $|B_1(i)+B_3(i)| > |B_2(n-i)+B_4(n-i)|$ for $i \leqq ((f-1)/(m-1))n$ whereas $|B_2(n-i)+B_4(n-i)| > |B_1(i)+B_3(i)|$ for $i \geqq n/2$. Consequently, by making $B_1(i)+B_3(i) \leqq 0$ for $i \leqq n/2$ and $B_2(n-i)+B_4(n-i) \leqq 0$ for $i \geqq ((f-1)/(m-1))n$, the sum, D_2, will be negative for all i.

For n' sufficiently large, all the numerator coefficients, a, b, c, and d, will be negative. For smaller n', $n' > 2n$, a and c are positive while b and d remain negative. It follows that if $B_1(i)+B_3(i) \leqq 0$ at $i = n/2$, it is negative for $i < n/2$. Likewise, $B_2(n-i)+B_4(n-i) \leqq 0$ for $i = ((f-1)/(m-1))n$ implies that it is negative for $i > ((f-1)/(m-1))n$.

Equating the denominators of $B_1(i)$ and $B_3(i)$ in (6) for $i = n/2$ gives the first required condition for n':

$$\left(\frac{1}{2}\frac{f-1}{m-1}(n'-2n)+i+\frac{1}{4}\right)\left(\frac{1}{2}\frac{f-1}{m-1}(n'-2n)+i+\frac{f}{2}-\frac{1}{4}\right)$$
$$\geqq \left(i+\frac{1}{2}+\frac{1}{8(i+1)}\right)\left(i+f-\frac{1}{2}+\frac{1}{8(i+1)}\right) \quad \text{at } i=\frac{n}{2}.$$

The denominators of $B_2(n-i)+B_4(n-i)$ give second condition

$$\left(\frac{1}{2}\frac{m-f}{m-1}(n'-2n)+n-1+\frac{1}{4}\right)\left(\frac{1}{2}\frac{m-f}{m-1}(n'-2n)+n-i+\frac{m-f}{2}+\frac{1}{4}\right)$$
$$\geqq \left(n-i+\frac{1}{2}+\frac{1}{8(n-i+1)}\right)\left(n-i+m-f+\frac{1}{2}+\frac{1}{8(n-i+1)}\right) \quad \text{at } i=\frac{f-1}{m-1}n.$$

Substituting in the value of i and simplifying, the first becomes

$$\frac{1}{4}(n'+1)(n'+m) \geqq \left(n+1+\frac{1}{2n}\right)\left(n+m+\frac{1}{2n}\right) \tag{7}$$

for m odd, and $n' = (n''-2n)(m-f)/(2(f-1))+2n$, where n'' satisfies

$$\frac{1}{4}(n''+1)(n''+m-1) \geqq \left(n+1+\frac{1}{2n}\right)\left(n+m-1+\frac{1}{2n}\right) \tag{8}$$

for m even. The second inequality requires that

$$(9)\quad \frac{1}{4}\left(n'+\frac{m-1}{2(m-f)}\right)\left(n'+m-1+\frac{m-1}{2(m-f)}\right)\geqq\left(n+1+\frac{1}{2n}\right)\left(n+m+\frac{1}{2n}\right).$$

Note that this is virtually identical to the n' required to double the search interval Δ.

For m odd, then, the maximum of the ratio, $(p(i;m,n))/(p(i';m,n'))$, must occur at $i=n/2$, $i'=n'/2$. For m even, the maximum occurs very near $i=((f-1)/(m-1))n$, as can be seen by consideration of the zero of the first derivative in (5). Rather than seeking the exact maximum, we wish to show that must be less than $(p(n/2;m,n))/(p(n'/2;m,n'))$. This is even a possibility only because, for even m, when $i=n/2$, $i'=((f-1)/(m-1))(n'-2n)+n<n'/2$. Of course, as m becomes larger, the slight asymmetry becomes less important.

Let $i''=(n'/n)i$. Let

$$R(m,n,n')\equiv\max_i\frac{p(i;m,n)}{p(i';m,n')}.$$

Now

$$\frac{p(i;m,n)}{p(i'';m,n')}=\frac{p(i;m,n)}{p(i';m,n')}\quad\text{at } i=\frac{f-1}{m-1}n.$$

Furthermore

$$\frac{d^2}{di^2}\left(\ln\frac{p(i;m,n)}{p(i'';m,n'')}-\ln R(m,n,n')\right)$$

$$=\sum_{k=1}^{f-1}\frac{1}{(i+k\cdot(n/n'))}2-\sum_{k=1}^{f-1}\frac{1}{(i+k)^2}+\sum_{k=1}^{m-f}\frac{1}{(n-i+k\cdot(n/n'))^2}$$

$$-\sum_{k=1}^{m-f}\frac{1}{(n-i+k)^2}>0.$$

Since this difference achieves a minimum of virtually zero near $i=((f-1)/(m-1))n$, it is not too difficult to show that it is positive at $i=n/2$. Thus

$$R(m,n,n')\leqq\frac{p(n/2;m,n)}{p(n'/2;m,n')}$$

for both even and odd m.

To evaluate $R(m,n,n')$ is now quite easy. Starting from (4), we have

$$R(m,n,n')<\frac{m+n'}{m+n}\frac{\binom{n}{n/2}\binom{n'+m-1}{n'/2+f-1}}{\binom{n+m-1}{n/2+f-1}\binom{n'}{n'/2}}.$$

Feller [2, pp. 179–182] gives a bounded approximation to the central binomial

coefficient, namely

$$\binom{n}{n/2} = 2^n \frac{2}{\sqrt{n}} \frac{1}{\sqrt{2\pi}} e^{-\varepsilon}, \quad \text{with} \quad \frac{1}{4n} - \frac{1}{20n^3} < \varepsilon < \frac{1}{4n} + \frac{1}{360n^3}. \tag{10}$$

Using this for the coefficients in the above inequality, we obtain

$$R(m, n, n') < \frac{m+n'}{m+n} \frac{\sqrt{n+m-1}}{\sqrt{n'+m-1}} \frac{\sqrt{n'}}{\sqrt{n}} \exp\left[\frac{1}{4}\left(\frac{1}{n} - \frac{1}{n'} - \frac{1}{n+m-1} + \frac{1}{n'+m-1}\right) + \delta\right]$$

where $|\delta| < 1/(5n^3)$ and can be neglected for $n \geqq 40$. This holds for m odd. For m even the right-hand side must be multiplied by a factor less than $((n'+1)/(n'+2))((n+2)/(n+1))$, to compensate for the slightly noncentral evaluation of two of the coefficients. For m odd (7) and (9) are the same. Using equality in (7) to define n' in the bound above gives

$$R(m, n, n') < 2\left(1 - \frac{m+3}{(n'+m-1)(n'+1)}\right)^{1/2} \left(1 + \frac{m + \frac{m+2}{2n}}{n(n+m+\frac{1}{2}n)}\right)^{1/2} \cdot \exp\left[\frac{1}{4}\left(\frac{1}{n} - \frac{1}{n'} - \frac{1}{n+m-1} + \frac{1}{n'+m-1}\right)\right].$$

Over the region $m \leqq 40$, $n \geqq 40$ this evaluates to

$$R(m, n, n') < 2(1.008).$$

For m even the upper bounding is slightly less tight because of the additional factor but still gives

$$R(m, n, n') < 2(1.022). \tag{11}$$

We thus have a very close comparison between $p(i; m, n)$ and the scaled distribution $2p(i'; m, n')$. Because of the negative second derivative (6), we know that $p(i; m, n) - 2p(i'; m, n')$ is positive in an interval containing $i = n/2$, negative in the tails and at most a probability of $\frac{1}{2}(0.022)$ moves from the central region to the tails.

If the scaling were perfect, and the search interval Δ' corresponding to n' were exactly 2Δ, both the entropy and the average number of comparisons would be increased by one, implying perfect incremental efficiency. Our next object is to show that the actual case is not sufficiently different to destroy the 6% bound.

Let $w(i; m, n)$ be the number of comparisons made by the algorithm in locating $A_{\lceil m/2 \rceil}$ in the ith position in B. In what follows we will assume that for the n' defined by (7), (8), and (9), $\Delta' = 2\Delta$, so that $w(i'; m, n') = w(i; m, n) + 1$. This will be the case at all but a few n' near the discontinuities in Δ as a function of n.[1]

[1] A simplified expression has been used in the algorithm for defining α and Δ. This leads to discrepancies of less than 1 in the location of the discontinuities of $\Delta'/2$ and Δ; they can be accounted for within the framework of interpolation of the computed values of $c(m, n)$.

Consider the first variation of $c(m,n)-h(m,n)$, labeled δV, due to an additive variational function $\delta p(i)$:

$$\delta V \triangleq \int \frac{d}{dp}[(w(i;m,n)+\log_2 p(i;m,n))p(i;m,n)]\,\delta p(i)\,di,$$

$$\delta V = \int (w(i;m,n)+\log_2 p(i;m,n)+\log_2 e)\,\delta p(i)\,di,$$

$$\delta V < \int_{\delta p(i)>0} \max_{\delta p(i)>0} (w(i;m,n)+\log_2 p(i;m,n)+\log_2 e)\,\delta p(i)\,di$$

$$+ \int_{\delta p(i)<0} \min_{\delta p(i)<0} (w(i;m,n)+\log_2 p(i;m,n)+\log_2 e)\,\delta p(i)\,di.$$

Since any $\delta p(i)$ which preserves a probability density satisfies

$$\int \delta p(i)\,di = 0 \qquad \left(\text{implying } \int_{\delta p(i)>0} \delta p(i)\,di + \int_{\delta p(i)<0} \delta p(i)\,di = 0\right),$$

$$\delta V < \int_{\delta p(i)>0} \delta p(i)\,di\Big[\max_{\delta p(i)>0} (w(i;m,n)+\log_2 p(i;m,n)$$

$$- \min_{\delta p(i)<0} (w(i;m,n)+\log_2 p(i;m,n))\Big].$$

The $\delta p(i)$ of interest are those which carry $p(i;m,n)$ into $2p(i';m,n')$. By the previous analysis the region $\delta p(i)>0$ is in the tails, whereas $\delta p(i)<0$ is an interval about the center of the distribution. Using the upper bound

$$\frac{\left|i-\frac{f}{m+1}n\right|}{\Delta}+2+\log_2\Delta$$

for $w(i;m,n)$ in the maximum and the lower bound

$$\frac{\left|i-\frac{f}{m+1}n\right|}{\Delta}+1+\log_2\Delta$$

in the minimum yields

$$\delta V < \int_{\delta p(i)>0} \delta p(i)\,di\Big[1+\max_{j\geq k\geq (f/(m+1))n}\Big(\frac{j}{\Delta}+\log_2 p(j;m,n)-\frac{k}{\Delta}$$

$$-\log_2 p(k;m,n)\Big)\Big].$$

Because $(d^2/di^2)(-\log_2 p(i;m,n))$ increases monotonically from its central value as i goes to the tails, the growth of $-\log_2 p(i;m,n)$ is greater than quadratic. Consequently,

$$\delta V < \int_{\delta p(i)>0} \delta p(i)\,di\Big[1+\max_{j>0}\Big(\frac{j}{\Delta}-\frac{1}{2}\gamma j^2\Big)\Big] \tag{12}$$

where γ is $(d^2/di^2)(-\log_2 p(i; m, n))$ evaluated at the central point. The maximum occurs at $j = 1/(\gamma\Delta)$ and has value $\frac{1}{2}(1/(\gamma\Delta^2))$. Using a convex variational path of the form $p_\alpha(i; m, n) = (1-\alpha)p(i; m, n) + \alpha 2p(i'; m, n')$, $0 \leqq \alpha \leqq 1$, will guarantee that, from the sums in (5) as bounded in (6),

$$\gamma > 16\frac{m-1}{n'(n'+m)}\left(\frac{n'}{n'+1}\right)^2 \log_2 e.$$

Now $\Delta = \frac{1}{2}\Delta' \geqq \frac{1}{2}(1/4.93)\sqrt{n'(n'+m)/m}$. Substituting into (12) with $n' > 100$ we find that

$$\delta V < \int_{\delta p(i)>0} \delta p(i)\, di\left[1 + \frac{1}{2}\frac{4(4.93)^2}{16(\log_2 e)}\frac{m}{m-1}\left(\frac{101}{100}\right)^2\right]. \tag{13}$$

Using (11) we obtain

$$\delta V < \tfrac{1}{2}(0.022)(2.44) < 0.03. \tag{14}$$

This is sufficient to enable completion of the proof for this region.

The assumption $m \geqq 4$ can now be removed as follows: For $m = 1$ the distribution is uniform for all n and thus by setting $n' = 2n + 1$, perfect scaling is achieved. For $m = 2$, we set $n' = 2n + \frac{3}{2}$. Since $i' = \frac{1}{3}n' + 2i - \frac{2}{3}n$,

$$\frac{p(i; m, n)}{p(i'; m, n)} = \frac{n+1-i}{(n+1)(n+2)} \cdot \frac{(n'+1)(n'+2)}{n'+1-i} = \frac{(n+\frac{7}{4})(n+\frac{5}{4})}{(n+2)(n+1)} \cdot 2,$$

a constant, for $i \in [-\frac{1}{2}, n+\frac{1}{2}]$. Compared to perfect scaling, a probability less than

$$\left(\frac{(n+\frac{7}{4})(n+\frac{5}{4})}{(n+2)(n+1)} - 1\right) \cdot 1 = \frac{3}{16}\frac{1}{(n+1)(n+2)}$$

has been moved to the region $i' \in [2n+\frac{3}{2}, 2n+2]$. For $m = 2$, $\frac{2}{3}n/\Delta < 10$, and so the most this could increase the average number of comparisons for $n \geqq 40$ is

$$10 \cdot \frac{3}{16}\frac{1}{(41)(42)} = 0.0011$$

which is stronger than the 0.03 of (14). Finally, for $m = 3$ the general argument holds but the tighter 1.008 figure applicable for m odd must be used in (14) (see (11)). Specifically, (14) becomes

$$\delta V < \frac{1}{2}(0.008)\left(\frac{m}{m-1}2.44\right) = \frac{1}{2}(0.008)\left(\frac{3}{2} \cdot 2.44\right) < 0.03.$$

We have shown that for any $1 \leqq m \leqq 40$ and any $n \in [40, 100]$ there exists an n', the minimum n' satisfying (8) and (9), which gives almost perfect scaling by a factor of 2. From the computed values of $h(m, n)$ and $c(m, n)$, for the

corresponding n'

$$\frac{c(m,n')}{h(m,n')} \leqq \frac{c(m,n)+1.03}{h(m,n)+1} < 1.06.$$

However, the relationship between n and n' is continuous, monotone, and invertible. Consequently for any $100 < n' < 200$ there exists a corresponding n and the above inequality obtains. All bounds used grow tighter as n increases, and therefore, by repeated use of the same scaling argument

$$\frac{c(m,n)}{h(m,n)} < 1.06 \quad \text{for all } 1 \leqq m \leqq 40, \quad n > 100.$$

Since the insertions are all adequately efficient, it follows immediately from recursive formulas for the work and the information theory bound, (1) and (2), that $W_{\lceil m/2 \rceil} < 1.06\, I(m,n)$ for all $1 \leqq m \leqq 40$, $n \geqq 1$.

Region III: $n \geqq m > 40$. Again we need only show that all insertions which have not been computed are efficient. For this region the underlying principle is very simple: for any merge in this parameter range, $p(i; m, n)$ is effectively a Gaussian distribution; the efficiency of the insertion is affected only by the ratio between the search interval Δ and the standard deviation, σ, of the Gaussian distribution. In computing the cases $m = 39$ and 40, $40 \leqq n \leqq 100$, we have exhausted all possible ratios allowed by the definition of Δ, as well as seeing the maximum effect of the m even asymmetry. Thus to understand an insertion with parameters m' and n', we need only look at the $m = 39$ and 40 cases with the same Δ/σ ratio.

The form of the proof is basically the same as that for Region II, although here, rather than using an induction with each step adding a scale factor of 2, we make the comparison directly with the computed case, using whatever power of 2 is necessary to give correct scaling down to the computed range. Specifically, for any $40 < m' \leqq n'$ we find a $40 \leqq n \leqq 100$ and an integer $k \geqq 0$ such that

$$\frac{(m'+n'-1)(n'+1)}{(m'-1)} = 4^k \frac{\left(m-1+n+\frac{1}{2n}\right)\left(n+1+\frac{1}{2n}\right)}{(m-1)} \tag{15}$$

with $m = 39$ or 40.

This amounts to finding a computed case with $\sigma'^2 = 4^k \sigma^2$. Note that, by choosing k properly, an n in the range 40 to 100 satisfying (15) can always be found. In almost all cases this leads to $\Delta' = 2^k \Delta$. (As before, at the discontinuities of Δ as a function of n, interpolation of the computed data is required.)

We then wish to show that with $i' = ((f'-1)/(m'-1))n' + 2^k(i - ((f-1)/(m-1))n)$

$$\frac{p(i; m, n)}{p(i'; m', n')} \approx 2^k.$$

The derivation is exactly analogous to that for Region II:

$$\frac{d^2}{di^2}\ln\frac{p(i;m,n)}{p(i';m',n')}$$

$$<-\frac{f-1}{\left(i+\frac{1}{2}+\frac{1}{8(i+1)}\right)\left(i+f-\frac{1}{2}+\frac{1}{8(i+1)}\right)}$$

$$-\frac{m-f}{\left(n-i+\frac{1}{2}+\frac{1}{8(n-i+1)}\right)\left(n-i+m-f+\frac{1}{2}+\frac{1}{8(n-i+1)}\right)}$$

$$+\frac{f'-1}{\left(\frac{1}{2^k}\frac{f'-1}{m'-1}n'-\frac{f-1}{m-1}n+i+\frac{1}{2^{k+1}}\right)\left(\frac{1}{2^k}\frac{f'-1}{m'-1}n'-\frac{f-1}{m-1}n+i+\frac{f'-\frac{1}{2}}{2^k}\right)}$$

$$+\frac{m'-f'}{\left(\frac{1}{2^k}\frac{m'-f'}{m'-1}n'-\frac{m-f}{m-1}n+(n-i)+\frac{1}{2^{k+1}}\right)\left(\frac{1}{2^k}\frac{m'-f'}{m'-1}n'-\frac{m-f}{m-1}n+(n-i)+\frac{m'-f'+\frac{1}{2}}{2^k}\right)},$$

Again we choose n' large enough to guarantee that the sum of the $f-1$ term and the $f'-1$ term is negative for $i \leqq n/2$, and the sum of the $m-f$ term and the $m'-f'$ term are negative for $i \geqq n/2$.[2] (Note that because $f' \geqq f$ and $m'-f' \geqq m-f$, the constant terms in the denominators of the primed terms must be relatively larger than in the Region II proof, which has only the effect of making the entire second derivative more negative off center.) Substituting $i = n/2$, we need

$$\frac{f-1}{\left(\frac{n}{2}+\frac{1}{2}+\frac{1}{8(n/2+1)}\right)\left(\frac{n}{2}+f-\frac{1}{2}+\frac{1}{8(n/2+1)}\right)} \geqq \frac{f'-1}{\left(\frac{1}{2^k}\frac{f'-1}{m'-1}n'+\frac{1}{2^{k+1}}\right)\left(\frac{1}{2^k}\frac{f'-1}{m'-1}n'+\frac{f'-\frac{1}{2}}{2^k}\right)},$$

$$\frac{(n'+1)(n'+m'-1)}{m'-1} \geqq 4^k\frac{\left(n+1+\frac{1}{2n}\right)\left(n+m-1+\frac{1}{2n}\right)}{m-1},$$

where we have assumed $2(f-1)=m-1$ and $2(f'-1)=m'-1$. The $m-f$ and $m'-f'$ lead to the same inequality. With equality, this is, of course, equation (15).

[2] For simplicity we will deal explicitly only with m' odd in this region. The even cases require minor adjustments analogous to those for Region II. As the effect there was greatest at $m=4$, it is here much smaller and, in any event, less than the $m=40$ asymmetry.

The central ratio again gives a bound on the amount of probability moving from the center to the tails. From (4) and (10),

$$\frac{p(n/2;m,n)}{p(n'/2;m',n')} \leqq \frac{m}{m+n}\frac{m'+n'}{m'}\frac{\binom{m-1}{f-1}}{\binom{m'-1}{f'-1}}\frac{\binom{n}{n/2}}{\binom{n'}{n'/2}}\frac{\binom{n'+m'-1}{n'/2+f'-1}}{\binom{n+m-1}{n/2+f-1}}$$

$$\leqq \frac{m}{m+n}\frac{m'+n'}{m'}\frac{\sqrt{m'-1}\sqrt{n'}\sqrt{n+m-1}}{\sqrt{m-1}\sqrt{n}\sqrt{n'+m'-1}}$$

$$\cdot \exp\left[\frac{1}{4}\left(-\frac{1}{m-1}-\frac{1}{n}+\frac{1}{n+m-1}+\frac{1}{m'-1}+\frac{1}{n'}-\frac{1}{n'+m'-1}\right)\right].$$

Terms less than $1/(20\,m^3)$ in the exponent have been dropped. Equation (15) can then be used in the first factor.

$$\frac{m}{m+n}\frac{\sqrt{n+m-1}}{\sqrt{m-1}\sqrt{n}}\frac{m'-1}{m'}\frac{m'+n'}{m'+n'-1}\sqrt{\frac{n'}{n'+1}}\left(\frac{\sqrt{n'+1}\sqrt{m'+n'-1}}{\sqrt{m'-1}}\right)$$

$$= 2^k\frac{m}{m-1}\frac{((n+m-1)(n+m-1+1/(2n)))^{1/2}}{m+n}\left(\frac{n+1+1/(2n)}{n}\right)^{1/2}$$

$$\cdot\frac{m'-1}{m'}\frac{m'+n'}{m'+n'-1}\sqrt{\frac{n'}{n'+1}}.$$

With $m = 39$ the ratio can be made largest by letting m', n', and n go to infinity.

$$\frac{p(n/2;m,n)}{p(n'/2;m',n')} \leqq 2^k\left(1+\frac{1}{m-1}\right)\exp\left(-\frac{1}{4(m-1)}\right) \leqq 2^k\left(1+\frac{3}{4}\frac{1}{m-1}\right) \leqq 2^k(1.02).$$

The variational argument leading to (12) again holds. Here

$$\gamma > 4\frac{(m-1)\log_2 e}{(n+1+1/(2n))(m+n-1+1/(2n))}, \qquad \Delta > \left(\frac{n(n+m)}{m}\right)^{1/2}\frac{1}{4.93}.$$

The bound (14) becomes

$$< \frac{1}{2}(0.02)\left[1+\frac{(4.93)^2}{\log_2 e}\frac{(n+1+1/(2n))(m+n-1+1/(2n))}{4n(n+m)}\frac{m}{m-1}\right]$$

$$< \frac{1}{2}(0.02)(3.2) = 0.032.$$

This is the maximum amount of waste over perfect scaling that occurs for any m' and n'. Thus

$$c(m',n')-h(m',n') < c(39,n)-h(39,n)+0.032.$$

For all $40 \leqq n \leqq 100$ the right side is less than 0.256. Since the n corresponding to m' and n' lies in the range 40 to 100, $c(m',n')-h(m',n') < 0.256$ for all $40 < m' \leqq n'$. We have actually computed all cases $m \leqq 50$, $n \leqq 100$. All unverified

cases have $h(m', n') > 4.35$. Therefore

$$\frac{c(m', n')}{h(m', n')} < 1.06 \quad \text{for all } 40 < m' \leqq n'$$

and, moreover, since the entropy goes to infinity as n' goes to infinity,

$$\frac{c(m', n')}{h(m', n')} \sim 1.$$

As before, this implies that

$$W_{\lceil m/2 \rceil}(m, n) < 1.06 I(m, n) \quad \text{all } 40 < m \leqq n.$$

The only remaining detail is the use of $p(i; m, n)$ as continuous function of both i and n when it is actually discrete. For any fixed m and n the error is bounded using a Taylor series expansion in the ith interval

$$\left| p(i; m, n) - \int_{i-1/2}^{i+1/2} p(j; m, n)\, dj \right| < \frac{1}{16} \max_{i-1/2<j<i+1/2} \left| \frac{d^2}{di^2} p(i; m, n) \right|_{i=j}.$$

Now

$$\frac{d^2}{di^2} p(i; m, n) = p(i; m, n)\left(\frac{d^2}{di^2} \ln p(i; m, n) + \left(\frac{d}{di} \ln p(i; m, n) \right)^2 \right).$$

Because this second derivative is negative in the center and positive in the tails, the effective central ratio for the actual discrete distributions is slightly larger than for the continuous approximation. However, using the expressions in (6) and recognizing that at the center $(d/di) \ln p(i; m, n) = 0$, the error in the ratio is less than

$$\frac{\frac{1}{16} p(n/2; m, n) \frac{d^2}{di^2} \ln p(i; m, n) \Big|_{i=n/2}}{p(n/2; m, n)} < \frac{m-1}{4n(n+m)} < 0.003$$

for all unverified cases.

Similarly, in implicitly defining an n, $40 \leqq n \leqq 100$, which corresponds to a larger n' we have tacitly assumed that the values of $c(m, n) - h(m, n)$ computed numerically for integer m and n can be accurately interpolated in n. By bounding $(\partial^2/\partial n^2)(c(m, n) - h(m, n))$, the interpolation error in any interval can be shown to be less than $\frac{1}{4}\sqrt{m}/((n+m)n)$, again negligible. This completes the proof.

In essence, the large subdividing insertions are done very efficiently, and they require only a small fraction of the total number of comparisons to be made. Consequently the algorithm's over-all performance is really determined by performance on the short merges, and this has been calculated.

Discussion of the result. Our concern here has been with an average number of comparisons measure. Although the demonstrated upper bound is within 6% of the information theory lower bound, for most problems the average number required is within 3% of $I(m, n)$. By the very nature of the measure, there is every reason to believe that the true asymptote is about 3%. Since any large problem gets broken into a spectrum of smaller problems, there is a blending and

smoothing effect which will utimately eliminate fluctuations in performance.

While establishing this tight bound on the average number of comparisons is of interest by itself, additional insights into the problem is provided by the method used:

First, large problems can be done with the almost same efficiency as small problems. Thus if some specialized hardware or a super algorithm were developed to give excellent speed on merges with say, $m < 10$, our algorithm enables that speed to be reflected in problems of arbitrary size.

Secondly, large problems can be done well only if small problems can be done well. Although it may seem trivial, this converse statement is in some ways more subtle than the original. Conceivably, any algorithm that decomposes the problem from the beginning is subdividing the tree of possibilities in a way that prevents "the perfect algorithm" from working. However, our technique breaks the problem down to problems of $m = k$ using a fraction roughly equal to $(1+\frac{1}{2}\log k)/k$ of the total work. To make $m = 2000$ requires only 0.003 of the total comparisons. Even ignoring the fact that the number of bits of information gained is virtually the same as the number of comparisons, this is so small as to be almost negligible. Therefore, given that this decomposition information is essentially free, "the perfect algorithm" has to be able to do $m \approx 2000$ problems well or it cannot do larger problems well either.

Although for the sake of proving the bound the median element, $A_{\lceil m/2 \rceil}$, was inserted, any fractile element can be used. The insertion technique is effective when the distribution is normal with variance $-\frac{1}{4}n(1+n/m)$. Using Gaussian approximations to the binomial coefficients in (4), it can be shown that this is the variance of the limiting distribution of every fractile. The only problem is when the element to be inserted is too near the end of the list for the normal approximation to be accurate. The first comparison of A_f should be with B_{k_1}, $k_1 = \lceil n \cdot f/(m+1) \rceil$. So long as

$$\min(k_1, n-k_1) > \frac{3(n+m)}{2\sqrt{m}}$$

the insertion should be very efficient. Near the ends, the distribution is more accurately given by an exponential, requiring a slightly different testing strategy. The discussion of the Hwang–Lin algorithm will return to this question.

As with any sophisticated technique that aims at minimization of one measure, there is the danger that the computation of the best next step dominates the useful computation. There is some leeway in the choice of α and Δ that should help this problem, however. Rather than calculating $\alpha = \lfloor \frac{1}{2}\log_2(n(1+n/m)) - 1.3 \rfloor$ and $\Delta = 2^\alpha$, one can instead straightforwardly set

$$\Delta = \text{ROUND}\left(\frac{n+m}{4\sqrt{m}}\right)$$

and, if convenient, use a binary insertion procedure that favors the middle. This combination has been shown by computation to be as good and perhaps better on small problems, and its asymptotic properties should be similar. It should also be noted that Δ is computed only once for every $2+\alpha$ comparisons on the average.

Even though the number of comparisons required by the fractile insertion algorithm can vary from problem to problem, the average measure used here by no means represents an unnaturally skewed weighting of the comparison tree nodes. For example, the assumption that both A and B have elements drawn from a uniform distribution on [0, 1] should not greatly affect the efficiency of the algorithm in performing the merge. On the other hand, the algorithm is not very good if the minimax (minimizing the maximum number of comparisons required) criterion is used.

Minimax properties. An exhaustive investigation of small problems reveals that the maximum number of comparisons required by the fractile insertion algorithm can be as much as 25% to 30% over the information theory bound. In contrast to the Hwang–Lin algorithm, the region where the difference is most pronounced is near $n \approx m$. The explanation for this weakness on approximately equal lists is easily found, however. It is known that for $m \geqq 6$([4, § 5.3.2]),

$$M(m, m+d) = 2m + d - 1, \qquad 0 \leqq d \leqq 4$$

where $M(m, n)$ is the lower bound for the best minimax algorithm merging m into n; and it is conjectured that the equality holds for any fixed d as $m \to \infty$. If this is true, then there is no difference in the maximum work that may have to be done if an (m, m) problem is subdivided by the fractile algorithm into two balanced problems or two slightly unbalanced problems. That is:

$$M(\lceil m/2 \rceil - 1, \lceil m/2 \rceil) + M(m - \lceil m/2 \rceil, m - \lceil m/2 \rceil)$$
$$= M(\lceil m/2 \rceil - 1, \lceil m/2 \rceil - d) + M(m - \lceil m/2 \rceil, m - \lceil m/2 \rceil + d) \quad \text{for } |d| \leqq 4.$$

Thus, in effect, the comparisons spent by the algorithm in determining the actual d do it no good against the minimax criterion. On the other hand, the $I(m, n)$ function does have the necessary convexity and, relative to the average metric, the same comparisons do pay off.

For unequal lists, the fractile insertion and the Hwang–Lin algorithms appear to have about the same maximum number of comparisons.

Sorting. It is well known that binary insertion and straight two way merging when used as sorting techniques are both asymptotically efficient, requiring only $n \log_2 n$ comparisons as $n \to \infty$. The underlying reason is that both are repeated merging schemes designed so that all the merging problems encountered can be done optimally by the associated merging procedures: Binary insertion is optimal for large n; the standard "pop the top", or two way merge ([4, p. 160]), is optimal when $m \approx n$. There may be times, however, when these list size constraints present awkward data handling problems. The implication of our theorem, combined with the noiseless coding theorem, is that any repeated merge sorting scheme based on the fractile insertion algorithm will require less than $1.06 \log_2 (n!)$ comparisons on the average to sort n elements. The proof is simply that every merge uses less than 1.06 times the corresponding information theory bound; by the noiseless coding theorem, the information theory bound for the whole problem is just the sum of the bounds for each part. But the average number of comparisons is likewise just the sum of the parts, and the result follows.

The Hwang–Lin algorithm. The best merging algorithm previously known for handling problems with ratios of m/n in the range between 0 and 1 is the generalized binary algorithm introduced by F. K. Hwang and S. Lin [3]. It has been shown to give minimax performance far superior to either straight binary insertion or "pop the top" merging. To use our information theory bound reference, the minimax upper bound is at worst only about 15% over $I(m, n)$. Although we have as yet been unable to prove a bound for arbitrarily large lists, on small problems the Hwang–Lin algorithm appears to be about as good if not slightly better than the fractile insertion algorithm even in terms of the average measure. In a way, it is actually fractile insertion adapted to the ends.

Briefly, the steps of their algorithm are:

1. Assuming $m \leqq n$, let $\alpha = \lfloor \log_2 (n/m) \rfloor$ and $x = n - 2^\alpha + 1$.
2. Compare A_m with B_x. If $A_m > B_x$, binary insert A_m in $B_{x+1}, \cdots, B_n$. Emit elements $A_m, B_{y+1}, \cdots, B_n$, where $B_y < A_m < B_{y+1}$, and apply the algorithm to $A_1, \cdots, A_m$ and $B_1, \cdots, B_y$.
3. If $A_m < B_x$, emit $B_x, \cdots, B_n$ and apply the algorithm to $A_1, \cdots, A_m$ and $B_1, \cdots, B_{x-1}$.

Given this recursive definition, it is easy to establish a recursion relation for the average number of comparisons required. This function was evaluated by computer for $m \leqq 25$, $n \leqq 100$. To summarize the results of a comparison of this function with the corresponding one for median insertion over the same region, neither one is uniformly better. Median insertion is slightly better for some problems, Hwang–Lin for others. On the average (that is, on a problem chosen at random), the Hwang–Lin is about half a percent better. The unresolved question is whether it continues to be so for large lists.

Suppose the fractile insertion algorithm were used with $f = m$. The first comparison would be between A_m and B_k,

$$k = \left\lceil n \cdot \frac{m}{m+1} \right\rceil = \left\lceil n \cdot \left(1 - \frac{1}{m+1}\right) \right\rceil \approx n + 1 - \lfloor n/m \rfloor \quad \text{for large } n \text{ and } m.$$

If $\Delta > \lfloor n/m \rfloor$ and the first comparison indicates $A_m > B_k$, A_m will be binary inserted in $B_{n+1-\lfloor n/m \rfloor + 1}, \cdots, B_n$. Otherwise it will be compared with $B_{n+1-\lfloor n/m \rfloor - \Delta}$. It does roughly the same thing as the Hwang–Lin method.

A closer look at distribution of location of A_m in B will show why this type of algorithm should give good average results. Let $p(0)$ be the probability $B_n < A_m$ and $p(i)$ the probability $B_{n-i} < A_m < B_{n-i+1}$.

$$\frac{p(i)}{p(i+1)} = \frac{\binom{n-i+m-1}{m-1} \Big/ \binom{n+m}{m}}{\binom{n-i-1+m-1}{m-1} \Big/ \binom{n+m}{m}} = \frac{(n-i+m-1)\cdots(n-i+1)}{(n-i-1+m-1)\cdots(n-i)}$$

$$= \frac{n-i+m}{n-i}.$$

So long as $i \ll n$, $p(i)/p(i+1) \approx (n+m)/n$. Thus an excellent approximation to $p(i)$ is

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

an exponential distribution. When $n = m$, for example, $p(0) = \frac{1}{2}$. It is not surprising that Hwang–Lin does well on equal length lists. Most of the comparisons are close to perfect! In general the probability that a first comparison of A_m with B_{n-i} will show $B_{n-i} < A_m$ is $q = \sum_{k=0}^{i} p(i) = 1-(n/(n+m))^i$, and the numbers of bits thereby obtained is $-q \log_2 q - (1-q) \log_2 (1-q)$. Similar expressions involving sums can be derived for the binary insertion comparisons. Although the $\alpha - \lfloor \log_2 (n/m) \rfloor$ and $x = n - 2^\alpha + 1$ do a good job of balancing probabilities in most cases, the efficiency of some comparisons can be as low as 80%. Indeed, an improvement can be made by simplifying the algorithm: The first comparison probability is bracketed by

$$1-\left(\frac{n}{n+m}\right)^{n/(2m)} < q \leqq 1-\left(\frac{n}{n+m}\right)^{n/m}.$$

The lower sequence as $n = m, 2m, 3m, \cdots$ is

$$1-\sqrt{1/2}, 1-\sqrt{4/9}, \cdots \to 1-1/\sqrt{e}$$

and the upper is

$$1-1/2, 1-4/9, \cdots \to 1-1/e.$$

Except at extremely large ratios, the second sequence stays closer to 1/2. The first comparison will be more efficient if the index in the B list is closer to $n+1-(n/m)$ than $n+1-(n/2m)$.

Also it should be noted that a unique property of an exponential is that it is meaningful to talk about a "half-life". In the present context, the critical "half-life" is that of the probability distribution, and as we have seen, its value is approximately $\Delta \approx n/m$. But the implication is that if the first comparison is efficient and $A_m < B_{n+1-\Delta}$, the next test may as well be at $B_{n+1-2\Delta}$. This suggests the following modification to the Hwang–Lin algorithm:

1. Assuming $m \leqq n$, let $\Delta = \text{ROUND}\ (n/m)$ and $x = n+1-\Delta$.
2. Compare A_m with B_x. If $A_m > B_x$, insert A_m in $B_{x+1}, \cdots, B_n$ using a binary procedure that favors the larger index values. (If $n-x$ is odd and the middle element is unique, use it; if even, use the middle with highest index in B as the comparison point.) Emit $A_m, B_{y+1}, \cdots, B_n$, where $B_y < A_m < B_{y+1}$, and apply the algorithm to $A_1, \cdots, A_{m-1}$ and $B_1, \cdots, B_y$.
3. If $A_m < B_x$, emit $B_x, \cdots, B_n$, set $n = x-1$ and then set $x = x-\Delta$. Return to 2.

The linear testing pattern of fractile insertion is very naturally appropriate here. Quite precisely, this is fractile insertion with a Δ adjusted to the tail distribution. Unless there is some difficulty in implementing the more general binary insertion, this modified version should run faster on the average than the original Hwang–Lin algorithm. Not only does it eliminate the taking of a base 2 logarithm and the exponentiation, it should actually make fewer comparisons. In addition, Δ will be calculated less than half as often.

Unfortunately, because the comparisons prescribed by the Hwang–Lin algorithm are not uniformly efficient, it appears difficult to prove the asymptotic

quality of its performance on the average. A proof by scaling, similar to the one given for Region II, would seem to be the path of least resistance. On the other hand, the modified version is better balanced, and it appears that a 6% bound could be established based simply on the efficiency of the worst comparison.

Conclusions. We have exhibited a merging algorithm based on fractile insertion which requires on the average less than 6% more comparisons than the information theoretic lower bound, $\log_2 \binom{n+m}{m}$. Moreover, $\log_2 \binom{n+m}{m}$ is not even the tightest lower bound (See [4, p. 194]). Given the tightness of the upper bound the algorithm establishes, any algorithm which is noticeably more efficient than fractile insertion (or the modified fractile insertion-Hwang–Lin algorithm above) will most probably be impractically complex. Indeed, at one point in our study, we calculated the performance of a very sophisticated median insertion algorithm; it actually computed the sequence of testing points that would best balance the tree. The reduction in the number of comparisons was unimpressive, less than 1% for a typical problem.

As we have mentioned, our result implies that it is possible to sort with an average number of comparisons close to the optimum. While this was known previously, there is now the additional freedom of knowing that the order of merging does not substantially affect the required number of comparisons.

Finally, the fractile insertion algorithm provides a highly efficient technique for decomposing a large merging problem into many separately processable smaller problems. As such it may make possible efficient parallelism in merging.

REFERENCES

[1] R. B. ASH, *Information Theory*, Interscience, New York, 1965.

[2] W. FELLER, *An Introduction to Probability Theory and Its Applications*, vol. I, John Wiley, New York, 1963.

[3] F. K. HWANG AND S. LIN, *A simple algorithm for merging two disjoint linearly ordered sets*, this Journal, 1 (1972), pp. 31–39.

[4] D. E. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

SIMPLE GÖDEL NUMBERINGS, ISOMORPHISMS, AND PROGRAMMING PROPERTIES*

MICHAEL MACHTEY,† KARL WINKLMANN† AND PAUL YOUNG†

Abstract. Restricted classes of programming systems (Gödel numberings) are studied, where a programming system is in a given class if every programming system can be translated into it by functions in a given restricted class. For pairs of systems in various "natural" classes, results are given on the existence of isomorphisms (one-to-one and onto translations) between them from the corresponding classes of functions. The results with the most computational significance concern polynomial time programming systems. It is shown that if $\mathcal{P} = \mathcal{NP}$ then every two polynomial time programming systems are isomorphic via a polynomial time computable function. If $\mathcal{P} \neq \mathcal{NP}$ this result points the way to the possible existence of "natural" but intractable computational problems concerning programming systems. Results are also given concerning the relationship between the complexity of certain important and commonly used properties of programming systems (such as effective composition of programs) and the complexity of translations into the systems.

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

1. Introduction and preliminaries. Rogers (1958) first defined and characterized acceptable Gödel numberings (programming systems). He did this in terms of the ability to translate arbitrary programs effectively between such systems. The primary result of this work was the isomorphism theorem, which states that between any two acceptable Gödel numberings there is an effective, one-to-one, and onto translation. Rogers' work was purely recursion-theoretic in spirit, with no regard for the computational complexity of translations or for program syntax (save the requirement that the set of valid programs be recursive). Subsequently, techniques for studying computational complexity and program syntax have been developed.

Schnorr (1975) has studied programming systems into which all programming systems can be translated effectively with no more than a linear growth in the Gödel numbers of the programs (i.e. by adding no more than a constant amount to the length of programs); he calls such systems optimal Gödel numberings. His main result is an isomorphism theorem which states that between any two optimal Gödel numberings there is an effective, one-to-one, and onto translation giving at most linear growth in the Gödel numbers of programs translated in either direction. Though this clearly represents an extension of Rogers' work, no attention was paid to the computational complexity of translations. Of special interest in Schnorr's paper are constructions (therein credited to the referee) for obtaining one-to-one translations into any acceptable programming system from an s_1^1 function for that system, and for obtaining isomorphisms with no greater rate of growth than those of the original (one-to-one) translations, a considerable improvement over Rogers' construction.

Hartmanis and Baker (1975) have defined and studied complexity classes of

* Received by the editors August 10, 1976, and in revised form April 13, 1977.

† Department of Computer Science and Department of Mathematics, Purdue University, West Lafayette, Indiana 47907. This work was supported in part by the National Science Foundation under Grants GJ27127A1 and MCS-76-09212.

programming systems: if C is a class of computable functions, they define the class GNC to consist of all those acceptable Gödel numberings into which every Gödel numbering can be translated by some function in C. Thus they consider GNPrefix, GNReg, GNLBA as the classes of acceptable programming systems into which all programming systems can be translated by prefix, regular, LBA-computable functions, respectively. Also, for any total computable function t they define GNC_t to be the class of programming systems into which translations can always be made by functions in the complexity class C_t (functions computable within resource bound $t(x)$ for all but finitely many x). The results given by Hartmanis and Baker and the questions they raise about classes of programming systems are of two general types: the first concerns the complexity of isomorphisms between members of a given class, the second concerns the complexity of certain important programming properties (such as fixed point functions as in the recursion theorem) in systems of a given class.

With regard to the existence of isomorphisms between acceptable Gödel numberings, Hartmanis and Baker show that between any two prefix (postfix) programming systems there is an isomorphism given by a particularly simple type of regular mapping, which they call restricted regular maps. They also show that for any complexity measure there are arbitrarily large total recursive functions t such that GNC_t is the closure of any member of GNC_t under the isomorphisms in C_t: between any two systems in GNC_t there is an isomorphism in C_t. They then raise the question of whether any "natural" classes of programming systems have such a closure property. Specifically, they conjecture that the classes of restricted regular, LBA, and polynomial time computable programming systems are closed under isomorphisms of the appropriate type, but that the class of regular systems is not. They also state a result, which they credit to R. Constable, that the class GNEXP of exponential time programming systems which they call GNPTIME (translations computable in time 2^{cn} for some constant c for programs of length n) is closed; they indicate a proof using the isomorphism construction from Rogers (1958). However, as we shall see in Proposition 2.3, the indicated proof cannot be correct.

In § 2 we give additional results on isomorphisms between programming systems, "almost" establishing all of Hartmanis' and Baker's conjectures as well as giving a correct proof of the result mentioned above on GNEXP (which they call GNPTIME). There is a restricted regular isomorphism between any two restricted regular programming systems, but uninterestingly so because every restricted regular Gödel numbering is in fact a prefix Gödel numbering. GNReg is not closed under regular isomorphisms, but GNLBA and GNEXP are closed under LBA and exponential time computable isomorphisms, respectively. And finally, between any two polynomial time programming systems there is an isomorphism computable in polynomial time *from a set* in $\mathcal{NP}$.

With regard to the complexity of certain programming properties such as s_1^1 functions and fixed point functions for recursion in systems of a given class, Hartmanis and Baker (1975) have shown that a programming system is a member of one of the "natural" classes which they consider if and only if it has an s_1^1 function in the corresponding class of computable functions. They have also shown that if a programming system is a member of one of the "natural" classes, then that system has a fixed point function (as in the recursion theorem) in the

corresponding class of computable functions. Their first result expands on the well-known result, implicit in Rogers (1958), that a universal programming system (one with an effective universal function) is acceptable if and only if it has an effective s_1^1 function; the second employs a standard construction of a fixed point function from an s_1^1 function. Hartmanis and Baker go on to show the existence of arbitrarily complex optimal Gödel numberings, and as a corollary to the first result above, the existence of optimal Gödel numberings with only complex s_1^1 functions. Using a different proof, they show the existence of optimal Gödel numberings with only complex fixed point functions. *If* there were a proof that any universal programming system with an effective fixed point function is in fact acceptable, *then* standard complexity-theoretic techniques would have probably shown the result as a corollary to the existence of complex optimal Gödel numberings. But as we shall show in Theorems 3.6 and 3.9, there is no such proof because a universal programming system can have an effective fixed point function but not be acceptable, and arbitrarily complex acceptable programming systems can have very simple fixed point functions.

The Hartmanis and Baker construction of complex optimal Gödel numberings uses a diagonalization technique; in § 3 we give an alternative "complexity-theoretic" construction and proof which yields a slightly stronger result, and also gives a rather different intuitive approach to this type of question. In § 3 we also give results which show that a universal programming system is acceptable if and only if it has an effective function for program composition and which in addition relate the complexity of the system to the complexity of the function for composition. These results strongly suggest that it is both more natural and more reasonable to *define* acceptable programming systems as universal programming systems with an effective function for composition.

We now give some basic definitions and notation which we shall use in this paper. We shall generally consider *programs* to be natural numbers represented in binary, though all the results and most of the proofs in this paper hold equally well if programs are considered to be finite strings over a finite alphabet (with at least two letters). In those few cases where it makes a slight difference we shall point out how to handle both notions. Similarly, we shall think of programs as operating either on binary integers or strings, whichever is more convenient. If x is a binary integer or a string, $|x|$ will denote its length. Strings will be assumed to be *ordered*, first by length and then lexicographically among those of the same length. Since in the general case it is always taken that the set of valid programs is recursive, and in practical cases valid programs are recognizable in less than cubic time, we shall use the convenient fiction that all integers or strings are valid programs by using the simple convention that any string or integer which is not a valid program is deemed to compute the empty (nowhere-defined) function.

We assume the reader is familiar with standard notation and terminology of recursive function theory, and we direct the reader to Rogers (1967) as a reference in this regard. We also assume some familiarity with abstract computational complexity theory, for which Hartmanis and Hopcroft (1971) is an appropriate reference.

Following Rogers (1958), we define a *Gödel numbering* (*programming system*) φ to be a function from the set of programs *onto* the partial recursive functions of one argument; we denote the image of the program i under such a

mapping by φ_i. Note that there are very simple coding and decoding functions which make it possible to interpret arbitrary strings or integers as n-tuples of strings or integers, respectively, and so such a mapping also induces mappings of the set of programs onto the partial recursive functions of n arguments for each positive integer n; we shall generally take these induced functions for granted. A programming system φ is *universal* if the partial function Φ such that $\Phi(i, x) = \varphi_i(x)$ for all i and x is a partial recursive function. A universal programming system φ is *acceptable* if there is a total recursive function t such that $\varphi_{t(i)} = \psi_i$ for all programs i, where ψ is some chosen standard Gödel numbering such as one based on Turing machines or ALGOL 60 programs.

If φ and ψ are programming systems, then a total function t is a *translation* of φ into ψ if $\varphi_i = \psi_{t(i)}$ for all programs i. Thus a universal Gödel numbering is a programming system which can be translated effectively into our standard programming system, and an acceptable Gödel numbering is a universal programming system into which our standard programming system can be translated effectively.

Following Hartmanis and Baker (1975), for any class C of total recursive functions we define GNC, the set of *C-computable programming systems*, to be the set of all acceptable programming systems into which all acceptable programming systems can be translated by functions in C. A *prefix* function is one which simply appends a given prefix to any argument; similarly for *postfix*. A *regular* function is one computed by a deterministic finite state machine which either has one symbol lookahead or has a special endmarker appended to every input string; note that such an assumption is necessary to get a reasonable notion of regular mappings. A *restricted regular* map is a regular function which on every argument either appends a given prefix to the argument or deletes another given prefix from the argument. A function is LBA-*computable* if it is computed by a deterministic Turing machine operating in space (tape) bounded by a linear function in the lengths of inputs. A function is *computable in exponential time* if it can be computed in any of the standard models of computation (such as Turing machines) in time bounded by 2^{cn} for a constant c and all inputs of length n. And a function is *polynomial time computable* if it can be computed in any of the standard models in time $n^c + c$ for a constant c and all inputs of length n. Then GNPrefix, GNPostfix, GNReg, GNRReg, GNLBA, GNEXP, and GN$\mathcal{P}$ will stand for the prefix, postfix, regular, restricted regular, LBA-computable, exponential time computable, and polynomial time computable programming systems, respectively.

Let φ be a universal programming system. A total function s of two arguments is an s_1^1 function for φ if for some acceptable Gödel numbering φ^2 of the partial recursive functions of two arguments we have $\varphi_i^2(x, y) = \varphi_{s(i,x)}(y)$ for all i, x, and y. It is well known that a universal Gödel numbering is acceptable if and only if it has an effective s_1^1 function. As Hartmanis and Baker (1975) have observed, if s is an s_1^1 function for φ then translations of programming systems into φ can always be gotten by functions t of the form $t(i) = s(e, i)$ for some fixed e, and if φ is in GNC then φ has an s_1^1 function gotten by composing a function in C with a function which is prefix in its second argument. Thus for all classes C which are closed under composition with prefix functions, every programming system in GNC has an s_1^1 function in C (when considered as a function in its second argument).

Finally, for the sake of completeness we close this section by giving the construction from the paper by Schnorr (1975) for obtaining one-to-one translations into programming systems as instances of an s_1^1 function. Let φ be an acceptable Gödel numbering with recursive s_1^1 function s, and let t be any total recursive function. With a standard application of the recursion theorem, we can effectively find an e such that for all i and x

$$\varphi_{s(e,i)}(x) = \varphi_e^2(i, x) = \begin{cases} 0 & \text{if } s(e, j) = s(e, i) \text{ for some } j < i, \\ 1 & \text{if } s(e, j) \neq s(e, i) \text{ for all } j < i \text{ and} \\ & \quad s(e, j) = s(e, i) \text{ for some } i < j \leqq x, \\ \varphi_{t(i)}(x) & \text{otherwise.} \end{cases}$$

For any such e, $s(e, i)$ is one-to-one as a function of i, because if for some k the set $S_k = \{i\colon s(e, i) = k\}$ would contain more than one element then we would have $0 = \varphi_{s(e,j)}(j) = \varphi_{s(e,i)}(j) = 1$ for $i = \min S_k$ and $j \in S_k$ with $i \neq j$. It follows that $\varphi_{s(e,i)} = \varphi_{t(i)}$ for all i.

2. Complexity of isomorphisms. We begin this section by establishing two of the conjectures on closure under isomorphisms made by Hartmanis and Baker (1975). The first is that there is a restricted regular isomorphism between any two restricted regular programming systems, but uninterestingly so because in fact:

PROPOSITION 2.1. *Every restricted regular Gödel numbering is a prefix Gödel numbering.*

Proof. The proof exploits the fact that there are prefix programming systems in which there are arbitrarily long, simple, and "meaningless" prefixes which can be added to programs. Specifically, let φ^2 be any acceptable Gödel numbering of the partial recursive functions of two arguments, and define $\varphi_0 = \varnothing$ (the empty function) and $\varphi_{10^i j}(x) = \varphi_i^2(j, x)$ where i is a natural number, j is an integer written in binary, and $10^i j$ is the result of prefixing a 1 and i 0's to j. Then it can easily be verified that $\varphi \in$ GNPrefix and that there are infinitely many i such that for all j, $\varphi_j = \varphi_{10^i j}$. For example, to see the latter let $\theta^2(j, x) = \varphi_j(x)$ for all j and x. Then there are infinitely many i such that $\varphi_i^2 = \theta^2$, and for each such i

$$\varphi_j(x) = \theta^2(j, x) = \varphi_i^2(j, x) = \varphi_{10^i j}(x)$$

for all j and x.

Let $\psi \in$ GNRReg and let t be a restricted regular translation of φ into ψ; that is $\psi_{t(j)} = \varphi_j$ for all j. Let d be the prefix deleted by t, let a be the prefix added by t, and let i be such that $i > |d|$ and $\varphi_j = \varphi_{10^i j}$ for all j. Then $t(10^i j) = a10^i j$, for otherwise $t(10^i j)$ would not be a binary integer. Let π be any acceptable programming system and let p be a prefix translation of π into φ; i.e. $\varphi_{pj} = \pi_j$ for all j. Then

$$\pi_j = \varphi_{pj} = \varphi_{10^i pj} = \psi_{t(10^i pj)} = \psi_{a10^i pj}$$

for all j, and thus $a10^i p$ is a prefix translation of π into ψ. Therefore φ is in fact a prefix programming system.

If we consider programs to be strings over a finite alphabet Σ then the proof is similar, but even simpler. Let $\varphi \in$ GNPrefix and define $\theta_{cx} = \varphi_x$ for all $c \in \Sigma$, then

$\theta \in$ GNPrefix. Let $\psi \in$ GNRReg and let t be a restricted regular translation of θ into ψ with d and a as above. If the first letter in d is b then $t(cx) = acx$ for all $c \neq b$ in Σ and all x. Then if π is any acceptable programming system and p is a prefix translation of π into φ we have

$$\pi_x = \varphi_{px} = \theta_{cpx} = \psi_{t(cpx)} = \psi_{acpx}$$

for all $c \neq b$ in Σ and all x. Thus acp is a prefix translation of π into ψ. □

Next we show that regular Gödel numberings need not be isomorphic under regular mappings. In fact, we shall show that no postfix programming system can be translated *onto* a prefix programming system by any function from an even wider class. Let us say that a function t from programs to programs is *front-to-back* if there is a function f from natural numbers to natural numbers such that for all n and programs x, the first n characters of $t(x)$ are determined by the first $f(n)$ characters of x; clearly all regular maps are front-to-back.

PROPOSITION 2.2.[1] *No postfix Gödel numbering can be translated onto a prefix Gödel numbering by a function which is front-to-back.*

Proof. The proof is a formalization of the intuition that in a postfix system the "meaningful" part of long programs can be at the end, while in a prefix system the "meaningful" part of some long programs is at the beginning; thus any front-to-back, *onto* function must necessarily take some "meaningless" code from the beginning of some postfix programs to some incorrect, "meaningful" code for some prefix programs.

Let $\varphi \in$ GNPrefix and $\psi \in$ GNPostfix. Let p and q be such that $\varphi_{pi}(x) = 0$ and $\psi_{iq}(x) = 1$ for all i and x; p and q exist since by Hartmanis and Baker (1975) φ and ψ have s_1^1 functions which are, respectively, prefix and postfix in the second argument. Let t be any onto front-to-back function and let f be such that the first n characters in $t(x)$ are determined by the first $f(n)$ characters in x. Take any j such that $f(|p|) < |j|$ and $t(j) = pi$ for some i; since t is onto, infinitely many such j's must exist. Then $t(jq) = pk$ for some k and we have

$$\psi_{jq}(0) = 1 \neq 0 = \varphi_{pk}(0) = \varphi_{t(jq)}(0)$$

and thus t cannot be a translation of ψ onto φ.

The same techniques can be used to show the existence of two simple regular programming systems such that neither one can be translated onto the other by any front-to-back mapping. □

While Rogers (1958) showed that all acceptable programming systems are isomorphic, the isomorphism construction given there can produce very complex isomorphisms between very simple Gödel numberings. The problem arises when the one-to-one translations to which the construction is applied have very long "cycles" in their composition which force the isomorphism to take on very large values. This phenomenon is exhibited precisely in the proof of the next proposition.

[1] P. van Emde Boas (private communication) has independently established that GNReg is not closed under regular isomorphisms, but via a weaker result than this proposition.

PROPOSITION 2.3. *There are two exponential time computable Gödel numberings and one-to-one translations of each into the other computable in exponential time such that the isomorphism constructed by* Rogers' method *cannot be bounded by any elementary function.*

Proof. In fact, a single programming system which can be chosen to be both LBA and polynomial time computable is used which is constructed by taking a prefix programming system and "spreading" it out to contain infinitely many large gaps filled with the empty function. The translations constructed are permutations which are the identity on the original programs, and which infinitely often map the new programs in the gaps in such a way that the composition of the permutations contains very large cycles. This is done in such a way that the isomorphism constructed by Rogers' method ends up mapping some relatively short programs in these cycles to very much longer programs.

Specifically, define the function f by setting $f(0)=0$ and

$$f(n+1)=\begin{cases} 2^{2^{y+1}}+1 & \text{if } f(n)+1=2^{2^y} \text{ for } y>0, \\ f(n)+1 & \text{otherwise.} \end{cases}$$

Let $\varphi \in \mathrm{GNPrefix}$ and define $\psi_{f(i)}=\varphi_i$ for all i and $\psi_j=\varnothing$ (the empty function) for all j not in the range of f; note that 2^y is not in the range of f for all $y \geqq 2$. Since f is computable in linear space and squared time, $\psi \in \mathrm{GNLBA} \cap \mathrm{GN}\mathscr{P}$. Let g be an increasing function such that if $y \geqq 2$ then $g(2^y)=2^z$ for some $z>y+1$. We now use g to construct two permutations, s and t, which are the identity on the range of f. For infinitely many "widely spaced" and "easily recognized" values of y of the form $y=2^{2^z}$ for some $z \geqq 2$ the definition of s and t on x such that $y \leqq x \leqq g^{2^{y+2}}(y)$ is given below (g^n stands for the n-fold composition of g with itself; $g^0(z)=z$); let the widely spaced and easily recognized values of y be given by some huge honest function such that if y' and y'' are successive values of y then $g^{2^{y'+2}}(y')<y''$. Let $y=2^{2^z}$ with $z \geqq 2$ be one of our chosen values and let x be such that $y \leqq x \leqq g^{2^{y+2}}(y)$; s and t on x are defined below:

$$s(x)=\begin{cases} x-1 & \text{if } y<x \leqq 2y, \\ g(x) & \text{if } x=g^{2i}(y) \text{ with } 0 \leqq i \leqq y, \\ g^{-1}(x) & \text{if } x=g^{2i+1}(y) \text{ with } 0<i \leqq y, \\ 2y & \text{if } x=g(y), \\ x & \text{otherwise;} \end{cases}$$

$$t(x)=\begin{cases} g(x) & \text{if } x=g^{2i+1}(y) \text{ with } 0 \leqq i \leqq y, \\ g^{-1}(x) & \text{if } x=g^{2i}y \text{ with } 0<i \leqq y+1, \\ x & \text{otherwise.} \end{cases}$$

Figure 1 illustrates the essential features of this definition. We see that if $x \in$ range f then $s(x)=t(x)=x$, as claimed, and therefore s and t are both one-to-one translations of ψ into itself. An examination of Rogers' (1958) isomorphism construction shows that if p is the isomorphism given by that construction for the two injections s and t and y is one of our chosen values, then

$$p(2y)=g^{2^{y+1}}(y)>g^{2^y}(2y)$$

and

$$p^{-1}(2y)=g^{2y+2}(y)>g^{2y}(2y).$$

Finally, if we take $g(x)=2^x$ then we get that s and t are both computable in exponential time but that p and p^{-1} are both not bounded by any elementary function.

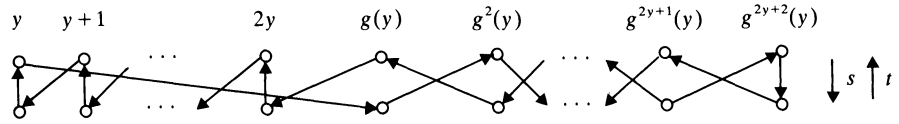


FIG. 1

The same type of construction yields similar results for other classes of Gödel numberings. For example, if we take $g(x)=x^2$ then we get that s and t are both LBA and polynomial time computable but that p is neither LBA nor polynomial time computable. □

The isomorphisms constructed by the techniques in Schnorr's paper (1975) do not suffer from the same rate-of-growth flaw as Rogers'; in fact a careful inspection of the construction yields that the isomorphisms will always be elementary in the translations from which they are constructed. However, it does not seem that these isomorphisms are necessarily computable in the time and space bounds desirable for GNLBA and GNEXP; the problem stems from the fact that the original translations might be translating very long programs in one system to very short ones in the other.[2] Thus one possible route to a solution is to insure the existence of length-increasing, one-to-one translations, and one way to do this is to construct padding functions within the appropriate class of functions. Call p a *padding function* for the programming system φ if for all programs x, $\varphi_x=\varphi_{p(x)}$ and $|x|<|p(x)|$.

PROPOSITION 2.4. (a) *Every* LBA *Gödel numbering has an* LBA*-computable padding function*; *and*

(b) *every exponential time Gödel numbering has exponential time computable padding function.*

Proof. The proof uses repeated translation into and padding within a prefix programming system, with careful counting and bounding to insure that the computation succeeds, and does so within the required resource bounds. Specifically, let $\psi\in$ GNLBA and let $\varphi\in$ GNPrefix be as in the proof of Proposition 2.1; that is $\varphi_{10^ij}(x)=\varphi_i^2(j,x)$ with φ^2 an acceptable Gödel numbering of the partial recursive functions of two arguments. For this system there is a $q=10^i$ such that

[2] There is implicit recognition of this technical problem in a recent paper by Hartmanis and Berman (1976) on a somewhat different topic.

$\varphi_x = \varphi_{qx}$ for all x and a $p = 10^j$ which is a prefix translation of ψ into φ ($\psi_x = \varphi_{px}$ for all x) such that $i < j$; note that $q^k x \neq py$ for all x, y, and k. Using the technique from Schnorr's paper given at the end of § 1, let t be a one-to-one LBA-computable translation of φ into ψ, and let n be such that $|t(x)| \leqq n|x|$ for all x.

Now suppose that we are given a program x. We wish to find a program y such that $\psi_x = \psi_y$ and $|x| < |y|$, and to have y LBA-computable from x. If we consider the programs px, qpx, $\cdots$, $q^{|x|}px$ then we have $|x|+1$ distinct programs in the system φ, each of which is equivalent to x and each of which has length less than or equal to $|x||q|+|p|+|x| \leqq c|x|$ for some constant c. Then $t(px)$, $t(qpx)$, $\cdots$, $t(q^{|x|}px)$ are $|x|+1$ distinct programs in the system ψ, each of which is equivalent to x; call them $x_0, \cdots, x_{|x|}$. If we are lucky, one of these will have length greater than $|x|$. Suppose we are not lucky; then $\{q^u px_v : 0 \leqq u, v \leqq |x|\}$ contains $(|x|+1)^2$ distinct programs in φ, all equivalent to x and all of length $\leqq c|x|$, and therefore $\{t(q^u px_v) : 0 \leqq u, v \leqq |x|\}$ contains $(|x|+1)^2$ distinct programs in ψ, all equivalent to x. If none of these has length greater than $|x|$ then we can repeat the process of translating them all into φ and padding each one up to $|x|$ many times and then translating back to ψ to get $(|x|+1)^3$ distinct programs all equivalent to x, and so on. Since $2^m = m^{m/(\log m)}$, by the time we have repeated this process $|x|/(\log|x|)$ many times we *must* get a program equivalent to x which is longer than x.

The argument we have just given shows that there exist integers $0 \leqq i_1, \cdots, i_j \leqq |x|$ with $1 \leqq j \leqq |x|/(\log|x|)$ such that if

$$y = t(q^{i_j} pt(q^{i_{j-1}} pt(\cdots t(q^{i_1} px)\cdots)))$$

then $\psi_x = \psi_y$, $|x| < |y|$ and

$$|t(q^{i_m} pt(\cdots t(q^{i_1} px)\cdots))| \leqq |x|$$

for all $1 \leqq m < j$ and

$$|q^{i_m} pt(\cdots t(q^{i_1} px)\cdots)| \leqq c|x|$$

for all $1 < m < j$. Therefore, given the integers $i_1, \cdots, i_j$ we can compute y in space less than or equal to $nc|x|$. Since $i_1, \cdots, i_j$ can be written in space $\leqq 2|x|$, an LBA operating in space $2nc|x|$ can simply search through all possible sequences $i_1, \cdots, i_j$ and give as output the y from the first such sequence which it encounters which meets the conditions stated above. This gives an LBA-computable padding function for ψ.

To prove part (b), suppose that $\psi \in$ GNEXP. Then exactly the same construction as was given above will work; t will be computable in exponential time, allowing all of the other computations to be performed in exponential time as well. □

We now use this proposition to establish closure of GNLBA and GNEXP.

THEOREM 2.5. (a) *Between every two* LBA *Gödel numberings there is an* LBA*-computable isomorphism*; *and*

(b) *between every two exponential time computable Gödel numberings there is an exponential time computable isomorphism.*

Proof. Let $\varphi \in$ GNLBA and let s be an LBA-computable s_1^1 function for φ; that is, for each e, $s(e, i)$ is LBA-computable as a function of i. Let p be an LBA-computable padding function for φ, and for all e and i, define $t(e, i) = p^j(s(e, i))$ where j is the least natural number such that $|i| < |p^j(s(e, i))|$. Then t is also an LBA-computable s_1^1 function for φ, and moreoever, t is length-increasing in the sense that for all e and i, $|i| < |t(e, i)|$. Now for any other acceptable programming system ψ, we can use the technique from Schnorr's paper given at the end of § 1 to get a one-to-one, length-increasing, LBA-computable translation of ψ into φ; note that for such a translation, an LBA can check whether a given program is in its range, and if it is the LBA can compute the program's inverse image under the translation.

Now suppose that φ and ψ are both LBA programming systems, and let s and t be one-to-one, length-increasing, LBA-computable translations of each into the other. Then we simply define the function f (as in the proof of the Cantor–Bernstein theorem) as follows: $f(x) = s(x)$ if there is an i such that $(s^{-1}t^{-1})^i(x) = y$ for some y and y is not in the range of t, and $f(x) = t^{-1}(x)$ if there is an i such that $t^{-1}(s^{-1}t^{-1})^i(x) = y$ for some y and y is not in the range of s (i.e. *otherwise*). Then it is easily seen that f is an LBA-computable isomorphism between φ and ψ.

The proof of part (b) is the same, with "LBA-computable" replaced by "exponential time computable." Indeed, the methods in the proofs of Proposition 2.4 and Theorem 2.5 show closure for any class of programming systems defined from a class of computable functions which is closed under composition and "exponential searches" of the type performed in these proofs. Thus these methods show closure for virtually all classes of programming systems defined from "natural" complexity classes which include the LBA-computable functions. □

With the previous theorem, we have now proved all but one of the conjectures on closure made by Hartmanis and Baker (1975), as well as having given a correct proof of closure for GNEXP. The remaining conjecture on closure made by Hartmanis and Baker concerns polynomial time programming systems. Just as $\mathcal{P}$, the class of problems solvable in polynomial time, is the class which theoretical computer science currently has to work with which comes closest to being the class of practically solvable problems, the class of polynomial time programming systems, GN$\mathcal{P}$, is the class which the theory of algorithms currently has to work with which is of greatest interest to practical computer science: it certainly includes all systems which are used in practice; it leaves leeway to account for program syntax (since all context-free languages can be recognized in less than cubic time); and it is invariant over a wide variety of formal models of computing.

The techniques used to prove Proposition 2.4 and Theorem 2.5 also yield results about polynomial time and "nondeterministic" polynomial time computable programming systems. For the purposes of this paper, we shall use a "conservative" definition of what it means for an arbitrary function to be computable nondeterministically in a given resource bound. A function f is *computable nondeterministically in time* t if there is a nondeterministic computation system (e.g. Turing machine) which always runs in time $t(n)$ on inputs of length n, which for every argument x has some computation which yields $f(x)$, and which for every argument x has no computation which yields a value other than $f(x)$. It is worth

noting that using techniques from Baker, Gill and Solovay (1975) there is a straightforward proof that if $\mathcal{P} = \mathcal{NP}$ then every *function* computable nondeterministically in polynomial time can be computed deterministically in polynomial time.[3] We let GN$\mathcal{NP}$ stand for the class of nondeterministic polynomial time programming systems.

THEOREM 2.6. (a) *If $\mathcal{P} = \mathcal{NP}$ then between any two polynomial time Gödel numberings there is a polynomial time computable isomorphism.*

(b) *If $\mathcal{NP}$ is closed under complemention then between any two nondeterministic polynomial time Gödel numberings there is an isomorphism computable nondeterministically in polynomial time.*

(c) *In any case, between any two nondeterministic polynomial time Gödel numberings there is an isomorphism computable (deterministically) in polynomial time from a set in $\mathcal{NP}$ (i.e. from an oracle for a set in $\mathcal{NP}$).*

Proof. Parts (a) and (b) follow directly from part (c). We shall employ the methods of Proposition 2.4 and Theorem 2.5, but we cannot be quite so direct in our approach. Let $\psi \in \text{GN}\mathcal{NP}$ and let s' be a nondeterministic polynomial time s_1^1 function for ψ. Let φ be the prefix system from the proof of Proposition 2.1; let p be a prefix translation of ψ into φ, and let q be a padding prefix for φ, both as in the proof of Proposition 2.4. Also let t be a one-to-one nondeterministic polynomial time computable translation of φ into ψ. For each x and y we define $S(x, y)$ to be a set of *possible* values of a length-increasing s_1^1 function for ψ; specifically, we define $S(x, y) = \{s'(x, y)\}$ if $|y| < |s'(x, y)|$, and

$$\begin{aligned} S(x, y) = \{z = t(q^{i_j}pt(\cdots t(q^{i_1}ps'(x, y))\cdots)): \\ 1 \leqq j \leqq |y|/\log|y|,\ 0 \leqq i_1, \cdots, i_j \leqq |y|, \\ |y| < |z|, \text{ and for all } 1 \leqq m < j, \\ |t(q^{i_m}pt(\cdots t(q^{i_1}ps'(x, y))\cdots))| \leqq |y|\} \end{aligned}$$

if $|s'(x, y)| \leqq |y|$. By the proof of Proposition 2.4, $S(x, y)$ is always nonempty. For each x, the set of possible pairs, $\text{PP}S_x$, defined by

$$\text{PP}S_x = \{(y, z): z \in S(x, y)\}$$

is a set in $\mathcal{NP}$. We define $s(x, y)$ to be the least element of $S(x, y)$ for all x and y; then s is a length-increasing s_1^1 function for ψ.

We now make a slight extension of the technique at the end of § 1. Let f be any total recursive function. With a standard application of the recursion theorem we can effectively find an e such that for all x and z

$$\psi_{s(e,x)}(z) = \begin{cases} 0 & \text{if } S(e, y) \cap S(e, x) \neq \varnothing \text{ for some } y < x, \\ 1 & \text{if } S(e, y) \cap S(e, x) = \varnothing \text{ for all } y < x \text{ and} \\ & \quad S(e, y) \cap S(e, x) \neq \varnothing \text{ for some } x < y \leqq z, \\ \psi_{f(x)}(z) & \text{otherwise.} \end{cases}$$

[3] Valiant (1974) has independently and previously obtained similar results. Others have also made essentially the same observation.

It is easily seen that $S(e, x) \cap S(e, y) = \varnothing$ for all $x \neq y$, and thus that $s(e, x)$ is one-to-one as a function of x and $\psi_{s(e,x)} = \psi_{f(x)}$ for all x. Therefore for any acceptable programming system there is an e with $s_e(x) = s(e, x)$ such that s_e is a one-to-one, length-increasing translation of that system into ψ, such that $\mathrm{PP}S_e \in \mathcal{NP}$, and such that $S(e, x)$ and $S(e, y)$ are disjoint for all $x \neq y$.

Now let φ and ψ be any two nondeterministic polynomial time programming systems. Let s and t be one-to-one, length-increasing translations of each into the other as above, and let S and T give the pairwise disjoint sequences of sets as above such that $\mathrm{PP}S = \{(x, y): y \in S(x)\}$ and $\mathrm{PP}T = \{(y, x): x \in T(y)\}$ are both in $\mathcal{NP}$. Then we define an isomorphism f between φ and ψ from s and t in the same way as in the proof of Theorem 2.5. We claim that f is computable (deterministically) in polynomial time from a set of $\mathcal{NP}$; note that a set computable in polynomial time from finitely many sets in $\mathcal{NP}$ is also computable in polynomial time from one set in $\mathcal{NP}$ using standard "joining" techniques. Define the set of possible elements of the range of s to be $\mathrm{PR}S = \{y: (x, y) \in \mathrm{PP}S \text{ for some } x\}$; define the set of nonminimal elements in the possible range of s to be $\mathrm{NM}S = \{z: y, z \in S(x) \text{ for some } x \text{ and some } y < z\}$; and define the set of "bounding" pairs for s to be $\mathrm{BP}S = \{(z, y): y \in T(x) \text{ for some } x < z\}$. Define $\mathrm{PR}T$, $\mathrm{NM}T$, and $\mathrm{BP}T$ similarly. Then all six of these sets are also in $\mathcal{NP}$.

We now show how to compute f in polynomial time from the eight sets $\mathrm{PP}S$, $\mathrm{PP}T$, $\mathrm{PR}S$, $\mathrm{PR}T$, $\mathrm{NM}S$, $\mathrm{NM}T$, $\mathrm{BP}S$, $\mathrm{BP}T$ in $\mathcal{NP}$. For any y, if $y \in \mathrm{PR}T$ and $y \notin \mathrm{NM}T$ then y is in the range of t. If y is in the range of t, then an oracle for $\mathrm{BP}T$ and binary search will locate $t^{-1}(y)$ in approximately $|y|$ many steps. Similarly, using $\mathrm{PR}S$, $\mathrm{NM}S$, and $\mathrm{BP}S$ we can determine whether any given x is in the range of s, and if so we can find $s^{-1}(x)$ in about $|x|$ steps. Thus for computing $f(x)$ we can determine which case in the definition of f to apply in about $|x|^2$ steps, and if the proper case is $f(x) = t^{-1}(x)$ we will have the value of $f(x)$ already. In the case that $f(x) = s(x)$, we compute the value as follows: using the polynomial bound on the original s_1^1 function, s', we compute a bound on $|s(x)|$. Then a binary search using oracles for $\mathrm{PP}S$ and $\mathrm{NM}S$ will locate $s(x)$ in a number of steps about equal to the bound on $|s(x)|$. □

It is worth noting about this proof that if we define a system in $\mathrm{GN}\mathcal{NP}$ to have *degree n* if every acceptable programming system can be translated into it by a function computable nondeterministically in time bounded by a polynomial of degree n, then the isomorphism between two systems in $\mathrm{GN}\mathcal{NP}$ of degree n will be computable in time bounded by a polynomial of degree at most $\max(n, 2)$ from a set in $\mathcal{NP}$. From this it follows that if $\mathcal{P} = \mathcal{NP}$ then there is a small constant c (which is certainly at most 10) such that between every two polynomial time programming systems of degree n there is an isomorphism computable in time bounded by a polynomial of degree at most $n + c$.

The proof of the previous theorem is one of many illustrations of the awkwardness caused when one is forced to talk only in terms of nondeterministic computations of sets and time-limited computations relative to sets (characteristic functions). Moreover, some computational problems, such as the one considered in the previous theorem, only make sense in terms of arbitrary functions. These difficulties are eliminated if the notions of $\mathcal{P}$, $\mathcal{NP}$, polynomial degrees, etc. are extended to apply to arbitrary partial functions, and this can be done in a very

straightforward manner (see Capka and Machtey). Using such extensions, we would conjecture that there are (nondeterministic) polynomial time Gödel numberings such that any isomorphism between them must be $\mathcal{NP}$-hard (i.e. every function computable nondeterministically in polynomial time is computable from it in polynomial time). Though this conjecture seems very likely, we have not yet succeeded in proving it. Note that if $\mathcal{P} \neq \mathcal{NP}$ then the strong version of our conjecture, that there are polynomial time programming systems such that any isomorphism between them must be $\mathcal{NP}$-hard contradicts the conjecture of Hartmanis and Baker (1975) that the polynomial time systems are closed. Finally, we close this section by raising the question of whether Theorem 2.6 provides the tightest possible bounds for isomorphisms between systems in GN$\mathcal{P}$ or in GN$\mathcal{NP}$.

3. Programming properties and acceptable programming systems. Implicit in Rogers (1958) is the well-known result that a universal programming system is acceptable if and only if it has an effective s_1^1 function. Hartmanis and Baker (1975) expand on this result when they observe that a programming system is a member of one of the "natural" classes which they consider if and only if it has an s_1^1 function in the corresponding class of computable functions; they also show that in any case an acceptable programming system is no more complex than its s_1^1 function, and that it has an s_1^1 function no more complex than the composition of a prefix (s_1^1) function with the translation of a prefix programming system into it. These results raise the questions of which commonly used programming properties of acceptable programming systems imply that any universal programming system with that property is in fact an acceptable programming system, and what the relationship is between the complexity of such properties and the complexity of the acceptable programming systems. In this section we give several results which answer questions of this type.

Hartmanis and Baker (1975) constructed arbitrarily complex optimal Gödel numberings using a diagonalization technique. In the proof of the next proposition we give an alternative "complexity-theoretic" construction which yields a slightly stronger result, but also gives a rather different intuitive approach to this type of question.

PROPOSITION 3.1. *For any acceptable Gödel numbering φ and total recursive function f, there is an optimal Gödel numbering ψ such that*

(a) (Hartmanis and Baker) *φ cannot be translated into ψ by a function in C_f, and*

(b) *ψ has no padding function in C_f, where the complexity class C_f is taken with respect to any given Blum complexity measure.*

Proof. Let $r(i)$ be the remainder when i is divided by 2, let k be any total recursive function mapping the natural numbers into $\{0, 1\}$ such that $k(2j) = k(2j+1)$ for all j, let $\lfloor x \rfloor$ stand for the largest integer less than or equal to x, and let θ be any optimal Gödel numbering. If we define the programming system ψ^k by

$$\psi_i^k = \begin{cases} \theta_{\lfloor i/2 \rfloor} & \text{if } k(i) = r(i), \\ \phi & \text{if } k(i) \neq r(i) \end{cases}$$

then ψ^k is an optimal Gödel numbering, and our intuition tells us that ψ^k can be made arbitrarily complex by taking k to be sufficiently complex. Basically, this is

because anything which enables us to effectively find an infinite recursive set of programs in ψ^k, none of which compute $\varnothing$, thereby enables us to "fairly easily" compute k on infinitely many arguments. We shall make this intuition more precise below. Because of the somewhat repetitive nature of this proposition, and because of the very standard nature of the complexity-theoretic arguments we shall use, the proof will be left a bit sketchy. The reader is referred to Hartmanis and Hopcroft (1971) and other literature in "abstract complexity theory" for details.

To prove part (a) of the theorem we show that there is a recursive operator $\mathscr{R}$ not depending on k such that if φ can be translated into ψ^k by a translator t in C_g then k can be computed within complexity $\mathscr{R}[g]$ on an infinite set T of arguments. Then for all k that are of complexity higher than $\mathscr{R}[f]$ almost everywhere φ cannot be translated into ψ^k by any translator in C_f.

Let A be an infinite recursive set of programs in φ such that $\varphi_i \neq \varnothing$ for all i in A and $\varphi_i \neq \varphi_j$ for all i and j in A. Then $t[A] = \{t(a): a \in A\}$ is an infinite set of programs in ψ^k and $T = \{t(a): a \in A \text{ and } t(a) = \max\{t(a'): a' \in A \text{ and } a' \leqq a\}\}$ is an infinite recursive subset of $t[A]$. Then there is a recursive operator $\mathscr{O}$ not depending on k such that T has a characteristic function in $C_{\mathscr{O}[g]}$. Furthermore $k(x) = r(x)$ for all x in T and hence k can be computed within complexity $\mathscr{P} \circ \mathscr{O}[g]$ on arguments from T, where $\mathscr{P}$ is another recursive operator not depending on k. Taking $\mathscr{R} = \mathscr{P} \circ \mathscr{O}$ proves part (a) of the theorem.

Part (b) is shown in a very similar way. There is a recursive operator $\mathscr{S}$ not depending on k such that if ψ^k has a padding function p in C_d then k can be computed within complexity $\mathscr{S}[d]$ on an infinite set P of arguments. Then any k with complexity higher than $\mathscr{S}[f]$ almost everywhere satisfies part (b).

To see that such an operator $\mathscr{S}$ exists we may assume without loss of generality that $\psi_1^k \neq \varnothing$. Then $k(x) = r(x)$ for all x in $P = \{p^i(1): i \geqq 0\}$. Clearly P is infinite and there is a recursive operator $\mathscr{T}$ not depending on k such that P has a characteristic function in $C_{\mathscr{T}[d]}$. Taking $\mathscr{S} = \mathscr{P} \circ \mathscr{T}$ with the same $\mathscr{P}$ as above proves part (b) of the theorem. □

Part (a) of the previous proposition asserts that any translation of φ into ψ must have complexity greater than f on infinitely many arguments; it is natural to ask whether this can be strengthened to require translations with complexity greater than f on all but finitely many arguments. The answer is "no". Let $\varphi \in \text{GNPrefix}$ and let ψ be any programming system. Let p be a prefix such that $\varphi_{px} = \varnothing$ for all x, and let y be such that $\psi_y = \varnothing$. Then φ can be translated into ψ by functions which map the set $p \cdot \{0, 1\}^*$ to y, and such translations will require constant time and zero space by an (off-line) Turing machine. If we wish to make the translations one-to-one, a slightly more complicated construction will do so, making the translations computable infinitely often in time $n \log n$ and space $\log n$ by a Turing machine. If φ is any acceptable programming system with s_1^1 function s, then by the proof of Proposition 2.4 φ has a padding function which is elementary in s. It follows that there is an elementary recursive operator $\mathscr{R}$ such that $\mathscr{R}[s]$ is the characteristic function of an infinite set of equivalent programs in φ, and therefore φ can be translated into any acceptable Gödel numbering by functions which are constant on this set of programs.

As is well known, standard elementary recursion theory uses an s_1^1 function in a simple construction of a total recursive function c for the effective composition

of programs (i.e. $\varphi_i \circ \varphi_j = \varphi_{c(i,j)}$); inspection of this construction shows that c is the composition of two instances of the s_1^1 function, and hence the complexity of c need be no more than that of such a function. Thus for any "natural" class of programming systems, GNC, c will be in C. The next theorem shows that a proof of the converse is fairly easy: for any function f, let us define an *iterate* of f on argument n to be the n-fold composition of f with itself evaluated on some fixed argument (i.e. $f^n(k)$ for some constant k).

THEOREM 3.2. *Any universal Gödel numbering with an effective function for composition must be an acceptable Gödel numbering, and the complexity of translations into the system need be no more than the complexity of an iterate of* (*an instance of*) *the composition function.*

Proof. Let φ be a universal programming system, c a total recursive function for composition in φ, and h an effective pairing function (mapping pairs of natural numbers one-to-one and onto the natural numbers). Let e and f be programs such that $\varphi_e(x) = h(0, x)$ and $\varphi_f(h(x, y)) = h(x+1, y)$ for all x and y. Define $d(x) = c(f, x)$ and $t(x) = d^x(e)$ for all x. Then $\varphi_{t(x)}(y) = h(x, y)$ for all x and y, as may be easily verified by induction on x. Finally, define $s(i, x) = c(i, t(x))$ and $\varphi_i^2(x, y) = \varphi_i(h(x, y))$ for all i, x, and y. Then φ^2 is a universal programming system and for all i, x, and y

$$\varphi_{s(i,x)}(y) = \varphi_i \circ \varphi_{t(x)}(y) = \varphi_i(h(x, y)) = \varphi_i^2(x, y).$$

Therefore any acceptable programming system can be translated into φ by an instance of s, which is essentially an iterate of an instance of c. (Actually, it is quite easy to verify directly from the definitions that φ^2 is in fact an acceptable Gödel numbering, and hence s is an s_1^1 function for φ.) □

The reason for labeling the previous theorem a "theorem" rather than a "proposition" was not so much its technical intricacy as what we believe are its implications for defining acceptable programming systems. From a programming point of view, a function c to effectively compose programs is certainly more natural and fundamental than an s_1^1 functions s for the "insertion of constants". Thus it would not be counterintuitive if s turned out to be significantly more complex than c, within the limits set by Theorem 3.2. The next proposition verifies this by showing there can be no significant improvement in the complexity bound given by Theorem 3.2. For these reasons we believe that the natural definition of an acceptable programming system should be a universal programming system with an effective function for the composition of programs.

PROPOSITION 3.3. *There is an acceptable Gödel numbering for which some translations into it must require exponential time to compute* (*infinitely often*), *but which has a regular composition function.*

Proof. Let $\varphi \in$ GNPostfix and define $\psi_{2^i} = \varphi_i$ for all i and $\psi_j = \varnothing$ for all j not of the form 2^i for any i. Because $\varphi \in$ GNPostfix, φ has a postfix s_1^1 function and therefore there is a p such that $\varphi_{ip}(x) = i$ for all x and i. Then if t is any translation of φ into ψ we must have $t(ip) \geqq 2^i$ for infinitely many i. For each i, let p_i be a postfix for composition with φ_i in φ; that is, $\varphi_i \circ \varphi_j = \varphi_{jp_i}$ for all j. Then for all j

$$\psi_{2^i} \circ \psi_{2^j} = \psi_{2^{jp_i}}.$$

But 2^{ip_i} *is* the binary string $1(O^{2^{|p_i|}})^i O^{p_i}$. It is thus easily verified that there is a total recursive composition function c for ψ such that for all x, $c(x, y)$ is a regular function of y. □

As we have mentioned, Hartmanis and Baker (1975) showed that any acceptable programming system has a fixed point function (as in the recursion theorem) which is not much more complex than an s_1^1 function for the system, and that there are optimal Gödel numberings with only complex fixed point functions. Rogers (1958) gave an example of a universal programming system which is not acceptable. By inspection, one sees that the system has an effective padding function but no fixed point function (in fact it does not even satisfy the weakest form of the recursion theorem). Friedberg (1958) constructed a universal programming system in which every partial recursive function has exactly one program; such a system satisfies neither the recursion theorem nor the padding lemma. These results raise the question of what "dependence" relationships exist among the properties of having effective s_1^1, fixed point, and padding functions in universal programming systems. Our next two theorems complete the answer to this question. We summarize the answer now as

COROLLARY 3.4. (a) *Universal Gödel numberings with effective s_1^1 functions are acceptable Gödel numberings and hence have effective fixed point and padding functions.*

(b) *There are universal Gödel numberings without effective s_1^1 functions which have effective fixed point functions or effective padding functions in any of the four possible combinations.*

Proof. Part (a) is implicit in Rogers (1958), and is well known. Two of the possible combinations in part (b) are in Rogers (1958) and Friedberg (1958), as we mentioned above. We prove the existence of the other two possible combinations in Theorems 3.6 and 3.7. □

If the previous corollary (specifically Theorem 3.6) were not true and every universal programming system with an effective fixed point function were acceptable, *then* we might expect there to be some bound on how much more complex an acceptable programming system could be than its fixed point function. But in view of the previous corollary, it is reasonable to guess that an acceptable programming system can be arbitrarily more complex than its fixed point function. The final three theorems of this paper verify this guess by giving the complexity theoretic version of Corollary 3.4, which we state now as

COROLLARY 3.5. (a) *There is a recursive operator $\mathcal{R}$ such that for any total recursive function f, if an acceptable Gödel numbering has an s_1^1 function in the complexity class C_f then it has fixed point and padding functions in $C_{\mathcal{R}[f]}$.*

(b) *For any total recursive function f, there are acceptable Gödel numberings with regular fixed point and padding functions, or regular fixed point functions, or regular padding functions, but with no s_1^1 functions, or no s_1^1 and no padding functions, or no s_1^1 and no fixed point functions, in C_f, respectively.*

Proof. Part (a) follows from the well known uniform constructions of fixed point and padding functions from s_1^1 functions. Part (b) will follow from our Theorems 3.8 through 3.10.

The remainder of this paper is devoted to the proofs of the theorems needed to establish Corollaries 3.4 and 3.5. In the proofs of these theorems it will be a convenient notation to write $\varphi(x, y)$ for $\varphi_x(y)$ on many occasions.

THEOREM 3.6. *There is a universal Gödel numbering which has prefix fixed point and padding functions, but which is not an acceptable Gödel numbering.*

Proof. Let $\varphi \in \mathrm{GNPrefix}$ and let ψ be the Gödel numbering constructed by Friedberg (1958) in which every partial recursive function has exactly one program. Define the system θ by

$$\theta_i = \psi_{\varphi(i,0)}$$

for all i; then θ is a universal programming system since both φ and ψ are. Notice that if $\varphi_i = \varphi_j$ then $\theta_i = \theta_j$, and so a padding prefix for φ is also a padding prefix for θ. Let n be a fixed point prefix for φ; that is, if φ_i is total then

$$\varphi_{\varphi(i,ni)} = \varphi_{ni}.$$

Let p be a prefix translation of θ into φ. Then if θ_i is total, so is φ_{pi}, and since

$$\varphi_{\varphi(pi,npi)} = \varphi_{npi}$$

we have

$$\theta_{\theta(i,npi)} = \theta_{\varphi(pi,npi)} = \theta_{npi}.$$

Therefore, np is a fixed point prefix for θ. Notice that for all partial recursive functions $f \neq \varnothing$, $\{i\colon \theta_i = f\} = \{i\colon \varphi_i(0) = c\}$ for some constant c, by the choice of ψ; hence $\{i\colon \theta_i = f\}$ is recursively enumerable. From this it follows that θ is not an acceptable programming system. □

THEOREM 3.7. *There is a universal Gödel numbering which has a prefix fixed point function, but which does not have an effective padding function (hence it is not acceptable).*

Proof. For notational convenience we shall work (mostly) over $\{a, b\}^*$, and we let ε denote the empty string. Let S be a simple nonhypersimple set with $D_\varepsilon, D_a, D_b, \cdots$ a canonical enumeration of pairwise disjoint finite sets, each of which intersects the complement of S and such that the union of all the D_x's is $\{a, b\}^*$. Let S_x be the set consisting of the first x elements in some recursive enumeration of S, and let ψ be any acceptable programming system.

We now define the system φ: for all x and y and $i \geqq 0$ let

$$\varphi(a^i, y) = \text{undefined};$$

$$\varphi(a^{i+1}bx, y) = \begin{cases} \varphi(\varphi(a^ibx, a^{i+1}bx), y) & \text{if } x \notin S_y \\ \text{undefined} & \text{if } x \in S_y; \end{cases}$$

$$\varphi(bx, y) = \begin{cases} \psi(z, y) & \text{if } x \in D_z \text{ and } x \notin S_y, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Since this construction is effective, φ has an effective universal function. For every z, since D_z intersects the complement of S, there is some x such that $\psi_z = \varphi_{bx}$. Thus φ is a universal programming system. If φ_x is total then x must be of the form a^ibz for some $i \geqq 0$ and $z \notin S$, hence $\varphi(ax, y) = \varphi(\varphi(x, ax), y)$ for all y. Thus a is a fixed point prefix for φ.

It remains to show that φ has no effective padding function. Let j be such that $\varphi_j = \varnothing$, and let k be such that $\varphi_k(x) = j$ for all x; notice that k must be of the form

$a^i bx$ for some $i \geqq 0$ and x. Then for all y

$$\varphi(ak, y) = \begin{cases} \varphi(\varphi(k, ak), y) = \varphi(j, y) = \text{undefined or} \\ \text{undefined,} \end{cases}$$

and hence by induction on i, $\varphi(a^i k, y)$ is undefined for all y and all $i \geqq 1$. It follows that if $a^m by \neq k = a^i bx$ is another program equivalent to k, then $x \neq y$. Suppose, for the sake of a contradiction, that T is an infinite, recursively enumerable set of programs equivalent to k. Then every member of T is of the form $a^i bx$ with a different x, and $\{x: a^i bx \in T\}$ is an infinite, recursively enumerable set which must therefore intersect S. But if $x \in S$, then for all i, $\varphi(a^i bx, y)$ is undefined for all but finitely many values of y, and thus $a^i bx$ cannot be equivalent to k. Therefore there can be no infinite, recursively enumerable set of programs equivalent to k, and so φ has no effective padding function. □

THEOREM 3.8. *For every total recursive function f there is an acceptable Gödel numbering with a prefix padding function but which has no s_1^1 or fixed point functions in C_f.*

Proof. For notational convenience we shall work (mostly) over $\{a, b\}^*$. Suppose that we define $\varphi(ax, y) = \varphi(x, y)$ for all x and y in $\{a, b\}^*$. Then the system φ will have a as a padding prefix, and we are left with the task of defining the rest of φ to meet the following two requirements: φ must be an acceptable programming system, and φ must have no s_1^1 or fixed point functions in C_f. We insure the first by constructing φ so that there is a total recursive translation of some given acceptable Gödel numbering ψ into φ. Let $\mathscr{R}$ be a recursive operator such that if an acceptable programming system has an s_1^1 function in C_f then it has a fixed point function in $C_{\mathscr{R}[f]}$, and let $g = \max(f, \mathscr{R}[f])$. Then if φ has no fixed point function in C_g it can have no s_1^1 or fixed point functions in C_f. Without loss of generality, we can assume that the class C_g is recursively enumerable, with $C_g = \{t_0, t_1, \cdots\}$.

We define $\varphi(\varepsilon, y) = \varepsilon$ and $\varphi(b, y) = b$ for all y, and the definition of φ_{bx} for all $x \neq \varepsilon$ is described as a construction which proceeds in stages. In part (A) of the stages we insure that there is a total recursive translation of the given acceptable programming system ψ into φ. In part (B) of the nth stage we insure that t_n is not a fixed point function of φ. Note that when part (B) of the nth stage of the construction below is entered, φ_z has been defined for at most $3n+3$ values z which do not begin with an a.

Stage n. (A) Let y be the least string bx such that φ_{bx} has not been defined, and define $\varphi_y = \psi_n$; go to part (B).

(B) Compute t_n on $3n+4$ arguments of the form bx such that φ_{bx} has not yet been defined; at least one of the following two cases holds—use one of them:

(a) $t_n(i) = a^k t_n(j)$ for some i, j, and k with $i \neq j$; in this case define $\varphi_i(y) = b$ and $\varphi_j(y) = \varepsilon$ for all y and go to stage $n+1$.

(b) $t_n(i) = a^k bx$ for some i, k, and x such that φ_{bx} has not yet been defined; in this case define $\varphi_i(y) = b$ and $\varphi_{bx}(y) = \varepsilon$ for all y and go to stage $n+1$.

Part (A) of this construction yields a total recursive translation of ψ into φ, making φ an acceptable programming system. For some n, suppose that case (a) of part (B) is used in stage n of the construction. Then $\varphi(t_n(i), y) = \varphi(t_n(j), y)$ for all

y, but

$$\varphi(\varphi(i, t_n(i)), y) = \varphi(b, y) = b$$

and

$$\varphi(\varphi(j, t_n(j)), y) = \varphi(\varepsilon, y) = \varepsilon$$

for all y. Thus in this case t_n cannot be a fixed point function for φ. Suppose that case (b) of part (B) is used. Then

$$\varphi(t_n(i), y) = \varphi(bx, y) = \varepsilon$$

and

$$\varphi(\varphi(i, t_n(i)), y) = \varphi(b, y) = b$$

for all y, and so t_n cannot be a fixed point function for φ in this case either. Therefore, φ has no fixed point function in C_g and the proof of the theorem is complete. □

THEOREM 3.9. *For every total recursive function f there is an acceptable Gödel numbering with a prefix fixed point function but which has no s_1^1 or padding functions in C_f.*

Proof. The proof is very similar to that of the previous theorem, and we shall only indicate the differences. Suppose that we define $\varphi(ax, y) = \varphi(\varphi(x, ax), y)$ for all x and y in $\{a, b\}^*$. Then the system φ will have a as a fixed point prefix, and we are left with the task of insuring that φ is acceptable and has no s_1^1 or padding functions in C_f. Let $\mathcal{R}$ be a recursive operator such that if φ has an s_1^1 function in C_f then it has a padding function in $C_{\mathcal{R}[f]}$, and let $g = \max(f, \mathcal{R}[f])$. We wish to insure that φ has no padding function in C_g, and we assume that C_g is recursively enumerable with $C_g = \{t_0, t_1, \cdots\}$.

We define $\varphi(\varepsilon, y) = \varepsilon$ and $\varphi(b, y) = a$ for all y. Note that by the definition of φ given so far, $\varphi(a, y) = \varepsilon$ for all y. The definition of φ_{bx} for all $x \neq \varepsilon$ is described as a construction in stages as in the previous proof; part (A) of the construction is the same. Part (B) of the nth stage insures that t_n is not a padding function for φ; note that when part (B) of the nth stage below is entered, φ_z has been defined for at most $2n+3$ values z which do not begin with an a.

Stage n. (A) Let y be the least string bx such that φ_{bx} has not yet been defined, and define $\varphi_y = \psi_n$; go to part (B).

(B) Compute $t_n(b), t_n^2(b), \cdots, t_n^{2n+5}(b)$; at least one of the following three cases holds:

(a) $t_n^i(b) = t_n^j(b)$ for some i and j;

(b) $t_n^i(b) = a^k t_n^j(b)$ for some i, j, and k with $i \neq j$ and $k > 0$;

(c) $t_n^i(b) = a^k bx$ for some i, k, and x such that φ_{bx} has not yet been defined; in this case define $\varphi_{bx}(y) = \varepsilon$ for all y;

in any case, go to stage $n+1$.

Part (A) makes φ an acceptable programming system as before. If case (a) of part (B) occurs at stage n of the construction, then t_n is obviously not a padding function for φ. Suppose that case (b) occurs at stage n. If $\varphi(t_n^j(b), y) \neq \varphi(b, y)$ for

some y then t_n is not a padding function. If $\varphi(t_n^l(b), y) = \varphi(b, y) = a$ for all y then

$$\begin{aligned}\varphi(t_n^i(b), y) = \varphi(a^k t_n^i(b), y) &= \varphi(\varphi(a^{k-1} t_n^i(b), a^k t_n^i(b)), y) \\ &= \varphi(\varphi(\cdots \varphi(\varphi(t_n^i(b), at_n^i(b)), aat_n^i(b)) \cdots), y) \\ &= \varphi(\varphi(\cdots \varphi(a, aat_n^i(b)) \cdots), y) = \varepsilon\end{aligned}$$

for all y, and so t_n is not a padding function for φ in this case either. Finally, suppose that case (c) occurs at stage n. Then

$$\begin{aligned}\varphi(t_n^i(b), y) = \varphi(a^k bx, y) &= \varphi(\varphi(a^{k-1} bx, a^k bx), y) \\ &= \varphi(\varphi(\cdots \varphi(\varphi(bx, abx), aabx) \cdots), y) \\ &= \varphi(\varphi(\cdots \varphi(\varepsilon, aabx) \cdots), y) = \varepsilon \neq \varphi(b, y)\end{aligned}$$

for all y. Therefore, φ has no padding function in C_g. □

THEOREM 3.10. *For every total recursive function f there is an acceptable Gödel numbering with a prefix padding function and a regular fixed point function but which has no s_1^1 function in C_f.*

Proof. The proof is similar to the previous two, but a bit more complicated. We work over $\{a, b, c\}^*$ and we let $d(x)$ denote the string which results from x by deleting all c's. Suppose that we define $\varphi(x, y) = \varphi(d(x), y)$ and $\varphi(ad(x), y) = \varphi(\varphi(d(x), ad(x)), y)$ for all x and y in $\{a, b, c\}^*$. Then the system φ will have c as a padding prefix and $n(x) = ad(x)$ will give a regular fixed point function. We are left with the task of defining the rest of φ to meet the following two requirements: φ must be an acceptable programming system, and φ must have no s_1^1 function in C_f. As before, we insure the first by constructing φ so that there is a total recursive translation of some given acceptable programming system ψ into φ. Without loss of generality, we can assume that the class C_f is recursively enumerable, with $C_f = \{t_0, t_1, \cdots\}$. If φ has an s_1^1 function in C_f then there is a t_n in C_f such that

$$\varphi(t_n(x), y) = \begin{cases} ax & \text{if } y = \varepsilon, \\ \varepsilon & \text{if } y \neq \varepsilon; \end{cases}$$

note that such a t_n must have the property that if $i \neq j$ then $d(t_n(i)) \neq d(t_n(j))$. Then φ will meet our second requirement if there is no t_n in C_f with these properties.

We define $\varphi_\varepsilon(y) = \varepsilon$ for all y, and the definition of φ_{bx} for all x in $\{a, b\}^*$ is described as a construction which proceeds in stages. In part (A) of the stages we insure that there is a total recursive translation of the given acceptable programming system ψ into φ. In part (B) of the nth stage we insure that t does not have the properties given above; note that when this part is entered, φ_{bx} has been defined for at most $2n+1$ values of x in $\{a, b\}^*$.

Stage n. (A) Let y be the least string bx such that φ_{bx} has not yet been defined, and define $\varphi_y = \psi_n$; go to part (B).

(B) Compute t_n on its first $2n+2$ arguments; at least one of the following

four cases holds:

(a) $d(t_n(i)) = d(t_n(j))$ for some $i \neq j$;
(b) $d(t_n(i)) = a^k$ for some i and k;
(c) $d(t_n(i)) = a^k bx$ and $d(t_n(j)) = a^m bx$ for some i, j, x, and $k < m$;
(d) $d(t_n(i)) = a^k bx$ for some i, k, and x such that φ_{bx} has not yet been defined; in this case define $\varphi_{bx}(y) = \varepsilon$ for all y;

in any case, go to stage $n+1$.

Part (A) makes φ an acceptable Gödel numbering, and we are left with verifying that no t_n has the properties given above. If case (a) occurs in part (B) of stage n then $d \circ t_n$ is not one-to-one. We claim that if cases (b), (c), or (d) occur, then either $\varphi(t_n(i), \varepsilon) = \varepsilon$ or $\varphi(t_n(i), az) \neq \varepsilon$ for some i and z. In case (b), $d(t_n(i)) = a^k$ and hence

$$\begin{aligned}\varphi(t_n(i), \varepsilon) = \varphi(a^k, \varepsilon) &= \varphi(\varphi(a^{k-1}, a^k), \varepsilon) \\ &= \varphi(\varphi(\cdots \varphi(\varphi(\varepsilon, a), aa) \cdots), \varepsilon) \\ &= \varphi(\varphi(\cdots \varphi(\varepsilon, aa) \cdots), \varepsilon) = \varepsilon.\end{aligned}$$

In case (c), $\varphi(t_n(i), a^{k+1}bx) = \varphi(a^k bx, a^{k+1}bx)$; if $\varphi(a^k bx, a^{k+1}bx) = \varepsilon$ then

$$\begin{aligned}\varphi(t_n(j), \varepsilon) = \varphi(a^m bx, \varepsilon) &= \varphi(\varphi(a^{m-1}bx, a^m bx), \varepsilon) \\ &= \varphi(\varphi(\cdots \varphi(\varphi(a^k bx, a^{k+1}bx), a^{k+2}bx) \cdots), \varepsilon) \\ &= \varphi(\varphi(\cdots \varphi(\varepsilon, a^{k+2}bx) \cdots), \varepsilon) = \varepsilon.\end{aligned}$$

In case (d),

$$\begin{aligned}\varphi(t_n(i), \varepsilon) = \varphi(a^k bx, \varepsilon) &= \varphi(\varphi(\cdots \varphi(\varphi(bx, abx), aabx) \cdots), \varepsilon) \\ &= \varphi(\varphi(\cdots \varphi(\varphi, aabx) \cdots), \varepsilon) = \varepsilon.\end{aligned}$$

Therefore no t_n has the properties given above, and so φ has no s_1^1 function in C_f. □

We conclude by noting that the same basic construction as in the previous proof produces an acceptable Gödel numbering with a prefix padding function and a regular fixed point function which has no s_1^1 function which is bounded by f almost everywhere.

REFERENCES

T. P. BAKER, J. GILL AND R. SOLOVAY (1975), *Relativizations of the P = NP? question*, this Journal, 4, pp. 431–442.

D. CAPKA AND M. MACHTEY, *Nondeterministic computation of partial functions and an extended P-hierarchy*, in preparation.

R. M. FRIEDBERG (1958), *Three theorems on recursive enumeration*, J. Symbolic Logic, 23, pp. 309–316.

J. HARTMANIS AND T. P. BAKER (1975), *On simple Gödel numberings and translations*, this Journal, 4, pp. 1–11.

J. HARTMANIS AND L. BERMAN (1976), *On isomorphisms and density of NP and other complete sets*, Proc. 8th ACM Symp. Theory Comp., pp. 30–40.

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

M. MACHTEY AND P. YOUNG (1976), *Simple Gödel numberings, translations, and the P-hierarchy: Preliminary report*, Proc. 8th ACM Symp. Theory Comp., pp. 236–243.

H. ROGERS, JR. (1958), *Gödel numberings of partial recursive functions*, J. Symbolic Logic, 23, pp. 331–341.

——— (1967), *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York.

C. P. SCHNORR (1975), *Optimal enumerations and optimal Gödel numberings*, Math. Systems Theory, 8, pp. 192–191.

L. VALIANT (1974), *Relative complexity of checking and evaluating*, manuscript, Univ. of Leeds, England.

SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

POLYNOMIALS WITH 0-1 COEFFICIENTS THAT ARE HARD TO EVALUATE*

RICHARD J. LIPTON†

Abstract. We show the existence of polynomials with 0-1 coefficients that are hard to evaluate even when arbitrary preconditioning is allowed. Further we show that there are power series with 0-1 coefficients such that their initial segments are hard to evaluate.

Key words. polynomial evaluation, 0-1 polynomials, preconditioning

1. Introduction. The well known results of Belaga, Motzkin, and Winograd [1], [4], [7] demonstrate that a polynomial of degree n requires $n/2$ multiplications (divisions) and n additions (subtractions) when the coefficients of the polynomial are algebraically independent. The model of computation they employ allows the use of arbitrary additional complex numbers at no cost. The selection of these numbers—called "preconditioning"—can depend in any way whatever on the original polynomial. In contrast to these results, as pointed out by Paterson and Stockmeyer [5] and Strassen [6], most polynomials that one wishes to evaluate have rational coefficients, not algebraically independent ones. The known results on rational polynomial evaluation are as follows:

1. Paterson and Stockmeyer have shown that there are rational polynomials that require $\sim\sqrt{n}$ nonscalar multiplications (divisions) when complex preconditioning is allowed. When only integer preconditioning is allowed and no division they show that there are 0-1 polynomials (polynomials with 0, 1 as coefficients) that require $\sim\sqrt{n}$ nonscalar multiplications. (We use $\sim f(n)$ to mean a function $g(n)$ such that $c_1 f(n) \leqq g(n) \leqq c_2 f(n)$ for some $c_1, c_2 > 0$.)

2. Strassen has shown that specific rational polynomials require $\sim n$ total operations when complex preconditioning is allowed.

3. Lipton and Dobkin [3] have shown that there are 0-1 polynomials that require $\sim n/\log n$ operations when finite preconditioning is allowed (that is, all scalars used must lie in some fixed finite set).

In summary, Strassen has shown that a specific polynomial is hard to evaluate when complex preconditioning is allowed. For weaker models (integer or finite preconditioning) Paterson and Stockmeyer, and Lipton and Dobkin have shown the existence of hard 0-1 polynomials.

As Strassen states [6], an interesting open question is the construction of hard 0-1 polynomials when complex preconditioning is allowed. In this direction it is interesting to note that the coefficients of Strassen's grow at double exponential rate, so they grow very fast indeed. Essentially Strassen's results are based on the fact that while his coefficients are algebraically dependent they do not satisfy any

* Received by the editors October 3, 1975, and in revised form November 5, 1976.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520. This represents work performed while visiting IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. Also supported in part by the National Science Foundation under Contract DCR 74-12870.

relation with "small" degree or height. Clearly this method cannot be directly applied to any 0-1 polynomial: The coefficients of a 0-1 polynomial satisfy a great number of very simple relations.

The main results of this paper are a step in the direction of answering Strassen's open question. We restrict ourselves to proving the existence of hard 0-1 polynomials. Our main results are Theorems 5 and 7. Theorem 5 shows that there are 0-1 polynomials that require $\sim n^{1/4}/\log n$ nonscalar multiplications/divisions when arbitrary complex preconditioning is allowed. Theorem 7 addresses a related question:

> Can one find a 0-1 power series $\sum_{i=0}^{\infty} a_i x^i$ such that each initial segment $\sum_{i=0}^{n} a_i x^i$ is a hard 0-1 polynomial?

The motivation for this question is twofold. First, many interesting polynomials that arise naturally are the initial segments of some power series. Second, could it be that there are hard 0-1 polynomials of every degree while all power series are easy? Theorems 7, 8, 9 show that this cannot be so. More exactly they show that:

1. There is a 0-1 power series whose initial segments require $\sim n^{1/4}/\log n$ nonscalar multiplications/divisions when arbitrary complex preconditioning is allowed.
2. There is a 0-1 power series whose initial segments require $\sim n^{1/2}$ nonscalar multiplications when integer preconditioning is allowed.
3. There is a 0-1 power series whose initial segments require $\sim n/\log n$ total operations when finite preconditioning is allowed.

It is interesting to note that these results on power series suggest a number of questions about the interplay between polynomial evaluation and language theory. These are discussed further in § 3.

2. Hard 0-1 polynomials. Our model of computation is the standard one based on "straightline programs." Suppose that $p(x)$ is a polynomial with complex coefficients. Then $S_1, \cdots, S_m$ is a computation of $p(x)$ over A where $A \subseteq \mathbb{C}$ ($\mathbb{C}$ = the set of complex numbers) provided for each step S_i either

1. $S_i \in A \cup \{x\}$ or
2. $S_i = S_j \circ S_k$ where $j, k < i$ and $\circ \in \{+, -, \times, \div\}$.

And

3. $S_m = p(x)$.

The set A determines what type of preconditioning is allowed. The measure of complexity used is either the total number of operations or the operations of some specific type. A step $S_i = S_j \circ S_k$ is a *nonscalar* $*$ *operation* provided $\circ$ is $\times$ and both S_j and S_k are not in A, or $\circ$ is $\div$ and S_k is not in A.

The proof of the existence of hard 0-1 polynomials is essentially two steps.

1. First, we show that "small" polynomials $\propto$ 0-1 polynomials. That is, the evaluation of polynomials with small coefficients (in a sense to be made precise) can be reduced to the evaluation of not too many (in a sense to be made precise) 0-1 polynomials. Thus $\propto$ acts here as an analogy to reducibility in the sense of automata theory.
2. Second, we show that there are hard small polynomials.

Of course, Theorem 5 is then a consequence of 1) and 2).

The details of these two steps are now presented.

DEFINITION. Say $(a_1, \cdots, a_n)$ is a *generalized* 0-1 *vector* provided for some x all a_i lie in $\{0, x\}$.

DEFINITION. Suppose that $a = (a_1, \cdots, a_n)$ is a vector of natural numbers. Then define $d(a)$ to be

$$\min \{k | \exists V^1, \cdots, V^k \text{ generalized 0-1 vectors with } V^1 + \cdots + V^k = a\}.^1$$

LEMMA 1. *The function $d(a)$ satisfies the following*:

1. $d(a_1, \cdots, a_n) = d(b_1, \cdots, b_n)$ *if* $a_1, \cdots, a_n$ *is a permutation of* $b_1, \cdots, b_n$.
2. $d(a_1, \cdots, a_n) \leqq d(b_1, \cdots, b_m)$ *if* $\{a_1, \cdots, a_n\} \subseteq \{b_1, \cdots, b_m\}$.
3. $d(a_1 + b_1, \cdots, a_n + b_n) \leqq d(a_1, \cdots, a_n) + d(b_1, \cdots, b_n)$.
4. $d(a_1, \cdots, a_n) \leqq d(1, 2, 3, \cdots, t)$ *where* $t = \max(a_1, \cdots, a_n)$.
5. $d(a_1, \cdots, a_n) \leqq \log (n+2)^2$ *provided* $a_1, \cdots, a_n$ *is an arithmetic progression.*
6. $d(a_1, \cdots, a_n) \leqq \log (t+2)$ *where* $t = \max(a_1, \cdots, a_n)$.

Proof. 1. If π is a permutation of $\{1, \cdots, n\}$, then define $\pi(a_1, \cdots, a_n) = (a_{\pi(1)}, \cdots, a_{\pi(n)})$. Now let us assume that $\pi(a_1, \cdots, a_n) = (b_1, \cdots, b_n)$ and also that $V^1 + \cdots + V^k = (a_1, \cdots, a_n)$ for some generalized 0-1 vectors. Then since π is additive,

$$\pi(V^1) + \cdots + \pi(V^k) = \pi(a_1, \cdots, a_n) = (b_1, \cdots, b_n).$$

Therefore, it follows that $d(a_1, \cdots, a_n) \geqq d(b_1, \cdots, b_n)$; the same argument with $(a_1, \cdots, a_n)$ and $(b_1, \cdots, b_n)$ interchanged shows that $d(a_1, \cdots, a_n) \leqq d(b_1, \cdots, b_n)$. Thus, $d(a_1, \cdots, a_n) = d(b_1, \cdots, b_n)$.

2. We need only prove that

$$d(a_1, \cdots, a_n) \leqq d(a_1, \cdots, a_n, b)$$

in order to prove 2): it follows by 1) and induction on $n - m$. Therefore, suppose that $V^1 + \cdots + V^k = d(a_1, \cdots, a_n, b)$ for some generalized 0-1 vectors. Then clearly

$$W^1 + \cdots + W^k = (a_1, \cdots, a_n)$$

where W^i is the projection of V^i into the first n coordinates. Hence, $d(a_1, \cdots, a_n) \leqq d(a_1, \cdots, a_n, b)$.

3. Suppose that $(a_1, \cdots, a_n) = V^1 + \cdots + V^k$ and $(b_1, \cdots, b_n) = W^1 + \cdots + W^m$. Then

$$(a_1 + b_1, \cdots, a_n + b_n) = V^1 + \cdots + V^k + W^1 + \cdots + W^m;$$

hence, $d(a_1, +b_1, \cdots, a_n + b_n) \leqq d(a_1, \cdots, a_n) + d(b_1, \cdots, b_n)$.

4. Clearly $\{a_1, \cdots, a_n\} \subseteq \{1, \cdots, t\}$ where $t = \max(a_1, \cdots, a_n)$; hence, $d(a_1, \cdots, a_n) \leqq d(1, \cdots, t)$ by 2).

5. Define $f(n)$ to be the maximum value of $d(a_1, \cdots, a_n)$ provided $a_1, \cdots, a_n$ is an arithmetic progression. We first assume that n is a power of 2. Let

[1] We use componentwise addition.

[2] All logarithms are to base 2.

$a_i = bi + c$ for $i = 1, \cdots, n$. Then by 1),

$$d(a_1, \cdots, a_n) \leqq d(a_1, a_3, \cdots, a_{2m-1}, a_2, a_4, \cdots, a_{2m})$$

where $n = 2m$. Moreover, by 3),

$$d(a_1, \cdots, a_n) \leqq d(a_1, a_3, \cdots, a_{2m-1}, a_1, a_3, \cdots, a_{2m-1}) + d(\underbrace{0, 0, \cdots, 0}_{m \text{ copies}}, \underbrace{c, c, \cdots, c}_{m \text{ copies}})$$

since $a_1 + c = a_2, \cdots, a_{2m-1} + c = a_{2m}$. By 2),

$$d(a_1, a_3, \cdots, a_{2m-1}, a_1, a_3, \cdots, a_{2m-1}) = d(a_1, a_3, \cdots, a_{2m-1}).$$

Therefore, $d(a_1, \cdots, a_n) \leqq d(a_1, a_3, \cdots, a_{2m-1}) + 1$. Clearly $a_1, a_3, \cdots, a_{2m-1}$ is an arithmetic progression of length $n/2$; hence,

$$f(n) \leqq f\left(\frac{n}{2}\right) + 1.$$

Since $f(1) = 1$, it follows that $f(n) \leqq \log(n+1)$ provided n is a power of 2. Next suppose that n is not a power of 2. Now $f(k)$ is clearly a nondecreasing function of k: this follows by 2) and the fact that any arithmetic progression of length k can be extended to one of length $k+1$. Thus $f(n) \leqq f(n')$ where n' is the least power of $2 \geqq n$; hence, $f(n) \leqq f(2n) \leqq \log(n+2)$.

6. Clearly 6) is an immediate consequence of 4) and 5). □

Let $C_{01}(n)$ be the number of nonscalar $*$ operations required to evaluate any 0-1 polynomial of degree $\leqq n$ over C.

LEMMA 2. *For any natural numbers $a_0, \cdots, a_n$ it follows that $C_{01}(n) \cdot d(a_0, \cdots, a_n)$ is an upper bound on the number of nonscalar $*$ operations needed to evaluate the polynomial*

$$p(n) = \sum_{i=0}^{n} a_i x^i$$

over C.

Proof. Let $d(a_0, \cdots, a_n) = m$. Then there are generalized 0-1 vectors $V^1, \cdots, V^m$ such that $V^1 + \cdots + V^m = (a_0, \cdots, a_n)$. Let

$$P_i(x) = \sum_{j=0}^{n} V_j^i x^j$$

where V_j^i = the jth component of the vector V^i. Then

$$\sum_{i=1}^{m} P_i(x) = \sum_{i=1}^{m} \sum_{j=0}^{n} V_j^i x^j$$

$$= \sum_{j=0}^{n} x^j \sum_{i=1}^{m} V_j^i$$

$$= \sum_{j=0}^{n} x^j a_j.$$

Therefore, $p(x)=\sum_{i=1}^{m} P_i(x)$. The number of nonscalar $*$ operations sufficient to evaluate $p(x)$ is thus bounded by the total number of nonscalar $*$ operations required to evaluate all the $P_i(x)$ polynomials. Each $P_i(x)$ is equal to $bq(x)$ for some scalar b and some 0-1 polynomial $q(x)$ of degree $\leqq n$; hence, each $P_i(x)$ can be evaluated in $C_{01}(n)$ nonscalar $*$ operations. Finally, the upper bound on the number of nonscalar $*$ operations for $p(x)$ is

$$C_{01}(n)\cdot d(a_0,\cdots,a_n). \quad \square$$

LEMMA 3. *There is a nontrivial polynomial* $H(a_1,\cdots,a_n)$ *of degree* $\leqq n^{18}$ *(for* $n>n_0$ *where* n_0 *is some constant) such that if* $H(a_1,\cdots,a_n)\neq 0$, *then*

$$\sum_{i=1}^{n} a_i x^i$$

requires at least $n^{1/4}$ *nonscalar* $*$ *operations when arbitrary preconditioning is allowed.*

Proof. Following Paterson and Stockmeyer [5], we first observe that if $p(x)$ can be computed with $\leqq n^{1/4}$ nonscalar $*$ operations, then $p(x)$ can be computed by the scheme $\mathcal{P}$ for some m_{ij}, m'_{ij} in $\mathbb{C}$:

$$\mathcal{P}: \quad P_{-1}=1.$$

$$P_0=x.$$

For $r-1,\cdots,\lfloor 2_n^{1/4}\rfloor$,

$$P_r=\Big(\sum_{i=-1}^{r-1} m_{r,i}P_i\Big)\circ_r\Big(\sum_{i=-1}^{r-1} m'_{r,i}P_i\Big)$$

where $\circ_r$ is $\times$ if r is odd and $\div$ if r is even. Finally,

$$p(x)=\sum_{r=1}^{\lfloor 2_n^{1/4}\rfloor} m_{0,i}P_i.$$

The total number of operations—scalar and nonscalar—is

$$3+\sum_{r=1}^{\lfloor 2_n^{1/4}\rfloor}(4r+5)\leqq 4n^{1/2}$$

for $n>n_0$ where n_0 is some constant.

We now proceed to apply the method of Strassen's Theorem 2.5 to the scheme $\mathcal{P}$ with the parameters m_{ij}, m'_{ij} considered as indeterminates. Viewed this way, $\mathcal{P}$ computes (by Strassen's Lemma 2.4)

$$q(\bar{m})+\sum_{i=1}^{\infty} q_i(\bar{m})x^i$$

for some polynomials[3] q_i where $\bar{m}$ is the vector of all the parameters m_{ij}, m'_{ij}. In

[3] $q(\bar{m})$ need not be polynomials, but are rational.

the notation of Strassen

$$g = n^{18}, \qquad m \leqq 4n^{1/2},$$
$$q = n, \qquad s \leqq 4n^{1/2},$$
$$d = n.$$

For $n > n_1$ where n_1 is some constant,

$$g^{q-m-2} = n^{18(n-4\sqrt{n}-2)}$$
$$> n^{4\sqrt{n}(4\sqrt{n}+1)} n^n$$
$$= d^{s(m+1)} q^q.$$

Now let $H(a_1, \cdots, a_n)$ be the nontrivial polynomial of degree $\leqq g$ that exists by Strassen's theorem.[4] Now suppose that $a_1, \cdots, a_n$ are natural numbers such that $H(a_1, \cdots, a_n) \neq 0$ and yet

$$p(x) = \sum_{i=1}^{n} a_i x^i$$

can be done in $< n^{1/4}$ nonscalar $*$ operations; we will reach a contradiction. For some complex parameters $\bar{t}$ the scheme $\mathscr{P}$ computes $p(x)$; hence,

$$p(x) = \sum_{i=1}^{n} q_i(\bar{t}) x^i + q(\bar{t}).$$

Thus $H(a_1, \cdots, a_n) = H(q_1(\bar{t}), \cdots, q_n(\bar{t}))$. But

$$H(q_1(\bar{t}), \cdots, q_n(\bar{t})) = 0$$

by the method of Strassen's theorem; hence, $H(a_1, \cdots, a_n) = 0$. This is a contradiction. □

LEMMA 4. *Suppose that $q(x_1, \cdots, x_k)$ is a polynomial of degree $\leqq g$ such that $q(x_1, \cdots, x_k) = 0$ for all natural numbers $0 \leqq x_i \leqq g$. Then $q(x_1, \cdots, x_k)$ is identically* 0.

Proof. We will use induction on k. When $k = 1$ the result is an immediate consequence of the fact that a polynomial of degree $\leqq g$ can have at most g zeros without being identically 0. Now suppose that $k > 1$. Then

$$q(x_1, \cdots, x_k) = \sum_{i=0}^{g} P_i(x_2, \cdots, x_k) x_1^i$$

for some polynomials $P_0, \cdots, P_g$ of degree $\leqq g$. Assume that $q(x_1, \cdots, x_k)$ is not identically 0. Then there is a $P_d(x_2, \cdots, x_k)$ that is not identically 0. Then

$$\exists 0 \leqq x_2 \leqq g \cdots \exists 0 \leqq x_k \leqq g \quad \text{with } P_d(x_2, \cdots, x_k) \neq 0;$$

for otherwise by induction $P_d(x_2, \cdots, x_k)$ is identically 0. Let $x_2', \cdots, x_k'$ be such natural numbers. Then $q(x, x_2', \cdots, x_k')$ is a polynomial in x of degree $\leqq g$ with

[4] The field k of Strassen's theorem is the complex numbers extended by the indeterminates m_{ij} and m_{ij}'.

$g+1$ zeros; hence $q(x, x'_2, \cdots, x'_k)$ is identically 0. This contradicts the fact that $P_d(x'_2, \cdots, x'_k) \neq 0$. □

We are finally ready to prove our main result.

THEOREM 5. *There are* 0-1 *polynomials that require* $\sim n^{1/4}/\log n$ *nonscalar* $*$ *operations when arbitrary preconditioning is allowed.*

Proof. Let $H(a_1, \cdots, a_n)$ be as in Lemma 3. Let $g = n^{18}$, the degree of H. Then by Lemma 4 there is a $(a_1, \cdots, a_n)$ such that $H(a_1, \cdots, a_n) \neq 0$ and $0 \leqq a_i \leqq g$ for $i = 1, \cdots, n$. By Lemma 3, with $a_0 = 0$

$$\sum_{i=0}^{n} a_i x^i$$

requires $n^{1/4}$ nonscalar $*$ operations. By Lemma 2,

$$C_{01}(n) \cdot d(a_0, \cdots, a_n) \geqq n^{1/4}.$$

Therefore, by Lemma 1 part 6),

$$C_{01}(n) \geqq n^{1/4}(18 \log n + 2). \quad \square$$

3. Hard 0-1 power series. In this section we study the complexity of the initial segments to power series whose coefficients are 0-1.

DEFINITION. Let $D_A(\alpha)$ be the number of nonscalar $*$ operations required to evaluate

$$p(x) = \alpha_0 + \cdots + \alpha_k x^k$$

over A where α is a 0-1 string of length $k+1$. Also we will say that $p(x)$ is the polynomial that *corresponds* to α.

LEMMA 6. *For any* 0-1 *strings* α *and* β,[5]

$$D_A(\alpha\beta) + D_A(\alpha) + 2 \log |\alpha| + 1 \geqq D_A(\beta).$$

Proof. Let $q(x) = \sum_{i=0}^{|\alpha|-1} \alpha_i x^i$ and $r(x) = \sum_{i=0}^{|\beta|-1} \beta_i x^i$. Then $p(x) = q(x) + x^{|\alpha|} r(x)$ is the polynomial that corresponds to $\alpha\beta$. Now in order to evaluate $r(x)$, the polynomial that corresponds to β, proceed as follows:

1. Compute $p(x)$.
2. Compute $q(x)$.
3. Form $g(x) = p(x) - q(x)$.
4. Compute $x^{|\alpha|}$.
5. Form $h(x) = g(x)/x^{|\alpha|}$.

The above computation clearly takes at most

$$D_A(\alpha\beta) + D_A(\alpha) + 2 \log |\alpha| + 2$$

nonscalar $*$ operations. Now $g(x) = x^{|\alpha|} r(x)$, and so $h(x) = r(x)$. □

Suppose that $\sum_{i=0}^{\infty} a_i x^i$ is a power series. Then the polynomials

$$\sum_{i=0}^{n} a_i x^i$$

are the initial segments of the given power series. The next theorem shows that

[5] $|\alpha|$ = length of α and $\alpha\beta$ is the concatenation of α and β.

there are power series with 0-1 coefficients such that their initial segments are hard infinitely often.

THEOREM 7. *There is a* 0-1 *power series whose initial segments of length n infinitely often require* $\sim n^{1/4}/\log n$ *nonscalar* $*$ *operations when arbitrary complex preconditioning is allowed.*

Proof. Let $p_k(X)$ be a 0-1 polynomial of degree 2^k that requires

$$\frac{\varepsilon 2^{k/4}}{k}$$

nonscalar $*$ operations where $\varepsilon > 0$; it exists of course by Theorem 5. Then let α^k be the 0-1 string of the coefficients of $p_k(x)$. Also let

$$\alpha = \alpha^1 \alpha^2 \cdots$$

be the infinite 0-1 string formed by concatenating together all the α^i's. By Lemma 6 for all k,

$$D_A(\alpha^1 \cdots \alpha^k) + D_A(\alpha^1 \cdots \alpha^{k-1}) + 2 \log |\alpha^1 \cdots \alpha^{k-1}| + 2 \geqq D_A(\alpha^k).$$

By the definition of α^k, $D_A(\alpha^k) \geqq \varepsilon 2^{k/4}/k$. Thus,

$$D_A(\alpha^1 \cdots \alpha^{k-1}) + D_A(\alpha^1 \cdots \alpha^k) \geqq \frac{\varepsilon' 2^{k/4}}{k}$$

for large k and some $\varepsilon' > 0$ since $|\alpha^1 \cdots \alpha^{k-1}| \leqq 2^{k+1}$. Thus either $D_A(\alpha^1 \cdots \alpha^k)$ or $D_A(\alpha^1 \cdots \alpha^{k-1})$ exceeds

$$m = \frac{\varepsilon' 2^{k/4}}{2k}.$$

In either case we have shown that there is an initial segment of length between 2^{k-1} and 2^{k+2} which requires $\geqq m$ nonscalar $*$ operations. Since k was arbitrary the theorem is proved. □

The same type of reasoning also yields:

THEOREM 8. *There is a* 0-1 *power series whose initial segments of length n infinitely often require* $\sim n^{1/2}$ *nonscalar* $*$ *operations when only integer preconditioning is allowed.*

THEOREM 9. *There is a* 0-1 *power series whose initial segments of length n infinitely often require* $\sim n/\log n$ *total operations when finite preconditioning is allowed.*

It is interesting to note in passing a connection between these results and language theory. Let α be the 0-1 infinite sequence that is constructed in Theorem 7. Also let

$$L = \{\alpha_0, \cdots, \alpha_i | i \geqq 0\}.^6$$

By construction L is recursive. (Actually one must choose αi to be the lexicographically smallest 0-1 string of length 2^k such that the corresponding polynomial requires the specified number of operations.) The exact complexity of L is of

[6] α_i is the ith symbol in α.

some interest since it hopefully would help one understand what makes a 0-1 polynomial hard. In this direction we have

THEOREM 10. *L cannot be context-free* (Book and Lipton [2]).

Proof. Suppose that L was context free. Then a "pumping lemma" argument shows that for some strings a and b with b nonempty,

$$ab^i \in L \quad \text{for all } i \geqq 0.$$

Now select any $x \in L$ with $|x| \geqq |a|$. Then x must be the prefix of ab^i for some i: this follows since there is exactly one string of each length in L.

Thus α is an ultimately periodic string. We will now show that the polynomial that corresponds to any initial segment is easy to evaluate. In order to prove this it is sufficient to show that for any string β the polynomial that corresponds to β^n (β concatenated n times) is easy to evaluate. Now this polynomial is equal to

$$\sum_{m=0}^{n} \sum_{i=0}^{k-1} \beta_i x^{km+i}$$

which is in turn equal to

$$\left(\sum_{m=0}^{n} x^{km}\right)\left(\sum_{i=0}^{k-1} {}_i x^i\right).$$

This polynomial can be evaluated in $\log n + O(k)$ total steps; hence, L cannot be context free. □

Acknowledgments. The author wishes to thank Larry Stockmeyer for a number of helpful suggestions, and the referees for their helpful comments.

REFERENCES

[1] E. G. BELAGA, *Some problems involved in the computation of polynomials*, Dokl. Akad. Nauk SSSR, 123 (1958), pp. 775–777.

[2] R. V. BOOK AND R. J. LIPTON, Unpublished manuscript.

[3] R. J. LIPTON AND D. P. DOBKIN, *Complexity measures and hierarchies for the evaluation of integers, polynomials, and N-linear forms*, Seventh Annual ACM Symposium on Theory of Computing Association for Computing Machinery, New York, 1975.

[4] T. S. MOTZKIN, *Evaluation of polynomials and evaluation of rational functions*, Bull. Amer. Math. Soc., 61 (1955), p. 163.

[5] M. PATERSON AND L. STOCKMEYER, *On the number of nonscalar multiplications necessary to evaluate polynomials*, this Journal, 2 (1973), pp. 60–66.

[6] V. STRASSEN, *Polynomials with rational coefficients which are hard to compute*, this Journal, 3 (1974), pp. 128–148.

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

SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

SOUNDNESS AND COMPLETENESS OF AN AXIOM SYSTEM FOR PROGRAM VERIFICATION*

STEPHEN A. COOK†

Abstract. A simple ALGOL-like language is defined which includes conditional, while, and procedure call statements as well as blocks. A formal interpretive semantics and a Hoare style axiom system are given for the language. The axiom system is proved to be sound, and in a certain sense complete, relative to the interpretive semantics. The main new results are the completeness theorem, and a careful treatment of the procedure call rules for procedures with global variables in their declarations.

Key words. program verification, semantics, axiomatic semantics, interpretive semantics, consistency, completeness

1. Introduction. The axiomatic approach to program verification along the lines formulated by C. A. R. Hoare (see, for example, [6] and [7]) has received a great deal of attention in the last few years. My purpose here is to pick a simple programming language with a few basic features, give a Hoare style axiom system for the language, and then give a clean and careful justification for both the soundness and adequacy (i.e., completeness) of the axiom system. The justification is done by introducing an interpretive semantics for the language, rather like that in [10] and [8]. These two papers also have outlined soundness arguments for axiom systems, but for somewhat different language features, axioms, and interpretive models. The completeness claim and argument presented here is new (although completeness and incompleteness proofs inspired by an earlier version of this paper [2] appear in [3], [11], [12], [13], and [14]). I have tried to choose the axioms and rules of the formal system to be as simple as possible, subject to the constraints that they be sound, complete, and in the style and spirit of Hoare's rules.

Donahue [4] presented a soundness argument for a similar axiom system, but soundness was proved in terms of mathematical semantics in the style of Dana Scott. This led to a rather different argument than that presented here.

Most of the complication in the present paper comes from handling procedure statements. The rules for procedure call statements often (in fact usually) have technical bugs when stated in the literature, and the rules stated in earlier versions of the present paper are not exceptions. In the process of trying to prove the soundness of these rules, I uncovered some of the bugs, and this led me to believe a careful and detailed proof of soundness is necessary to have any confidence that there are no further bugs. I have allowed procedure declarations to have global variables (subject to some restrictions) and this has added to the complications of the rules and their justifications.

In addition to procedure statements, the programming language used allows assignment, conditional, while, compound, and block statements, but disallows input/output statements, jumps, functions, and data structures.

* Received by the editors July 1, 1976, and in revised form March 31, 1977.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A7.

The programming language is specified in § 2, the interpretive model appears in § 3, and the axiom system in § 4. The soundness of the system is proved in § 5, and the completeness is proved in § 6.

2. The language Al[$\mathscr{L}_1$, $\mathscr{L}_2$]. We do not want to specify the particular primitive relations and operations used to form expressions in our ALGOL fragment, so we assume these are the same as a given language $\mathscr{L}_1$ for the first order predicate calculus. For concreteness we could take $\mathscr{L}_1 = \mathscr{L}_N$ where nonlogical symbols in $\mathscr{L}_N$ are $\{<, =, +, \cdot, 0, 1\}$, but more generally any list $\{P_1, P_2, \cdots\}$ of predicate symbols and any list $\{f_1, f_2, \cdots\}$ of function symbols will do. In addition, we assume we are given a predicate calculus language $\mathscr{L}_2$, an extension of $\mathscr{L}_1$, which will be used for assertions. In most examples, we will take $\mathscr{L}_2 = \mathscr{L}_1 = \mathscr{L}_N$.

The programming language Al[$\mathscr{L}_1, \mathscr{L}_2$] will be a modified subset of ALGOL 60, with the following objects.

Variables. The variables (identifiers) will coincide with the variables of $\mathscr{L}_1$. All variables have the same (unspecified) type. In our example in which $\mathscr{L}_1 = \mathscr{L}_N$, we think of that type as "integer".

Declarations. a) *Procedure declarations* have the form

$$p(\bar{x} : \bar{v})\ \textbf{proc}\ K$$

where p is the procedure name, $(\bar{x} : \bar{v})$ is the formal parameter list, and K is the procedure body. $\bar{x}$ and $\bar{v}$ are disjoint lists of distinct variables, and the variables in $\bar{v}$ cannot occur to the left of any assignment statement in K, nor can they appear as actual parameters to the left of the colon (:) in any procedure call statement in K. The variables $\bar{x}$ and $\bar{v}$ are considered local to the declaration. We allow global variables in K (in addition to $\bar{x}$ and $\bar{v}$). However, the variables in $\bar{x}$ and $\bar{v}$ cannot occur globally in any procedure declaration for another procedure which could be activated by executing K. Also note the restriction on procedure calls stated below.

To avoid confusion over associating procedure names with procedure bodies, we require that no procedure name can be declared more than once in any program. In general we shall assume that some fixed procedure declaration is associated with each procedure p.

We shall assume in this paper that no procedure is recursive. That is, there is no chain of procedure names $p_1, \cdots, p_n$ such that $p_1 = p_n$, and the procedure body for p_i contains a call to p_{i+1}, $1 \leqq i < n$.

b) *Variable declarations* have the form

$$\textbf{new}\ x$$

where x is any variable. Both procedure and variable declarations occur at the beginning of blocks. A variable can occur without being declared, in which case it acts as an input to the program (it must have a value before the program is executed). Also, a given variable can be declared in any number of blocks.

Expressions. a) A *Boolean expression* is any quantifier-free formula of $\mathscr{L}_1$. For example, in the case of $\mathscr{L}_N$, $0 = 0, x + 1 < y + 1 \ \&\ z < x$ are Boolean expressions.

b) Numerical expressions are terms of $\mathscr{L}_1$. For example, in the case of $\mathscr{L}_N$, $(x + 1) \cdot (z + 1) + 1$, 0, 1 are numerical expressions.

Statements. a) *Assignment* statements have the form $x := e$, where x is a variable and z is a numerical expression.

b) *Procedure calls* have the form **call** $p(\bar{u} : \bar{e})$, where $\bar{u}$ is a list of distinct variables, $\bar{e}$ is a list of expressions containing no variable in $\bar{u}$, and no variable in $(\bar{u} : \bar{e})$ occurs as a nonlocal variable either in the procedure declaration for p or in the procedure declaration for any procedure which can be activated indirectly by activating p. (Formal parameters are local to a procedure declaration.)

c) *Conditional, while, compound,* and *block* statements are as in ALGOL 60, except we use **fi** and **od** to punctuate the end of conditional and while statements, respectively, as in ALGOL 68.

3. The interpretive model. An *interpretive model* $\mathcal{M} = \mathcal{M}[\mathcal{I}]$ for the programming language Al$[\mathcal{L}_1, \mathcal{L}_2]$ is determined by giving an interpretation $\mathcal{I}$ for the predicate calculus language $\mathcal{L}_2$ (of course $\mathcal{I}$ also interprets the language $\mathcal{L}_1$). Thus $\mathcal{I} = \langle D, \bar{P}_1, \bar{P}_2, \cdots, \bar{f}_1, \bar{f}_2, \cdots \rangle$ where D is a nonempty domain, $\{\bar{P}_1, \bar{P}_2, \cdots\}$ are the predicates on D interpreting the predicate symbols of $\mathcal{L}_2$, and $\{\bar{f}_1, \bar{f}_2, \cdots\}$ are the functions on D interpreting the function symbols of $\mathcal{L}_2$. As usual in predicate calculus interpretations, the predicates $\bar{P}_i$ and the functions $\bar{f}_i$ are assumed to be total. In our example $\mathcal{L}_N$, the natural interpretation is $\mathcal{I}_N$, in which D is the set of integers, and $<$, $=$, $+$, $\cdot$, 0, 1 are all given their standard meanings. If the function symbol $\div$ is included in the language and it is interpreted as division, then some value must be assigned to $n \div 0$. One way to do this is to add an extra element Ω, the "undefined" element to the domain D, and let $n \div 0$ be Ω. In this case, all function and predicate symbols must have their interpretations extended to be defined at Ω, although their values at Ω could be Ω. In any case, during execution of a program, all expressions which must be evaluated will have well-defined values in D, so an undefined expression is never a cause for termination.

Notation. If E is a term or formula, $t_1, \cdots, t_k$ are terms, and $y_1, \cdots, y_k$ are distinct variables, then

$$E\frac{t_1 \cdots t_k}{y_1 \cdots y_k}$$

indicates the result of simultaneously substituting $t_1, \cdots, t_k$ for free occurrences of $y_1, \cdots, y_k$, respectively, in E. In the definitions of $P(s, \delta)$ and $e(s, \delta)$ below, the role of t_i is played by $s(\delta(y_i))$. The latter object is an element of D rather than a term, so that strictly speaking a constant c_i should be introduced whose value under the interpretation is $s(\delta(y_i))$, and then

$$e(s, \delta) = \mathcal{I}\left(e\frac{c_1 \cdots c_k}{y_1 \cdots y_k}\right).$$

However, the abuse of notation below is convenient and, we hope the intended meaning is clear.

The set of *registers* $\mathcal{R}$ is the infinite set $\{X_1, X_2, \cdots\}$. A *state* of $\mathcal{M}[\mathcal{I}]$ is a total map $s: \mathcal{R} \to D$. A *variable assignment* is a one-one partial map

$$\delta: \{\text{variables of } \mathcal{L}_2\} \to \mathcal{R}$$

with a finite domain. If P is a formula of $\mathcal{L}_2$ with free variables $y_1, \cdots, y_k$, and s is

a state, δ is a variable assignment to $\{y_1, \cdots, y_k\}$, then

$$P(s, \delta) \equiv \begin{cases} \text{true} & \text{if } P\dfrac{s(\delta(y_1)) \cdots s(\delta(y_k))}{y_1 \cdots y_k} \text{ is true in } \mathscr{I}, \\ \text{false} & \text{otherwise.} \end{cases}$$

Thus P becomes either true or false in $\mathscr{I}$ when its free variables are given values according to s and δ.

If e is a term (i.e., numerical expression) with free variables $y_1, \cdots, y_k$, and δ, s are as above, then

$$e(s, \delta) = \mathscr{I}\left(e\frac{s(\delta(y_1)) \cdots s(\delta(y_k))}{y_1 \cdots y_k}\right).$$

Thus $e(\delta, s)$ is that element of D which is the value of the expression e when the free variables of e are assigned values according to s and δ.

A *procedure assignment* is a partial map

$$\pi\colon \{\text{procedure names}\} \to \{\text{procedure bodies}\} \times \{\text{formal parameter lists}\}$$

such that π has a finite domain. Thus, if the procedure declaration $p(\bar{x} : \bar{v})$ **proc** K occurs in a program, we will define $\pi(p) = \langle K, (\bar{x} : \bar{v})\rangle$.

The heart of the model $\mathscr{M}$ is the function Comp(A, s, δ, π), which assigns to a statement A, state s, variable assignment δ, and procedure assignment π, a finite or infinite sequence $\langle s_1, s_2, \cdots\rangle$ which represents the successive states of the computation determined by the statement A when the initial state is s. This computation is not defined unless δ assigns a register at least to each free (i.e., global) variable of A and π assigns a procedure body and formal parameter list at least to each procedure name associated with A which has no corresponding procedure declaration. (In general, A will be taken from the interior of a block B, and the declarations of B must be recorded in δ and π, as shown below.)

The function Comp(A, s, δ, π) is defined below by giving one defining clause for each of 8 possible forms which the statement A can take. The reader can check that every legal statement A in Al$[\mathscr{L}_1, \mathscr{L}_2]$ fits one and only one of the 8 cases. The definition is recursive, in the sense that Comp appears on the right side of the clauses. This may appear ironic in a paper on program verification, since one of the important issues in programming language semantics is interpreting recursively defined procedures. However, one does not have to understand recursive procedures in general in order to understand this specific definition. Suffice it to say that we intend Comp to be evaluated by "call by name," in the sense that occurrences of Comp are to be replaced successively by their meanings according to the appropriate clauses in the definition. Simplifications are to be made using knowledge about the model $\mathscr{M}[\mathscr{I}]$. Of course the process may not terminate, in which case an infinite sequence of states will be generated.

Notation. A^* stands for a sequence $A_1; A_2; \cdots; A_k$, $k \geqq 0$, of statements of Al$[\mathscr{L}_1, \mathscr{L}_2]$, and D^* stands for a sequence $D_1; D_2; \cdots; D_l$, $l \geqq 0$, of declarations of Al$[\mathscr{L}_1, \mathscr{L}_2]$, and A is a statement of Al$[\mathscr{L}_1, \mathscr{L}_2]$. The symbol $\hat{}$ indicates concatenation. More precisely, $C_1 \hat{} C_2$ is the concatenation of sequences C_1 and

C_2, if C_1 is finite, and $C_1 \hat{} C_2$ is C_1 if C_1 is infinite. If K is a procedure body, then

$$K\frac{\bar{u}, \bar{e}}{\bar{x}, \bar{v}}$$

indicates the result of substituting the actual parameters $\bar{u}, \bar{e}$ for the corresponding free (i.e., global) occurrences of the formal parameters $\bar{x}$ and $\bar{v}$ (respectively) in K. If any variable z in $(\bar{u}, \bar{e})$ is declared locally in K and if the formal parameter corresponding to z occurs within the scope of the declaration of z in K, then the local variable z must be renamed in K before the substitution takes place, so that no actual parameter gets "caught" by the declaration when it is substituted.

Out(A, s, δ, π) is the last state in the sequence given by Comp(A, s, δ, π), when this is a finite sequence, and is undefined otherwise.

DEFINITION. Comp$(A, s, \delta, \pi) =$

Cases A: (The value of Comp appears to the right of the arrow for each of the eight cases of the form of A given below.)

begin new x; D^*; A^* **end** $\rightarrow \langle s \rangle \hat{}$ Comp (**begin** D^*; A^* **end**, s, δ', π),

$$\text{where } \delta'(y) = \begin{cases} \delta(y), & \text{if } y \neq x, \\ X_{k+1}, & \text{if } y = x, \text{ where } X_k \text{ is the highest indexed register in the range of } \delta, \end{cases}$$

begin $p(\bar{x} : \bar{v})$ **proc** K; D^*; A^* **end** $\rightarrow \langle s \rangle \hat{}$ Comp(**begin** D^*; A^* **end**, s, δ, π'),

$$\text{where } \pi'(q) = \begin{cases} \pi(q), & \text{if } q \neq p, \\ \langle K, (\bar{x} : \bar{v}) \rangle, & \text{if } q = p. \end{cases}$$

begin A_1; A^* **end** $\rightarrow$ Comp$(A_1, s, \delta, \pi) \hat{}$ Comp(**begin** A^* **end**, Out$(A_1, s, \delta, \pi), \delta, \pi)$.

begin end $\rightarrow \langle s \rangle$.

$$x := e \rightarrow \langle s' \rangle, \text{ where } s'(X_i) = \begin{cases} s(X_i), & \text{if } \delta(x) \neq X_i, \\ e(s, \delta), & \text{if } \delta(x) = X_i. \end{cases}$$

call $p(\bar{u} : \bar{e}) \rightarrow \langle s \rangle \hat{}$ Comp$\left(K\frac{\bar{u}, \bar{e}}{\bar{x}, \bar{v}}, s, \delta, \pi\right)$, where $\pi(p) = \langle K, (\bar{x} : \bar{v}) \rangle$.

if R **then** A_1 **else** A_2 **fi** $\rightarrow \begin{cases} \langle s \rangle \hat{}\, \text{Comp}(A_1, s, \delta, \pi), & \text{if } R(s, \delta) \text{ is true}, \\ \langle s \rangle \hat{}\, \text{Comp}(A_2, s, \delta, \pi), & \text{otherwise} \end{cases}$

while R **do** A_1 **od** $\rightarrow \begin{cases} \text{Comp}(A_1, s, \delta, \pi) \hat{}\, \text{Comp}(\textbf{while } R \textbf{ do } A_1, \text{Out}(A_1, s, \delta, \pi), \delta, \pi), & \text{if } R(s, \delta) \text{ is true}. \\ \langle s \rangle, \quad \text{otherwise}. \end{cases}$

where δ is defined for all variables global in A and π is defined for all undeclared procedure names in A.

Note that the clause defining the statement **call** $p(\bar{u} : \bar{e})$ means procedures have dynamic rather than static scope.

4. The axioms and rules. The axioms and rules of inference of our deductive system are basically those of Hoare [6], [7], with amendments due to Lauer [10], Igarashi, London and Luckham [9] (among others) and modified so as to reflect the structure of the recursive definition of the function Comp.

The basic object in the system is the formula $P\{A\}Q$, where P and Q are formulas in $\mathscr{L}_2$ and A is a statement of Al$[\mathscr{L}_1, \mathscr{L}_2]$. Informally, $P\{A\}Q$ is true (relative to our interpretation $\mathscr{I}$) iff whenever the assertion P is true before A is executed, either A will fail to terminate, or Q will hold after A is executed.

DEFINITION. The *free set* of a statement A consists of all variables with global occurrences in A, together with variables with global occurrences in the procedure bodies of any procedures which might be activated by executing A. A formal definition can be given recursively by considering each of the possible statement types for A (as in the definition of Comp). We give four of the more interesting clauses in this definition. The free set of **begin new** x; D^*; A^* **end** consists of the union of the free sets of D^* and A^*, with x deleted. The free set of **begin** $p(\bar{x} : \bar{v})$ **proc** K; D^*; A^* **end** consists of the union of the free sets of K, D^*, and A^*, except $\bar{x}$ and $\bar{v}$ are excluded from the free set of K. The free set of $x := e$ consists of the variable x, together with all variables in e. The free set of **call** $p(\bar{u} : \bar{e})$ consists of the variables in $(\bar{u} : \bar{e})$, together with the free set of

$$K\frac{\bar{u}, \bar{e}}{\bar{x}, \bar{v}},$$

where K is the procedure body for p and $(\bar{x} : \bar{v})$ are the formal parameters for p. (Note that K and $(\bar{x} : \bar{v})$ are uniquely determined in any program by our convention of unique declarations, and see the remarks at the end of this section.)

Notation. P, Q, R, S stand for the formulas of $\mathscr{L}_2$.

$$\frac{\alpha_1, \cdots, \alpha_n}{\beta}$$

indicates the rule: from the formula(s) α_i, $i = 1, \cdots, n$, deduce the formula β.

$$\frac{D, \alpha}{\beta},$$

where D is a declaration of some procedure p, indicates the rule

$$\frac{\alpha}{\beta},$$

with the understanding that all calls of p in α and β are according to D (see the remarks at the end of this section).

The rules and axiom schemes of the system $\mathscr{H}$ consist of 1) through 11) below.

1) *Rule of variable declarations.*

$$\frac{P\frac{y}{x}\{\textbf{begin } D^*; A^* \textbf{ end}\}Q\frac{y}{x}}{P\{\textbf{begin new } x; D^*; A^* \textbf{ end}\}Q}$$

where y is not free in P or Q, and is not in the free set of D^* or A^*.

2) *Rule of procedure declarations.*

$$\frac{D, P\{\textbf{begin } D^*; A^* \textbf{ end}\}Q}{P\{\textbf{begin } D; D^*; A^* \textbf{ end}\}Q}$$

where D is any procedure declaration.

3) *Rule of compound statements.*

$$\frac{P\{A\}Q,\ Q\{\textbf{begin } A^* \textbf{ end}\}R}{P\{\textbf{begin } A; A^* \textbf{ end}\}R}$$

4) *Axiom of compound statements.*

$$P\{\textbf{begin end}\}P$$

5) *Axiom of assignment statements.*

$$P\frac{e}{x}\{x := e\}P$$

6) *Rule of conditional statements.*

$$\frac{P \,\&\, R\{A_1\}Q,\ P \,\&\, \neg R\{A_2\}Q}{P\{\textbf{if } R \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}\}Q}$$

7) *Rule of while statements.*

$$\frac{P \,\&\, Q\{A\}P}{P\{\textbf{while } Q \textbf{ do } A \textbf{ od}\}P \,\&\, \neg Q}$$

8) *Rule of procedure calls.*

$$\frac{p(\bar{x} : \bar{v}) \textbf{ proc } K,\ P\{K\}Q}{P\{\textbf{call } p(\bar{x} : \bar{v})\}Q}$$

9) *Rule of parameter substitution.*

$$\frac{P\{\textbf{call } p(\bar{x}' : \bar{v}')\}Q}{P\frac{\bar{u}, \bar{e}}{\bar{x}', \bar{v}'}\{\textbf{call } p(\bar{u} : \bar{e})\}Q\frac{\bar{u}, \bar{e}}{\bar{x}', \bar{v}'}}$$

provided that no variable in $\bar{u}$ (except possibly one in $\bar{x}'$) occurs free in P or Q. Here $\bar{x}'$ and $\bar{v}'$ are lists of distinct variables which may be, but need not be, the same as the formal parameters $(\bar{x} : \bar{v})$ for the procedure p. We require, of course, that the statements **call** $p(\bar{x}' : \bar{v}')$ and **call** $p(\bar{u} : \bar{e})$ be syntactically correct, which means (for **call** $p(\bar{u} : \bar{e})$) that the variables $\bar{u}$ be distinct and have no occurrence in the expressions $\bar{e}$, and no variable in $\bar{u}$ or $\bar{e}$ (other than one in $\bar{x}$ or $\bar{v}$) can be in the free set of the procedure body K of p.

10) *Rule of variable substitution.*

$$\frac{P\{\textbf{call } p(\bar{u} : \bar{e})\}Q}{P\sigma\{\textbf{call } p(\bar{u} : \bar{e})\}Q\sigma}$$

where

$$\sigma = \frac{\bar{z}'}{\bar{z}}$$

is a substitution of expressions for variables such that no variable in $\bar{z}$ or $\bar{z}'$ occurs in the free set of **call** $p(\bar{u}:\bar{e})$.

11) *Rule of consequence.*

$$\frac{P \supset R,\ R\{A\}S,\ S \supset Q}{P\{A\}Q}$$

Note that in the rule of consequence, $P \supset R$ and $S \supset Q$ are formulas of $\mathscr{L}_2$. We assume they are correct (that is their universal closures are true in the model $\mathscr{I}$) but the manner in which they are deduced is not the concern of this axiom system. This point is discussed further in later sections.

It is worth pointing out that rules 9) and 10) could be replaced by the simpler rule

$$\frac{p(\bar{x}:\bar{v})\ \textbf{proc}\ K,\ P\left\{K\frac{\bar{u},\bar{e}}{\bar{x},\bar{v}}\right\}Q}{P\{\textbf{call}\ (\bar{u}:\bar{e})\}Q}$$

and soundness and completeness could be preserved (in fact the justification would be much simpler). However, this rule is unsatisfactory because its use requires a separate proof of the hypothesis

$$P\left\{K\frac{\bar{u},\bar{e}}{\bar{x},\bar{v}}\right\}Q$$

each time a **call** statement for the procedure p appears with different actual parameters. In contrast, the present rule 8) requires the proof just once of a general property $P\{K\}Q$ of the procedure body, and rule 9) (with rules 10) and 11)) allows the deduction of suitable instances of the property for different sets of actual parameters. (The use of rule 10) will come out in the completeness argument in the last section.)

A second objection to the above alternative to rules 9) and 10) is that it spoils the pleasing principle that no substitutions for variables are made in the program text for the hypothesis of any rule.

The rules of our system are a little awkward in handling procedure declarations. This is not a real issue for our particular programming language, since we do not allow a given procedure name to have more than one declaration in a given program. If more than one such declaration were allowed (as in ALGOL 60), some device would have to be introduced in the rules to keep track of which declaration applied to a given procedure call statement. One possibility suggested in Gorelick [5], is to transform each rule

$$\frac{\alpha_1, \cdots, \alpha_n}{\beta} \quad \text{into} \quad \frac{D^*/\alpha_1, \cdots, \alpha_n}{D^*/\beta}$$

so that the context of procedure declarations D^* is made explicit at each rule

application. The rule for blocks with procedure declarations would now become

$$\frac{D^*; D/P\{\textbf{begin } D_1^*; A^* \textbf{ end}\}Q}{D^*/P\{\textbf{begin } D; D_1^*; A^* \textbf{ end}\}Q}$$

enabling us to "discharge" heretofore implicit assumptions about the context of procedure declarations. Similarly, the rule for procedures would become

$$\frac{D^*/P\{K\}Q}{D^*; p(\bar{x}:\bar{v})\ \textbf{proc}\ K/P\{\textbf{call}\ p(\bar{x}:\bar{v})\}Q},$$

where p is a new procedure name. This would be a possible way of handling the problem if one thought it were important to allow a given name to refer to two different procedures. However, in practice, this flexibility would probably cause more confusion than it would save. Furthermore, our definition of Comp would have to be significantly more complicated, requiring that the map π store the environment of the procedure body, as well as the body and formal parameters. Therefore, we shall stick to our simplifying assumption, and our simpler rules.

5. The model satisfies the axioms and rules. Most of the rules and axioms seem to be clearly valid, given the informal meaning for $P\{A\}Q$ stated above. However, there is always a worry that some condition or possibility has been overlooked. This is particularly true of rules 8)–10) for procedure calls, variations of which have been stated incorrectly in the literature several times.

How can we be sure we aren't omitting some restrictions on these rules or the use of parameters that are necessary to ensure the validity of the rules? One way is to prove that all the axioms and rules are true in our interpretive model $\mathcal{M}[\mathcal{I}]$. (At least this ensures that the axioms and rules are correct, provided it is agreed that the model is correct.)

Thus suppose we are given an interpretation $\mathcal{I}$ for $\mathcal{L}_2$, and hence a model $\mathcal{M} = \mathcal{M}[\mathcal{I}]$ for our language Al$[\mathcal{L}_1, \mathcal{L}_2]$. We say a formula P of $\mathcal{L}_2$ is *true in* $\mathcal{I}$ (or in $\mathcal{M}$) iff the closure of P is true in $\mathcal{I}$, where by closure we mean universal closure (i.e., the result of prefixing to P universal quantifiers for all free variables in P).

DEFINITION. A formula $P\{A\}Q$ is *true in* $\mathcal{M}$ (denoted by: $\models_{\mathcal{M}} P\{A\}Q$) iff for all states s, s', if $P(s, \delta)$ is true in $\mathcal{M}$ and $s' = \text{Out}(A, s, \delta, \pi)$, then $Q(s', \delta)$ is true in $\mathcal{M}$, where δ is any assignment to the free variables of P, Q, and the free set of A, and π assigns the proper bodies and parameter lists to all necessary procedure names. The subscript $\mathcal{M}$ on $\models$ is sometimes omitted.

Lemma 4, at the end of this section, states that this definition of truth is independent of δ.

DEFINITION. A formula $P\{A\}Q$ is *valid* iff it is true in all models $\mathcal{M}[\mathcal{I}]$. A rule

$$\frac{\alpha_1, \cdots, \alpha_n}{\beta}$$

is *valid* iff β is true in every model in which $\alpha_1, \cdots, \alpha_n$ are true.

Theorem 1 below states that all our axioms and rules are valid. However, they are not sufficient in the following sense: It is usually necessary to use the rule of consequence (rule 11)) to prove interesting formulas, and for this we need a

method of establishing the truth of the (universal closures of) the formulas $P \supset R$ and $S \supset Q$ in $\mathscr{L}_2$. Hence we will need to supplement our rules and axioms by a deductive system $\mathscr{D}$ for deducing formulas in $\mathscr{L}_2$ whose closures are true in $\mathscr{I}$. Of course $\mathscr{D}$ must be *sound* relative to $\mathscr{I}$, in the sense that the universal closure of every formula deducible in $\mathscr{D}$ is true in $\mathscr{I}$. We should emphasize that the soundness of $\mathscr{D}$ has meaning only relative to $\mathscr{I}$, in contrast to our system $\mathscr{H}$, whose rules and axioms are valid for all interpretations. Further discussion of the system $\mathscr{D}$ appears in § 6.

THEOREM 1. *Axioms and rules* 1) *through* 11) *for the system* $\mathscr{H}$ *are valid.*

A formal proof in the system $(\mathscr{H}, \mathscr{D})$ consists of a finite sequence of formulas, each either of the form $P\{A\}Q$, or P, with P, Q in $\mathscr{L}_2$ and A a statement in $\mathrm{Al}[\mathscr{L}_1, \mathscr{L}_2]$. Each formula is either an axiom of $\mathscr{H}$, a formula deducible in $\mathscr{D}$, or follows from earlier formulas in the sequence by one of the rules of $\mathscr{H}$. We use the notation $\vdash_{\mathscr{H},\mathscr{D}} P\{A\}Q$ to mean $P\{A\}Q$ is provable in this sense.

COROLLARY. *If* $\mathscr{D}$ *is sound relative to* $\mathscr{I}$ *and* $\vdash_{\mathscr{H},\mathscr{D}} P\{A\}Q$, *then* $\models_{\mathscr{M}[\mathscr{I}]} P\{A\}Q$.

The main tool in the proof of Theorem 1 is "induction on the definition of Comp." This principle allows us to conclude an assertion of the form "for all A, s, δ, π, if the sequence $\mathrm{Comp}(A, s, \delta, \pi)$ is finite, then it has a certain property $P(A, s, \delta, \pi)$." To make this conclusion, it is sufficient to prove, for each of the eight cases in the definition of Comp, that if A takes the form of the case, then $\mathrm{Comp}(A, s, \delta, \pi)$ satisfies P (provided it is finite), assuming as an induction hypothesis that $\mathrm{Comp}(A', s', \delta', \pi')$ (and sometimes $\mathrm{Comp}(A'', s'', \delta'', \pi'')$) satisfy P (provided they are finite), where the latter are the occurrence(s) of Comp that appear on the right hand side of the case. The justification of this principle comes directly from the definition of Comp, by a simple induction on the length of the sequence $\mathrm{Comp}(A, s, \delta, \pi)$. The principle is used in the proofs of Lemmas 1 and 3 below.

We will now prove Theorem 1 for rules 1), 9), and 10). The other cases are straightforward. For the rest of this section, "true" means true in some arbitrary model $\mathscr{M}$. Starting with rule 1, we assume the premise

$$\text{(i)} \qquad \models P\frac{y}{x}\{\textbf{begin } D^*; A^* \textbf{ end}\}Q\frac{y}{x}$$

where y is not free in P or Q, and is not in the free set of D^* or A^*.

In order to verify the conclusion of the rule, we assume $P(s, \delta)$ is true, where δ assigns registers to all relevant variables. Now let $s' = \mathrm{Out}(\textbf{begin new } x; D^*; A^* \textbf{ end}, s, \delta, \pi)$, where π makes the proper assignments to procedure names. According to the appropriate clause in the definition of Comp, we have

$$\text{(ii)} \qquad s' = \mathrm{Out}(\textbf{begin } D^*; A^* \textbf{ end}, s, \delta', \pi)$$

where δ' agrees with δ everywhere except at the variable x, to which δ' assigns a new register. Now if we define the assignment δ'_1 by

$$\delta'_1(z) = \begin{cases} \delta'(z) & \text{if } z \neq y, \\ \delta(x) & \text{if } z = y, \end{cases}$$

then δ'_1 is one to one (because $\delta(x)$ is not in the range of δ'), and furthermore $P(s, \delta)$ has the same truth value as

$$P\frac{y}{x}(s, \delta'_1),$$

namely true. (This is because y has no occurrence in P, x has no occurrence in

$$P\frac{y}{x},$$

and $\delta(x) = \delta'_1(y)$.) By our premise (i), we conclude

$$Q\frac{y}{x}(s'_1, \delta'_1)$$

is true, where

(iii) $$s'_1 = \text{Out}(\textbf{begin}\ D^*;\ A^*\ \textbf{end}, s, \delta'_1, \pi).$$

LEMMA 1. *If δ_1 and δ_2 are two variable assignments which agree on the free set of a statement A, and if the largest-numbered registers in the ranges of δ_1 and δ_2 are the same, then* $\text{Comp}(A, s, \delta_1, \pi) = \text{Comp}(A, s, \delta_2, \pi)$, *provided the computations are finite.*

The proof is by induction on the definition of Comp. All clauses except that for variable declarations are immediate, because δ does not enter into the definition, and the exception is also easily handled.

We notice that δ' and δ'_1 satisfy the hypotheses of the lemma for the statement **begin** D^*; A^* **end**, (because by our assumptions on the rule 1) y has no free occurrence in D^* or A^*), so that it follows from (ii) and (iii) that $s' = s'_1$. Therefore

$$Q\frac{y}{x}(s', \delta'_1)$$

is true and hence $Q(s', \delta)$ holds by the same reasoning that showed

$$P\frac{y}{x}(s, \delta'_1) \Leftrightarrow P(s, \delta).$$

This establishes the conclusion of the rule 1) and completes the proof of the validity of 1).

The rule 9) of parameter substitution is worth verifying in some detail. Assume the premises to the rule hold, so that

(1) $$\models P\{\textbf{call}\ p(\bar{x}' : \bar{v}')\}Q$$

Let us use the abbreviation

(2) $$\sigma = \frac{\bar{u}, \bar{e}}{\bar{x}', \bar{v}'}$$

and assume

(3) $$P\sigma(s, \delta)$$

holds for some δ which assigns registers to all relevant free variables. If we set

$$s' = \mathrm{Out}(\textbf{call}\ p(\bar{u} : \bar{e}), s, \delta, \pi), \tag{4}$$

and assume s' is defined, and $\pi(p) = \langle K, (\bar{x} : \bar{v})\rangle$, then by the definition of Comp for procedures we have

$$s' = \mathrm{Out}\left(K\frac{\bar{u}, \bar{e}}{\bar{x}, \bar{v}}, s, \delta, \pi\right). \tag{5}$$

Suppose

$$\bar{x}' = x_1', \cdots, x_m', \quad \bar{u} = u_1, \cdots, u_m, \quad \bar{v}' = v_1', \cdots, v_n', \quad \bar{e} = e_1, \cdots, e_n. \tag{6}$$

In order to use our premise (1), we must find a new pair (s_1, δ_1) such that

a) $s_1(\delta_1(z)) = s(\delta(z))$ for all $z \notin (\bar{x}', \bar{v}')$ if both $\delta(z)$ and $\delta_1(z)$ are defined,

b) $s_1(\delta_1(x_i')) = s(\delta(u_i))$, $i = 1, \cdots, m$,

c) $s_1(\delta_1(v_i')) = e_i(s, \delta)$, $i = 1, \cdots, n$,

d) $s_1(X_{m_1+i}) = s(X_{m+i})$, $i = 1, 2, \cdots$, where X_{m_1} (respectively X_m) is the highest indexed register in the range of δ_1 (respectively δ).

Let us say (s_1, δ_1) is *matched* to (s, δ) relative to σ if a)–d) are satisfied. There is no difficulty in finding such an (s_1, δ_1), since the variables in $(\bar{x}', \bar{v}')$ are all distinct.

LEMMA 2. *If (s_1, δ_1) is matched to (s, δ) relative to*

$$\sigma = \frac{\bar{u}, \bar{e}}{\bar{x}', \bar{v}'},$$

and R is an assertion (of $\mathscr{L}_2$) and e is an expression (term of $\mathscr{L}_1$), then

$$R\sigma(s, \delta) \equiv R(s_1, \delta_1), \quad \textit{and} \quad (e\sigma)(s, \delta) = e(s_1, \delta_1)$$

(assuming δ and δ_1 assign registers to all the free variables of $R\sigma$ and $e\sigma$).

Proof. We have

$$R\sigma(s, \delta) \equiv R\frac{\cdots s(\delta(u_i)) \cdots e_i(s, \delta) \cdots s(\delta(z))}{\cdots x_i' \cdots\cdots v_i' \cdots\cdots z},$$

and, using equations a)–c),

$$R(s, \delta_1) \equiv R\frac{\cdots s(\delta(u_i) \cdots e_i(s, \delta) \cdots s(\delta(z))}{\cdots x_i' \cdots\cdots v_i' \cdots\cdots z}.$$

This establishes $R\sigma(s, \delta) \equiv R(s_1, \delta_1)$, and the equation in the lemma is established in the same way.

LEMMA 3. *Suppose (s_1, δ_1) is matched to (s, δ) relative to*

$$\sigma = \frac{\bar{u}, \bar{e}}{\bar{x}', \bar{v}'},$$

and $s' = \mathrm{Out}(A\sigma, s, \delta, \pi)$, and $s_1' = \mathrm{Out}(A, s_1, \delta_1, \pi)$, where A is any statement such that $p(\bar{x}' : \bar{v}')$ **proc** *A could be a legal procedure declaration for a legal statement* **call** *$p(\bar{u} : \bar{e})$. Then (s_1', δ_1) is matched to (s', δ) relative to σ.*

The hypotheses of the lemma imply that no variable in $(\bar{u}, \bar{e})$ (except possibly one in $(\bar{x}', \bar{v}')$) occurs in the free set of A, and no variable in $\bar{v}'$ occurs on the left

side of any assignment statement in A or to the left of the colon in any procedure call statement of A. Also, no variable in $(\bar{x}' : \bar{v}')$ occurs globally in any procedure declaration of a procedure which could be activated by executing A.

We wish to apply the lemma for the case A is

$$K\frac{\bar{x}', \bar{v}'}{\bar{x}, \bar{v}}.$$

The hypotheses of the lemma are satisfied in this case because $p(\bar{x} : \bar{v})$ **proc** K is a legitimate procedure declaration, and both $(\bar{u} : \bar{e})$ and $(\bar{x}' : \bar{v}')$ are legitimate sets of actual parameters for p. These hypotheses were stated explicitly because the proof is by induction on the definition of Comp, and it is important to check that the induction hypothesis can be legally applied to the appropriate statement A' in each case. It seems that all hypotheses stated are necessary for one case or another, in order to push the argument through.

For most cases in the definition of Comp the argument is straightforward. For example, in the case of conditional and while statements, we can apply Lemma 2 to see that $R\sigma(s, \delta) \equiv R(s_1, \delta_1)$, so the same branch of the conditional and the same case of the while definition will apply for both A and $A\sigma$.

The case of a procedure call statement is more subtle. Suppose A is **call** $p'(\bar{u}' : \bar{e}')$, where the procedure declaration for p' is $p'(\bar{x}'' : \bar{v}'')$ **proc** K'. In order to apply the induction hypothesis, we must verify that

$$A' = K'\frac{\bar{u}', \bar{e}'}{\bar{x}'', \bar{v}''}$$

satisfies the hypotheses of the lemma. First, the free set of A' can have no variable in $(\bar{u}, \bar{e})$ (other than one in $(\bar{x}', \bar{v}')$), by the hereditary nature of the definition of "free set". Second, no variable v_i' in $\bar{v}'$ can occur either in $\bar{u}'$ or globally in the procedure declaration of p', so by our restrictions on procedure call statements, v_i' cannot occur to the left of any assignment statement or to the left of the colon in any procedure call statement in A'. Third, A' satisfies the final hypothesis of Lemma 3 because A does. Hence the induction hypothesis applies to A'. By definition of Comp,

$$s_1' = \mathrm{Out}(A, s_1, \delta_1, \pi) = \mathrm{Out}(A', s_1, \delta, \pi)$$

and

$$s' = \mathrm{Out}(A\sigma, s, \delta, \pi) = \mathrm{Out}\left(K'\frac{\bar{u}'\sigma, \bar{e}'\sigma}{\bar{x}'', \bar{v}''}, s, \delta, \pi\right).$$

Since no variable in $\bar{x}'$ or $\bar{v}'$ occurs globally in the procedure declaration for p', it follows that

$$K'\frac{\bar{u}'\sigma, \bar{e}'\sigma}{\bar{x}'', \bar{v}''} = K'\frac{\bar{u}', \bar{e}'}{\bar{x}'', \bar{v}''}\sigma.$$

Therefore

$$s' = \mathrm{Out}(A'\sigma, s, \delta, \pi).$$

Hence, by the induction hypothesis, (s_1', δ_1) is matched to (s', δ) relative to σ. This completes the case of procedure call statements.

In the case of an assignment statement, A is simply $x := e$, $A\sigma$ is $x\sigma := e\sigma$,

$$s'(X_i) = \begin{cases} s(X_i) & \text{if } \delta(x\sigma) \neq X_i, \\ e\sigma(s, \delta) & \text{if } \delta(x\sigma) = X_i \end{cases}$$

and

$$s_1'(X_i) = \begin{cases} s_1(X_i) & \text{if } \delta_1(x) \neq X_i, \\ e(s_1, \delta_1) & \text{if } \delta_1(x) = X_i. \end{cases}$$

To check condition a) for (s_1', δ_1), (s', δ) we note that if z is not x then z is unchanged by the assignment, so a) for the pair (s_1', δ_1), (s', δ) follows from our assumption a) for the pair (s_1, δ_1), (s, δ). If z is x, then $s_1'(\delta_1(z)) = e(s_1, \delta_1) = e\sigma(s, \delta)$ (by Lemma 2) $= s'(\delta(z\sigma)) = s'(\delta(z))$. Condition b) is proved similarly, but it is necessary to use the facts that the u_i are distinct and that the assignment statement A cannot be $u_i := e$, (unless u_i is some x_j') because no variable in $(\bar{u}, \bar{e})$ (other than one in $(\bar{x}', \bar{v}')$) is global in A. To verify condition c), we note that v_i' is not on the left side of the assignment statement A, and no variable in e_i is on the left side of the assignment statement $A\sigma$ by our restrictions that actual parameters are not global in A and no u_j can occur in e_i. Therefore the values of v_i' and e_i remain unchanged by A and $A\sigma$, respectively. Condition d) is obviously unaffected by the assignment statement.

The remaining troublesome case in the proof of Lemma 3 is that of variable declaration in the definition of Comp. In this case, A is **begin new** x; D^*; A^*; **end**, and we again assume (s_1, δ_1) is matched to (s, δ) relative to σ. We can assume that the variable x being declared does not occur in $(\bar{u}, \bar{e})$, because of our convention for renaming locally declared variables in $A\sigma$ explained before the definition of Comp. If x is in $(\bar{x}', \bar{v}')$, then let σ' be σ with the substitution for x deleted. Otherwise, let $\sigma' = \sigma$. Note that $A\sigma = A\sigma'$, because x is not free in A. We claim (s_1, δ_1') is matched to (s, δ') relative to σ', where δ_1' and δ' are the variable assignments determined from δ_1 and δ, respectively, in the first clause in the definition of Comp. The claim is staightforward to verify, using in particular condition d) from the definition of "(s_1, δ_1) is matched to (s, δ) relative to σ," to verify condition a). From the claim and the easily verified fact that the induction hypothesis applies to $A' =$ **begin** D^*; A^* **end** we can conclude (s_1', δ_1') is matched to (s', δ') relative to σ'. From this we can conclude (s_1', δ_1) is matched to (s', δ) relative to σ, where we must also use the fact that (s_1, δ_1) is matched to (s, δ) relative to σ and the contents of the register $\delta_1(x)$ (respectively $\delta(x)$) remains unchanged during the computation of $A\sigma$ under δ_1 and s_1 (respectively A under δ and s). This completes the proof of Lemma 3.

Using Lemmas 2 and 3 it is easy to complete the proof of the validity of the parameter substitution rule. We assume equations (1) to (6) hold, and select (s_1, δ_1) to match (s, δ) relative to σ. By Lemma 2 and our assumption (3) that $P\sigma(s, \delta)$ is true, it follows that $P(s_1, \delta_1)$ is true. If we set

$$s_1' = \text{Out}(\textbf{call } p(\bar{x}' : \bar{v}'), s_1, \delta_1, \pi), \tag{7}$$

then by our assumption (1), $Q(s_1', \delta_1)$ is true. On the other hand, by (7) and the definition of Comp we have

$$s_1' = \mathrm{Out}\left(K\frac{\bar{x}', \bar{v}'}{\bar{x}, \bar{v}}, s_1, \delta_1, \pi\right). \tag{8}$$

If we now take A in Lemma 3 to be

$$K\frac{\bar{x}', \bar{v}'}{\bar{x}, \bar{v}}$$

and note that then

$$K\frac{\bar{u}, \bar{e}}{\bar{x}, \bar{v}} = A\sigma,$$

it follows (using (5) and (8)) that (s_1', δ_1) is matched to (s', δ) relative to σ. Hence by Lemma 2, $Q\sigma(s', \delta)$ is true..This establishes the truth of the conclusion of rule 9), and hence the validity of rule 9).

Let us now verify rule 10), the rule of variable substitution. This rule can just as easily be verified in the more general setting

$$\frac{P\{A\}Q}{P\sigma\{A\}Q\sigma}, \quad \text{where } \sigma = \frac{\bar{z}'}{\bar{z}}$$

is a substitution of expressions for variables such that no variable in $\bar{z}$ or $\bar{z}'$ occurs in the free set of A. The reason we selected a special case of this rule for rule 10) is that this special case is precisely what is needed to prove completeness of the system $\mathcal{H}$.

To verify the more general rule, assume the premise $P\{A\}Q$, and suppose $P\sigma(s, \delta)$ is true for some state s and variable assignment δ. Let $\bar{z} = z_1, \cdots, z_k$ and $\bar{z}' = z_1', \cdots, z_k'$. Let the state s_1 be given by

$$s_1(\delta(y)) = s(\delta(y)) \quad \text{for all } y \text{ not in } \bar{z} \text{ for which } \delta(y) \text{ is defined,}$$

$$s_1(\delta(z_i)) = z_i'(s, \delta), \quad 1 \leqq i \leqq k,$$

$$s_1(X_j) = s(X_j) \quad \text{for all registers } X_j \text{ not in the range of } \delta.$$

Then (s_1, δ) is matched to (s, δ) relative to σ, in the sense defined before Lemma 2. Hence by Lemma 2, $P(s_1, \delta)$ is true. Let $s' = \mathrm{Out}(A\sigma, s, \delta, \pi)$ and $s_1' = \mathrm{Out}(A, s_1, \delta, \pi)$. Then the hypotheses of Lemma 3 are easily verified, so (s_1', δ) is matched to (s', δ) relative to σ. Since $P(s_1, \delta)$ is true and $P\{A\}Q$ holds, $Q(s_1', \delta)$ is true. By Lemma 2, $Q\sigma(s', \delta)$ is true. Since $A\sigma = A$, this establishes the truth of $P\sigma\{A\}Q\sigma$, and hence rule 10) is valid.

All other rules can be verified directly from the corresponding clause in the definition of Comp. This completes the proof of Theorem 1.

Using Lemma 3 we can give a short proof that the truth of a formula $P\{A\}Q$, as defined at the beginning of this section, is independent of the choice of δ. Of course it would have been more logical to prove this immediately after the definition, and in fact Lemmas 1 through 3 could have been proven there. However, Lemma 3 would have been very difficult to motivate.

LEMMA 4. *The truth of $P\{A\}Q$ is independent of the choice of δ.*

Proof. Suppose δ_1 and δ_2 both assign registers to the free variables of formulas P and Q, and the free set of statement A, and π makes the proper procedure assignments for A. Suppose for all states s_1 and s_1', if $P(s_1, \delta_1)$ is true and $s_1' = \text{Out}(A, s_1, \delta_1, \pi)$, then $Q(s_1', \delta_1)$ is true. We wish to show the same can be said with δ_1 replaced by δ_2. Hence let s_2 be any state, and suppose $P(s_2, \delta_2)$ is true. Choose a state s_1 such that (s_1, δ_1) is matched to (s_2, δ_2) relative to the empty (or identity) substitution σ_0. Then by Lemma 2, $P(s_1, \delta_1)$ is true. Let $s_2' = \text{Out}(A\sigma_0, s_2, \delta_2, \pi)$ (note that $A\sigma_0 = A$). Then the hypotheses of Lemma 3 are satisfied, since the lists $\bar{u}$, $\bar{e}$, $\bar{x}$, $\bar{v}$ are all empty, so (s_1, δ_1) is matched to (s_2, δ_2) relative to σ_0. By Lemma 2, $Q(s_1', \delta_1) \equiv Q(s_2', \delta_2)$, so $Q(s_2', \delta_2)$, is true. This completes the proof of Lemma 4.

6. The question of completeness of the rules. The corollary to Theorem 1 states that if $\mathscr{D}$ is a sound deductive system for the language $\mathscr{L}_2$ relative to an interpretation $\mathscr{I}$, and if $\vdash_{\mathscr{H},\mathscr{D}} P\{A\}Q$, then $P\{A\}Q$ is true in $\mathscr{M}[\mathscr{I}]$. We turn now to the converse question, and ask under what conditions every true formula $P\{A\}Q$ is provable in the system $(\mathscr{H}, \mathscr{D})$.

If we assume $\mathscr{D}$ is an axiomatic deductive system, then the formulas $P\{A\}Q$ provable in the system $(\mathscr{H}, \mathscr{D})$ are recursively enumerable. On the other hand, the formula true $\{A\}$ false is true in $\mathscr{M}[\mathscr{I}]$ iff A fails to halt for all initial values of its global variables. Therefore the true formulas cannot be recursively enumerable in case $\mathscr{L}_1$, $\mathscr{L}_2$ and $\mathscr{I}$ are such that the halting problem for $\text{Al}[\mathscr{L}_1, \mathscr{L}_2]$ is recursively unsolvable. In particular, we have

THEOREM 2. *If $\mathscr{L}_1$ is $\mathscr{L}_N$ (or $\mathscr{L}_+$: i.e., $\mathscr{L}_N$ without multiplication) and $\mathscr{D}$ is an axiomatic deductive system for $\mathscr{L}_2$, then $(\mathscr{H}, \mathscr{D})$ is incomplete in the sense that there is a formula $P\{A\}Q$ true in $\mathscr{M}[\mathscr{I}]$, but such that not $\vdash_{\mathscr{H},\mathscr{D}} P\{A\}Q$, where $\mathscr{I}$ includes the standard interpretation in the natural numbers for $\mathscr{L}_1$.*

On the other hand, one has a feeling that the axioms and rules 1)–11) (or small modifications of 1)–11)) are complete in some sense, and the incompleteness is probably due to the incompleteness of the system $\mathscr{D}$. But there is another way in which the system can fail to be complete, and that it is if the assertion language $\mathscr{L}_2$ is not powerful enough to express invariants for the loops. Let us fix the language $\mathscr{L}_1$, $\mathscr{L}_2$ and the interpretation $\mathscr{I}$ with domain $\mathscr{D}$. Suppose $P \in \mathscr{L}_2$ and A is a statement of $\text{Al}[\mathscr{L}_1, \mathscr{L}_2]$, and $\bar{x} = (\bar{x}_1, \cdots, x_n)$ is a list of all variables occurring either free in P or in the free set of A. Then we say the *post relation corresponding to* P and A is the relation $\bar{Q}(x_1, \cdots, x_n)$ on D such that $\bar{Q}(d_1, \cdots, d_n)$ is true iff there is a state s and variable assignment δ to $x_1, \cdots, x_n$ such that $d_i = s'(\delta(x_i))$, $i = 1, \cdots, n$, where $s' = \text{Out}(A, s, \delta, \pi)$, $P(s, \delta)$ is true, and π is appropriate to the context of A in the program. The formula Q in $\mathscr{L}_2$ *expresses* the relation $\bar{Q}$ iff Q has free variables $x_1, \cdots, x_n$, and

$$\mathscr{I}\left(Q\frac{d_1, \cdots, d_n}{x_1, \cdots, x_n}\right) \Leftrightarrow \bar{Q}(d_1, \cdots, d_n)$$

for all $d_1, \cdots, d_n \in D$. We let 'post(P, A)' denote a particular formula in $\mathscr{L}_2$ (say the one with the least Gödel number) which expresses the post relation corresponding to P and A.

DEFINITION. The language $\mathscr{L}_2$ is *expressive* (relative to $\mathscr{L}_1$ and $\mathscr{I}$) iff

(i) "=" is in $\mathscr{L}_1$ and receives its standard interpretation in $\mathscr{I}$,

(ii) For every formula P in $\mathscr{L}_2$ and statement A there is a formula Q in $\mathscr{L}_2$ which expresses the post relation corresponding to P and A.

LEMMA 5. *$\mathscr{L}_N$ is expressive (relative to $\mathscr{L}_N$, $\mathscr{L}_N$, and $\mathscr{I}_N$).*

Proof. Given P and A, then roughly speaking the post relation $\bar{Q}$ is true of numbers $a_1, \cdots, a_n$ iff there are initial values $b_1, \cdots, b_n$ satisfying P such that A will terminate with these as initial values, and the final values of $(x_1, \cdots, x_n)$ are $(a_1, \cdots, a_n)$. Since A describes a partial recursive function and partial recursive functions are expressible in $\mathscr{L}_N$, the lemma follows.

We remark that $\mathscr{L}_+$ (i.e., $\mathscr{L}_N$ without multiplication) is *not* expressive (relative to $\mathscr{L}_+$, $\mathscr{L}_+$, and $\mathscr{I}_+$) because every recursively enumerable (r.e.) set is the post condition corresponding to $0 = 0$ and some A. On the other hand, truth in $\mathscr{L}_+$ is decidable (by Presburger's result), so not every r.e. set is expressible in $\mathscr{L}_+$.

Let us say that the proof system $\mathscr{D}$ for $\mathscr{L}_2$ is *semantically complete relative to* $\mathscr{I}$ iff: a formula P of $\mathscr{L}_2$ is provable in $\mathscr{D}$ iff its universal closure is true under $\mathscr{I}$. Of course by the Gödel Incompleteness Theorem, no axiomatic system $\mathscr{D}$ for number theory can be semantically complete, but for the purpose of stating a completeness theorem for $\mathscr{H}$, we shall assume we have a complete nonaxiomatizable system.

THEOREM 3 (Completeness of $\mathscr{H}$). *Let $\mathscr{T}$ be a semantically complete proof system for $\mathscr{L}_2$ (relative to $\mathscr{I}$) and suppose $\mathscr{L}_2$ is expressive relative to $\mathscr{L}_1$ and $\mathscr{I}$. Then $\vdash_{\mathscr{H},\mathscr{T}} P\{A\}Q$ whenever $\models_{\mathscr{M}} P\{A\}Q$.*

COROLLARY. *Let $\mathscr{T}$ be a complete (noneffective) proof system for $\mathscr{L}_N$. Then $\vdash_{\mathscr{H},\mathscr{T}} P\{A\}Q$ if and only if $P\{A\}Q$ is true in $\mathscr{M}[\mathscr{I}_N]$.*

Proof of theorem. Given a statement A (part of a larger program) let A' be the result of substituting all procedure bodies, with formal parameters replaced by actual parameters and local variables renamed where necessary, for procedure calls repeatedly until no procedure calls remain. This process terminates because of our outlawing of recursive procedures. The theorem is proved by induction on the sum of the length of A' and the number of procedure body substitutions necessary to convert A to A'. If A is not a procedure call statement, then exactly one of the rules 1)–7) can be applied nicely (sometimes with rule 11), the rule of consequence) to prove A from previously proved statements. Rules 8), 9), and 10) are needed for procedure call statements. We will discuss several of the more interesting cases.

a) *Compound statements.* Suppose $P\{\textbf{begin } A\,;\, A^* \textbf{ end}\}R$ is true in $\mathscr{M}[\mathscr{I}]$. Let Q be a formula expressing the post relation corresponding to P and A. Then by the definitions involved, it is easy to see that $P\{A\}Q$ and $Q\{\textbf{begin } A^* \textbf{ end}\}R$ are both true, and so both are provable in the system $(\mathscr{H}, \mathscr{T})$ by the induction hypothesis. Therefore by rule 3),

$$\vdash_{\mathscr{H},\mathscr{T}} P\{\textbf{begin } A\,;\, A^* \textbf{ end}\}R.$$

The case $P\{\textbf{begin end}\}Q$ is handled by using rule 4) and the rule of consequence.

b) *Assignment statements.* Suppose $P\{x := e\}Q$ is true. Then the universal

closure of

$$P \supset Q\frac{e}{x}$$

must be true under the interpretation $\mathcal{I}$, so

$$\vdash_{\mathcal{T}} P \supset Q\frac{e}{x}.$$

But

$$\vdash_{\mathcal{K},\mathcal{T}} Q\frac{e}{x}\{x := e\}Q$$

by axiom 5), and $\vdash_{\mathcal{T}} Q \supset Q$, so $\vdash_{\mathcal{K},\mathcal{T}} P\{x := e\}Q$ by the rule of consequence.

c) *While statements.* Suppose $P\{\textbf{while } R \textbf{ do } A \textbf{ od}\}Q$ is true in $\mathcal{M}[\mathcal{I}]$. In order to apply rule 7), the rule of **while** statements, we must find a loop invariant P_1 with the properties that $P_1 \,\&\, R\{A\}P_1$ is true, and (the universal closures of) $P \supset P_1$ and $(P_1 \,\&\, \neg R) \supset Q$ are true. The induction hypothesis can then be applied to assume $\vdash_{(\mathcal{K},\mathcal{T})} P_1 \,\&\, R\{A\}P_1$, and by the completeness of $\mathcal{T}$, $\vdash_{\mathcal{T}} P \supset P_1$ and $\vdash_{\mathcal{T}} (P_1 \,\&\, \neg R) \supset Q$. Hence by rules 7) and 11), $\vdash_{\mathcal{K},\mathcal{T}} P\{\textbf{while } R \textbf{ do } A \textbf{ od}\}Q$.

Let $y_1, \cdots, y_n$ be a list of the variables in the free set of A, together with the free variables in P, R, and Q. We will construct the loop invariant P_1 with free variables $y_1, \cdots, y_n$ such that $P_1(d_1, \cdots, d_n)$ holds iff there are initial values $d_1', \cdots, d_n'$ for $y_1, \cdots, y_n$ such that $P(d_1', \cdots, d_n')$ is true, and after some finite number of passes through the while loop (i.e. after A has been executed some finite number of times with the condition R satisfied before each time) the values of $y_1, \cdots, y_n$ will be $d_1, \cdots, d_n$. More precisely, P_1 is equivalent to the infinite disjunction $Q_1 \vee Q_2 \vee \cdots$, where Q_1 is P and Q_{i+1} is $\text{post}(Q_i \,\&\, R, A)$, $i = 1, 2, \cdots$. The reader can easily verify that if such a finite P_1 can be found, then the conditions in the previous paragraph are satisfied. But in fact, P_1 is just $\exists z_1 \cdots \exists z_n P_2$, where P_2 expresses the post relation corresponding to P and

$$\textbf{while } (R \,\&\, (y_1 \neq z_1 \vee y_2 \neq z_2 \vee \cdots \vee y_n \neq z_n)) \textbf{ do } A \textbf{ od},$$

and $z_1, \cdots, z_n$ are new variables. P_2 is in the language $\mathcal{L}_2$, by our assumption that $\mathcal{L}_2$ is expressive. This completes case c).

d) *Procedure calls.* Suppose $P\{\textbf{call } p(\bar{u} : \bar{e})\}Q$ is true, where the procedure declaration for p is $p(\bar{x} : \bar{v})$ **proc** K. By definition of Comp,

$$P\left\{K\frac{\bar{u}, \bar{e}}{\bar{x}, \bar{v}}\right\}Q$$

is true. The naive argument, which only works if there are no variable clashes, is the following. Since

$$\bar{v} = \bar{e} \,\&\, P\frac{\bar{x}}{\bar{u}}\{K\}Q\frac{\bar{x}}{\bar{u}}$$

is true, it is provable by the induction hypothesis. By the rule of procedure calls

(rule 8)),

$$\bar{v} = \bar{e} \;\&\; P\frac{\bar{x}}{\bar{u}}\{\textbf{call}\, p(\bar{x} : \bar{v})\}\, Q\frac{\bar{x}}{\bar{u}}$$

is provable. By the rule of parameter substitution (rule 9)), $\bar{e} = \bar{e}$ & $P\{\textbf{call}\, p(\bar{u} : \bar{e})\}Q$ is provable. Finally $P\{\textbf{call}\, p(\bar{u} : \bar{e})\}Q$ is provable by the rule of consequence.

The difficulties with this argument are first, the formal and actual parameters might have variables in common, and second, P and Q may have occurrences of the formal parameters even if the first condition does not hold. To handle the second problem, we let τ be a substitution which assigns distinct new variables to all variables in $(\bar{x}, \bar{v})$ which do not occur in $(\bar{u}, \bar{e})$. We will concentrate on showing $P\tau\{\textbf{call}\, p(u : e)\}Q\tau$ is provable, and then apply the rule of variable substitution. Since the variables renamed by τ do not occur in the free set of **call** $p(\bar{u} : \bar{e})$, and since $P\{\textbf{call}\, p(\bar{u} : \bar{e})\}Q$ is true, it is intuitively clear (and follows formally from Lemmas 2 and 3) that

(1′) $$\models P\tau\{\textbf{call}\, p(\bar{u} : \bar{e})\}Q\tau.$$

To handle the first problem (along with the second problem) let $\bar{f}$ be a list of the variables occurring in the expressions $\bar{e}$, and let $\bar{f}'$ be a list of distinct new variables of the same length, and let $\bar{e}'$ be the result of substituting these new variables for the old in $\bar{e}$. By definition of Comp and (1′),

$$P\tau\left\{K\frac{\bar{u}, \bar{e}}{\bar{x}, \bar{v}}\right\}Q\tau$$

is true. We will show $P_1\{K\}Q_1$ is true, where

$$P_1 \text{ is } \bar{v} = \bar{e}' \;\&\; P\tau\frac{\bar{x}, \bar{f}'}{\bar{u}, \bar{f}}, \quad \text{and} \quad Q_1 \text{ is } Q\tau\frac{\bar{x}, \bar{f}'}{\bar{u}, \bar{f}}.$$

Suppose s_1 and δ_1 are state and variable assignments such that $P_1(s_1, \delta_1)$ is true, and δ_1 assigns registers to the free set of K and the free variables of P_1 and Q_1, and to no other variables. Thus δ_1 is not defined for any variable in $(\bar{u}, \bar{e})$ unless that variable is also in $(\bar{x}, \bar{v})$. Hence we can find s and δ so that

A) $s(\delta(z)) = s_1(\delta_1(z))$ for all variables $z \notin (\bar{x}, \bar{v})$, if $\delta_1(z)$ is defined,

B) $s(\delta(u_i)) = s_1(\delta_1(x_i))$, $i = 1, \cdots, m$,

C) $s(\delta(f)) = s_1(\delta_1(f'))$ for each variable f in $\bar{e}$ and corresponding f' in $\bar{e}'$,

and further, condition d), stated before Lemma 2, is satisfied. Hence (s_1, δ_1) is matched to (s, δ) relative to

$$\sigma = \frac{\bar{u}, \bar{e}}{\bar{x}, \bar{v}}.$$

[Condition c) is satisfied by condition C) above, and the fact that $P_1(s_1, \delta_1)$ is true, and P_1 includes the conjunct $\bar{v} = \bar{e}'$.] Thus by Lemma 3, (s_1', δ_1) is matched to (s', δ) relative to σ, where $s' = \text{Out}(K\sigma, s, \delta, \pi)$ and $s_1' = \text{Out}(K, s_1, \delta_1, \pi)$. Now by comparing the values (s_1, δ_1) given to variables in P_1 with those (s, δ) gives to the variables in $P\tau$, it is easy to see $P\tau(s, \delta)$ is true. Since $P\tau\{K\sigma\}Q\tau$ is true, it follows

that $Q\tau(s', \delta)$ is true. Again by comparing values assigned to variables, and noting property C) above holds when s and s_1 are replaced by s' and s_1' respectively (since K and $K\sigma$ leave the values of $\bar{f}'$ and $\bar{f}$ unchanged), we see that $Q_1(s_1', \delta_1)$ is true. This completes the proof that $P_1\{K\}Q_1$ is true.

By the induction hypothesis, $P_1\{K\}Q_1$ is provable. By the rule of procedure calls, $P_1\{\textbf{call } p(\bar{x} : \bar{v})\}Q_1$ is provable. By the rule 9) of parameter substitution, we can rename $\bar{v}$ by $\bar{v}'$ and rename $\bar{x}$ by $\bar{x}'$ (these new variables are distinct from the ones τ assigns) to obtain that

$$\bar{v}' = \bar{e}' \;\&\; P\tau\frac{\bar{x}', \bar{f}'}{\bar{u}, \bar{f}}\left\{\textbf{call } p(\bar{x}' : \bar{v}')\right\}Q\tau\frac{\bar{x}', \bar{f}'}{\bar{u}, \bar{f}}$$

is provable. Again by the rule 9) of parameter substitution and the substitution

$$\frac{\bar{f}}{\bar{f}'},$$

we obtain that

$$\bar{v}' = \bar{e} \;\&\; P\tau\frac{\bar{x}'}{\bar{u}}\{\textbf{call } p(\bar{x}' : \bar{v}')\}Q\tau\frac{\bar{x}'}{\bar{u}}$$

is provable. Again by the rule 9) of parameter substitution and the substitution

$$\frac{\bar{u}, \bar{e}}{\bar{x}', \bar{v}'}$$

we obtain that

$$\bar{e} = \bar{e} \;\&\; P\tau\{\textbf{call } p(\bar{u} : \bar{e})\}Q\tau$$

is provable. Finally, conjunct $\bar{e} = \bar{e}$ can be removed by the rule of consequence, and the rule 10) of variable substitution with substitution τ^{-1} can be applied to prove $P\{\textbf{call } p(\bar{u} : \bar{e})\}Q$.

Acknowledgments. I am grateful to Jim Donahue and Bob Constable for reading and pointing out errors in earlier versions of this paper. Also, Gerry Gorelick, whose M.Sc. thesis [5] attempts to extend these results to the case of recursive procedures, spent a great deal of time thinking about the ideas presented here and his comments and criticisms were especially helpful.

REFERENCES

[1] S. A. COOK AND D. C. OPPEN, *An assertion language for data structures*, Conference Record of the Second ACM Symposium on Principles of Programming Languages (Palo Alto, CA), Jan. 1975, pp. 160–166.

[2] S. A. COOK, *Axiomatic and interpretive semantics for an ALGOL fragment*, Tech. Rep. 79, Dept. of Computer Sci., Univ. of Toronto, Feb. 1975.

[3] E. M. CLARKE, JR., *Completeness and incompleteness theorems for Hoare-like axiom systems*, Ph.D. thesis, Cornell Univ., Ithaca, NY, 1976.

[4] J. E. DONAHUE, *Complementary definitions of programming language semantics*, Computer Systems Research Group Tech. Rep. CSRG-62, Univ. of Toronto, Nov. 1975.

[5] G. A. GORELICK, *A complete axiomatic system for proving assertions about recursive and non-recursive programs*, Tech. Rep. 75, Dept. of Computer Sci., Univ. of Toronto, Feb. 1975.

[6] C. A. R. HOARE, *An axiomatic basis for computer programming*, Comm. ACM, 12 (1969), pp. 576–580.

[7] ———, *Procedures and parameters: An axiomatic approach*, Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., Springer-Verlag, Berlin, 1971, pp. 102–116.

[8] C. A. R. HOARE AND P. E. LAUER, *Consistent and complementary formal theories of the semantics of programming languages*, Acta Informat., 3 (1974), pp. 135–153.

[9] S. IGARASHI, R. L. LONDON AND D. C. LUCKHAM, *Automatic program verification I: A logical basis and its implementation*, Tech. Rep. AIM-200, Artificial Intelligence Laboratory, Stanford Univ., Stanford, CA, July 1973.

[10] P. E. LAUER, *Consistent formal theories of the semantics of programming languages*, Tech. Rep. TR 25.121, IBM Laboratory, Vienna, Nov. 1971.

[11] D. C. OPPEN, *On logic and program verification*, Tech. Rep. 82, Dept. of Computer Sci., University of Toronto, April 1975.

[12] D. C. OPPEN AND S. A. COOK, *Proving assertions about programs that manipulate data structures*, Proceedings of Seventh Annual ACM Symposium on Theory of Computing (Albuquerque, NM), May 1975, pp. 107–116.

[13] S. OWICKI, *A consistent and complete deductive system for the verification of parallel programs*, Proceedings Eighth Annual ACM Symposium on Theory of Computing (Hershey, PA), May 1976, pp. 73–86.

[14] V. R. PRATT, *Semantical considerations on Floyd–Hoare logic*, 17th Annual IEEE Symposium on Foundations of Computer Science (Houston, TX), Oct. 1976.

SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

AN EXTENSION OF COMPUTATIONAL DUALITY TO SEQUENCES OF BILINEAR COMPUTATIONS*

ROBERT L. PROBERT†

Abstract. An extension of an earlier result on the computational duality of the problem of computing sets of bilinear forms by a single bilinear algorithm is given to that of computing a set of multilinear forms by a sequence of bilinear algorithms. Such problems are called piecewise bilinear computations. It is shown that piecewise bilinear computations represented by dihedral permutations of the dimensions of a multiple matrix product, for example, have exactly the same computational complexity with respect to multiplication operations as the original multiple matrix multiplication problem, and additive complexity given by the additive complexity of the original problem plus the decrease in size of the new product matrix from that of the original product matrix. This result proves that duality preserves arithmetic optimality even for algorithms developed by such local optimization techniques as dynamic programming. Finally, it is noted that since the result is constructive, it yields a method for generating equicomplex new algorithms from a given one, even if the given algorithm is nonoptimal or "approximately optimal".

Key words. matrix multiplication, additive complexity, multilinear forms, duality, analysis of algorithms, dynamic programming, bilinear computations

Introduction. The discovery by Strassen [13] of an algorithm for multiplying two matrices of order n using a number of arithmetic operations proportional to $n^{2.81}$ provided considerable impetus to the study of the arithmetic complexity of straight-line computations of common functions. In particular, Strassen's algorithm belongs to a very simple class of algorithms called bilinear algorithms [6]. These algorithms are allowed to use as multiplication steps products of only a linear sum of elements of the left matrix by a linear sum of elements of the right matrix. Since such algorithms are not able to exploit commutativity of multiplication, these algorithms may be implemented recursively to multiply matrices. As well, there is at worst a linear increase in arithmetic cost of considering only bilinear algorithms rather than more powerful algorithms which are able to exploit multiplicative commutativity [2], [12], [14]. We will study only bilinear algorithms and sequences of such algorithms in this paper.

Algorithms in this class are amenable to representation by an ordered triple of directed acyclic graphs called computation graphs [11]. By utilizing simple dihedral graph operations, i.e., operations equivalent to sequences of rotations and reflections (as in dihedral permutations), it was shown that matrix multiplication problems represented by all permutations of a given triple of dimensions were of equal multiplicative complexity to the original problem [3], [9], [10] and of additive complexity equal to that of the original problem plus the decrease in product matrix size [7], [11]. This result is known as the "computational duality" or "symmetry" theorem for matrix multiplication problems.

In this paper, we study the specific problem of efficiently computing a given product of several matrices by a sequence of products of pairs of matrices where

* Received by the editors January 28, 1976, and in revised form July 26, 1976.

† Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada S7N 0W0. This work was supported by the National Research Council of Canada under Grant A8982.

each product is computed by a bilinear algorithm. A sequence of bilinear algorithms for computing such a sequence of pairwise matrix products will be called a piecewise bilinear algorithm. Our objective is to define dihedral transformations on such algorithms and to utilize these operations to prove an extension of computational duality to sequences of matrix multiplications.

For example, consider the problem of computing $A_{3\times 2}B_{2\times 2}C_{2\times 2}$ by a piecewise bilinear algorithm. There are two possible schemes for such algorithms, namely compute $(AB)C$ and compute $A(BC)$. By computational duality applied to lower bounds due to Hopcroft and Kerr as in [10], the computational complexities with respect to multiplications of performing the piecewise bilinear algorithms over say the integers are 22 and 18 respectively. Thus, the optimal piecewise bilinear algorithm for computing ABC is to compute BC using Strassen's algorithm, and $A(BC)$ using block multiplication. We say the piecewise bilinear complexity of computing ABC is 18.

For a given multiple matrix multiplication problem, a dynamic programming strategy such as in [1] can be used to find the optimal piecewise bilinear algorithm for that problem, given the costs of optimal bilinear algorithms for all two-matrix multiplication subproblems. For a problem of multiplying n matrices, the dynamic programming method uses time proportional to n^3. The method used in [4] finds an "approximately optimal" piecewise bilinear algorithm in linear time, i.e. an algorithm whose cost for the given problem is no worse than twice that of an optimal algorithm. We will show that optimality and approximate optimality are both preserved by the duality transformations we define.

For a different class of algorithms for multiplying several matrices in which all multiplication steps involve elements from all matrices, computational duality has been shown by de Pillis [5] and Dobkin [15].

Since piecewise optimization techniques cannot be applied to such algorithms, from a computational viewpoint that class of algorithms may appear somewhat unnatural. For example, four multiplications are required to compute $A_{2\times 1}B_{1\times 1}C_{1\times 1}$ if all multiplications must involve elements from each original matrix. Only three multiplications are needed over piecewise bilinear algorithms if some elementary optimization is performed, and the problem computed as $A(BC)$.

Basic definitions. More precise definitions of some of the terms below are given in [10]; they are sketched here to make the results in this paper somewhat self-contained.

An (m, n, p) *product* is the problem of multiplying two matrices with elements belonging to a (possibly noncommutative) ring with unit, an $m \times n$ matrix A by an $n \times p$ matrix B, to yield an $m \times p$ product matrix, C.

A *bilinear algorithm* α for computing (m, n, p) products is a finite sequence of steps $f_1, f_2, \cdots$, such that

(i) for each k, f_k is an element of A or of B, *or* $f_k = f_i$ "op" f_j for $i, j < k$ and "op" $\in$ {"+", "−", "*"}, *or* $f_k = r * f_i$ where $i < k$ and $r \in \{0, 1, -1\}$, and

(ii) whenever $f_k = f_i * f_j$, f_i and f_j must be linear forms (sums/differences) in elements of A and B respectively, and

(iii) for each element z in $C = AB$, there is at least one k such that $f_k = z$.

The classical algorithm and Strassen's algorithm for computing (2, 2, 2) products for example are both bilinear algorithms.

The problem of computing $(m_1, m_2, \cdots, m_{n+1})$ *products* is the problem of multiplying n matrices of dimensions $m_1 \times m_2, m_2 \times m_3, \cdots, m_n \times m_{n+1}$ to yield a matrix of dimension $(m_1 \times m_{n+1})$.

A *piecewise bilinear algorithm* α for computing $(m_1, \cdots, m_{n+1})$ products is an algorithm which consists of $n-1$ successive bilinear subalgorithms $(\alpha_1, \cdots, \alpha_{n-1})$ computing products of distinct pairs of matrices such that the final subalgorithm, α_{n-1}, yields the correct $m_1 \times m_{n+1}$ product matrix.

A *rotation* of a given $n+1$-tuple of matrix dimensions $(m_1, m_2, \cdots, m_{n+1})$ yields a new $n+1$-tuple $(m_{n+1}, m_1, m_2, \cdots, m_n)$. This is written as $(m_1, m_2, \cdots, m_{n+1})^R = (m_{n+1}, m_1, \cdots, m_n)$.

Similarly, we define a *reflection* operation D on $n+1$-tuples as $(m_1, m_2, \cdots, m_{n+1})^D = (m_{n+1}, m_n, \cdots, m_1)$.

A *dihedral permutation* of an $n+1$-tuple is any permutation consisting of an arbitrary sequence of rotations and reflections.

Finally, to simplify notation, we define a sequence of matrix dimension triples to describe the sequence of pairwise matrix products that make up the given piecewise bilinear algorithm. Let α be the algorithm given for computing $(m_1, \cdots, m_{n+1})$ products. Then, $S = \langle s_1, s_2, \cdots, s_{n-1} \rangle$ is said to be a *subcomputation sequence* for algorithm α where the order of appearance of the s_i's reflects the order of subcomputations of α, and for each i,

(i) $s_i = (p_{i_1}, p_{j_1}, p_{k_1})$ where $p_{i_1} = m_{i_2}$, $p_{j_1} = m_{j_2}$, $p_{k_1} = m_{k_2}$ and $i_2 < j_2 < k_2$,

(ii) if $s_i = (m_{i_1}, m_{i_2}, m_{i_3})$ and $i_2 - i_1 > 1$, there is a unique $j < i$ such that $s_j = (m_{i_1}, m_k, m_{i_2})$ and $i_1 < k < i_2$,

(iii) if $s_i = (m_{i_1}, m_{i_2}, m_{i_3})$ and $i_3 - i_2 > 1$, there is a unique $l < i$ such that $s_l = (m_{i_2}, m_q, m_{i_3})$ and $i_2 < q < i_3$, and

(iv) if $s_i = (m_{i_1}, m_{i_2}, m_{i_3})$ and $i_3 - i_1 = 2$, there is no $j < i$ such that $s_j = (m_k, m_{i_1}, m_l)$ or $s_j = (m_k, m_{i_3}, m_l)$.

It is not hard to show that a subcomputation sequence $S = \langle s_1, \cdots, s_{n-1} \rangle$ for computing $(m_1, \cdots, m_{n+1})$ products has the following properties:

(a) for each m_j, $1 < j < n+1$, there is exactly one i such that $s_i = (m_k, m_j, m_l)$, and

(b) $s_{n-1} = (m_1, m_q, m_{n+1})$ where $1 < q < n+1$, (since s_{n-1} is the triple describing the final matrix multiplication, the triple of matrix dimensions must begin and end with the dimensions of the product matrix, m_1 and m_{n+1} respectively).

These properties together guarantee that any piecewise bilinear algorithm to compute $(m_1, \cdots, m_{n+1})$ products can be stepwise described by a subcomputation sequence of $n-1$ triples, and, in turn, any such subcomputation sequence describes a valid sequence of matrix multiplications in the piecewise bilinear computation of $(m_1, \cdots, m_{n+1})$ products. Of course, several such sequences may be computationally equivalent.

Computational duality of piecewise bilinear algorithms. In order to relate computational duality for $(m_1, \cdots, m_{n+1})$ products to that for simple (m_1, m_2, m_3) products, we review the computational duality theorem for matrix multiplication.

LEMMA 1 (Computational duality) [10], [11]. *Given a bilinear algorithm α for computing (m_1, m_2, m_3) products using t multiplication steps and a addition/subtraction steps, and any permutation $\mathcal{D}$ on triples, there is a uniform method of constructing a dual algorithm α' from α such that α' computes $(m_1, m_2, m_3)^{\mathcal{D}} = (u_1, u_2, u_3)$ products in t multiplication steps and $(a + m_1 m_3 - u_1 u_3)$ additions/subtractions.* □

In this paper, we wish to show that, given a piecewise bilinear algorithm α for computing $(m_1, m_2, \cdots, m_{n+1})$ products using t multiplication steps and a additions/subtractions, and any *dihedral* permutation $\mathcal{D}$ on $n+1$-tuples, there is a uniform method of constructing a dual piecewise bilinear algorithm α' from α such that α' computes $(m_1, m_2, \cdots, m_{n+1})^{\mathcal{D}} = (u_1, u_2, \cdots, u_{n+1})$ products in t multiplication steps and $(a + m_1 m_{n+1} - u_1 u_{n+1})$ additive operations. This result is clearly a generalization of the previous result, since every permutation on triples is a dihedral permutation.

The method involves transforming some of the subalgorithms into corresponding dual subalgorithms, then combining, possibly in a different order than in the original algorithm, the resultant new collection of subalgorithms to yield the required dual piecewise bilinear algorithm.

We define operations of reflection and rotation on subcomputation sequences, then show that from a sequence $S = \langle s_1, s_2, \cdots, s_{n-1} \rangle$ for $(m_1, m_2, \cdots, m_{n+1})$ products, given a dihedral permutation $\mathcal{D}$ on $n+1$-tuples, a subcomputation sequence for $(m_1, m_2, \cdots, m_{n+1})^{\mathcal{D}}$ products is obtained by applying to S any sequence of rotations and reflections equivalent to the sequence of rotation and reflection operations on $n+1$-tuples which constitute $\mathcal{D}$.

LEMMA 2. *If $S = \langle s_1, s_2, \cdots, s_{n-1} \rangle$ is a subcomputation sequence for $(m_1, \cdots, m_{n+1})$ products, then $S^D = \langle s_1^D, s_2^D, \cdots, s_{n-1}^D \rangle$ is a subcomputation sequence for $(m_1, \cdots, m_{n+1})^D = (m_{n+1}, m_n, \cdots, m_1)$ products.*

Proof. The proof proceeds by induction on n. The result holds trivially for $n = 2$. Assume the lemma holds for matrix products of up to $n-1$ matrices.

By the properties of subcomputation sequences, $s_{n-1} = (m_1, m_l, m_{n+1})$. By the definition of a subcomputation sequence, there is a subcomputation sequence $S' = \langle s_{i_1}, \cdots, s_{i_{n-l}} \rangle$ of length $n-l$ for $(m_l, m_{l+1}, \cdots, m_{n+1})$ products or a subcomputation sequence $\bar{S}$ of length $l-2$ for computing $(m_1, m_2, \cdots, m_l)$ products or both. By the inductive hypothesis, $(S')^D = \langle s_{i_1}^D, \cdots, s_{i_{n-l}}^D \rangle$ is a subcomputation sequence for $(m_l, m_{l+1}, \cdots, m_{n+1})^D$ products, if $l < n$.

Then, the same argument yields a subcomputation sequence $(\bar{S})^D$ for $(m_1, \cdots, m_l)^D$ products, if $l > 2$.

Finally, either $\langle (\bar{S})^D, (S')^D, s_{n-1}^D \rangle$ or $\langle (S')^D, (\bar{S})^D, s_{n-1}^D \rangle$ (equivalent sequences) are subcomputation sequences for $(m_{n+1}, m_n, \cdots, m_1)$ products. To see this, note that

$$\begin{aligned} s_{n-1}^D &= (m_1, m_l, m_{n+1})^D \\ &= (m_{n+1}, m_l, m_1) \end{aligned}$$

and $(S')^D$, $(\bar{S})^D$ are subcomputation sequences for $(m_{n+1}, \cdots, m_l)$ and $(m_l, \cdots, m_1)$ products, respectively.

Thus, by induction, the lemma holds for all n, as required. □

Before deriving the analogous result for the rotation operation on subcomputation sequences, we note that given any such sequence $S = \langle s_1, s_2, \cdots, s_{n-1} \rangle$ there is an equivalent sequence $T = \langle t_1, t_2, \cdots, t_{n-1} \rangle$ where the last k triples (for some $k \geqq 1$) have the form (m_{j_1}, m_n, m_{n+1}), $(m_{j_2}, m_{j_1}, m_{n+1})$, $\cdots$, $(m_{j_{k-1}}, m_{j_{k-2}}, m_{n+1})$, $(m_1, m_{j_{k-1}}, m_{n+1})$. Thus, without loss of generality, we will now assume all subcomputation sequences are written in this "normal form" before any duality operations are applied.

LEMMA 3. *Given a subcomputation sequence* $S = \langle s_1, s_2, \cdots, s_k, s_{k+1}, \cdots, s_{n-1} \rangle$ *in normal form to compute* $(m_1, m_2, \cdots, m_{n+1})$ *products where k is defined above,*

$$S^R = \langle s_1, s_2, \cdots, s_{n-k-1}, s_{n-1}^R, \cdots, s_{n-k}^R \rangle$$

is a subcomputation sequence for $(m_{n+1}, m_1, \cdots, m_n) = (m_1, m_2, \cdots, m_{n+1})^R$ *products.*

Proof. This follows by the definition of a subcomputation sequence and straightforward substitution. It should be noted that S^R will usually not be in normal form if S is. □

By applying computational duality for matrix multiplication (Lemma 1) to individual triples in the subcomputation sequences S^D and S^R, we have

LEMMA 4. *Given a piecewise bilinear algorithm* α *with subcomputation sequence S which computes* $(m_1, m_2, \cdots, m_{n+1})$ *products using t multiplication steps and a additions/subtractions, it is possible to construct a dual algorithm, with subcomputation sequence* S^D, *which computes* $(m_1, m_2, \cdots, m_{n+1})^D$ *products in t multiplications and a additions/subtractions, and one with subcomputation sequence* S^R, *which computes* $(m_1, m_2, \cdots, m_{n+1})^R$ *products in t multiplications and* $a + m_1 m_{n+1} - m_{n+1} m_n$ *additions/subtractions.*

Proof. Compare the arithmetic costs of new and old corresponding subcomputation triples using Lemma 1. The details of new algorithm construction from algorithm α follow from earlier computational duality [10], [11]; the reader interested in implementing these transformations is referred to [10] for details of bilinear algorithm representations and transformations. □

Since any arbitrary dihedral permutation $\mathcal{D}$ may be written as a sequence of rotations and reflections, given any such $\mathcal{D}$ and a piecewise bilinear algorithm (from which a subcomputation sequence is easily derived) for computing $(m_1, m_2, \cdots, m_{n+1})$ products, we can successively apply Lemma 4 according to any sequence of reflections and rotations equivalent to $\mathcal{D}$ to construct a piecewise bilinear algorithm for $(m_1, m_2, \cdots, m_{n+1})^{\mathcal{D}}$ products. Noting that reflections do not alter product matrix size and successive rotations each increase cost by the amount of decrease of product matrix size, we have the main result.

THEOREM 5. *Given a piecewise bilinear algorithm* α *for computing* $(m_1, m_2, \cdots, m_{n+1})$ *products and any dihedral permutation* $\mathcal{D}$, *there is a uniform method for constructing a piecewise bilinear algorithm* α' *from* α *for computing* $(m_1, m_2, \cdots, m_{n+1})^{\mathcal{D}} = (u_1, u_2, \cdots, u_{n+1})$ *products using exactly the same number of multiplications as* α *and* $m_1 m_{n+1} - u_1 u_{n+1}$ *more additions/subtractions.* □

COROLLARY 6. *If* α *is an optimal piecewise bilinear algorithm for* $(m_1, m_2, \cdots, m_{n+1})$ *products with respect to number of multiplications (additions/subtractions) used,* α' *is an optimal piecewise bilinear algorithm for*

$(m_1, m_2, \cdots, m_{n+1})^{\mathscr{D}}$ *products with respect to number of multiplications (add/subtract operations).* □

Thus, by demonstrating constructively duality among algorithms, we have demonstrated a tight computational complexity relationship among dual problems.

Ramifications of extended computational duality. Corollary 6 is actually a theorem on algorithm duality; problem duality with respect to essential computational complexity follows by assuming that the original algorithm for the original problem is optimal. If that algorithm is nonoptimal (with respect to multiplications, say, since multiplicative complexity of dual problems is identical whereas additive cost of dual problems varies conversely as the product matrix size), then any dual algorithm will be exactly as nonoptimal for the dual problem.

This leads us to consider the class of "approximately optimal" piecewise bilinear algorithms for $(m_1, m_2, \cdots, m_{n+1})$ products [4]. There appears to be a tradeoff between the cost of finding efficient algorithms for this problem of multiplying n matrices and the degree of optimality of the algorithms found. For example, the dynamic program algorithm in [1] finds an optimal algorithm in time n^3; the "min method" [4] (the resultant algorithm computes in both directions beginning with a matrix product triple which begins or ends with a minimum dimension) finds an algorithm at worst half as optimal in linear time and such bad algorithms can be derived by the min method infinitely often. This result is proved in [4] with $(1, 1, x, 1)$ as the input problem assuming pairs of matrices are always multiplied classically, but actually holds for any piecewise bilinear algorithm. Thus, as a corollary to Theorem 5 we have

COROLLARY 7. *Given an approximately optimal algorithm α for computing $(m_1, m_2, \cdots, m_{n+1})$ products and any dihedral permutation $\mathscr{D}$, we can construct an algorithm α' from α for computing $(m_1, m_2, \cdots, m_{n+1})^{\mathscr{D}}$ products, and α' has the same approximate optimality as α.* □

Thus, for the problem of computing $(m_1, m_2, \cdots, m_{n+1})$ products, computational duality is a property of algorithms produced by common optimization techniques such as dynamic programming and the min method. This in turn implies a symmetry of structure for such optimization techniques which may possibly be exploited to gain more understanding of these techniques and simplify proofs and algorithms for them.

The fact that the matrix problem is so simply stated may indicate it is an ideal vehicle for studying design and optimization.

This leads us to note the existence of complexity classes of optimal (or approximately optimal) algorithms for dual computations. Given an $n+1$-tuple $(m_1, m_2, \cdots, m_{n+1})$ of matrix dimensions, if all dimensions are distinct there are $(n+1)!$ problems in the class of problems given by all (symmetric) permutations of $(m_1, m_2, \cdots, m_{n+1})$. A given problem in this class (actually the optimal algorithm for this problem) has the same complexity as all problems given by dihedral permutations of the $n+1$-tuple describing the given problem. There are $2(n+1)$ problems in the "complexity class" of all dihedral permutations of a given problem. Thus, by Theorem 5, there are at most $(n+1)!/[2(n+1)] = n!/2$ equivalence class of problems of equal complexity (or of equal approximate complexity).

Of course, if the matrix dimensions are not all distinct, we may have fewer complexity classes than this (for $n \geqq 3$).

An open question is whether computational duality can be extended to a larger class of problems than those problems represented by all dihedral permutations of a given $(n+1)$-tuple of matrix dimensions. Certainly, the extension cannot be made to the class of problems characterized by all symmetric permutations of a given $(n+1)$-tuple of dimensions. As an example, consider the problems of computing (3, 2, 3, 2) products versus (3, 2, 2, 3) products. By computational duality applied to some known lower bounds on the complexity of 2-matrix multiplication [10], the former has multiplicative complexity 22, the latter has complexity 26. Thus, multiplicative complexity is not invariant over the problems represented by all symmetric permutations of a given $n+1$-tuple of matrix dimension (for $n \geqq 3$, of course).

Before concluding, rather than repeating implementation details for 2-matrix multiplication given in [10], we merely refer the reader to that paper. The straightforward techniques for constructing new bilinear algorithms for dual problems are illustrated in a detailed example. Using Hopcroft and Kerr's algorithm [8] for computing (4, 2, 4) products, algorithms of equal cost are constructed for the dual problems of computing (2, 4, 4) and (4, 4, 2) products. After transformations on bilinear algorithms for matrix multiplication are understood, all that remains given a piecewise bilinear algorithm α for computing $(m_1, \cdots, m_{n+1})$ products, and a dihedral permutation $\mathscr{D}$, is to derive the subcomputation sequence S corresponding to α, factor $\mathscr{D}$ into rotation and reflection operations, using Lemmas 2 and 3 apply an equivalent sequence of rotation and reflection operations to S, apply duality transformations to the bilinear subalgorithms of α where required, and reorder the new collection of bilinear subalgorithms according to $S^{\mathscr{D}}$. This yields a piecewise bilinear algorithm for computing $(m_1, m_2, \cdots, m_{n+1})^{\mathscr{D}}$ products by Theorem 5.

Thus, we have demonstrated the existence of a large class of computationally dual problems of dually complex algorithmic structure, and therefore, of course, of dual arithmetic complexity.

Pairwise multi-matrix multiplication is one example of a particular kind of computationally dual problem mentioned in [11], namely one in which the essential kind of problem is invariant over dihedral duality transformations. Of course, new lower bounds for specific algebraic problems must be derived before lower bounds can be given by our approach to classes of problems.

Acknowledgments. The author is grateful to Professor Charles Fiduccia for many stimulating debates on the nature of computational duality, and his encouragement to formalize the results presented in this paper. The author also gratefully acknowledges the contribution of the referee's comments to the improvement of the presentation of this paper. In particular, the referee suggested the normal form of subcomputation sequences, which simplified significantly the proof of Lemmas 3 and 4.

REFERENCES

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

[2] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[3] R. BROCKETT AND D. DOBKIN, *On the optimal evaluation of a set of bilinear forms*, Proc. Fifth Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, 1973, pp. 88–95.

[4] A. K. CHANDRA, *Computing matrix chain products in near-optimal time*, IBM Res. Rep. RC 5625, Yorktown Heights, New York, 1975.

[5] J. DE PILLIS, *An extended duality theorem and multiplication of several matrices*, J. Linear and Multilinear Alg., to appear.

[6] C. M. FIDUCCIA, *On obtaining upper bounds on the complexity of matrix multiplication*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 31–40.

[7] ———, *On the algebraic complexity of matrix multiplication*, Doctoral Thesis, Brown Univ., Providence, RI, 1973.

[8] J. E. HOPCROFT AND L. KERR, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–36.

[9] J. E. HOPCROFT AND J. MUSINSKI, *Duality applied to the complexity of matrix multiplication and other bilinear forms*, this Journal, 2 (1973), pp. 159–173.

[10] R. L. PROBERT, *On the complexity of symmetric computations*, Canad. J. Information Processing and Operational Res., 12 (1974), pp. 71–86.

[11] ———, *On the additive complexity of matrix multiplication*, this Journal, 5 (1976), pp. 187–203.

[12] ———, *Commutativity, non-commutativity, and bilinearity*, Information Processing Lett., 5 (1976), pp. 46–49.

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

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

[15] D. DOBKIN, *On the arithmetic complexity of a class of arithmetic computations*, Doctoral Thesis, Harvard Univ., Cambridge, MA, 1973.

SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

COMPUTATIONAL PARALLELS BETWEEN THE REGULAR AND CONTEXT-FREE LANGUAGES*

H. B. HUNT, III† AND D. J. ROSENKRANTZ‡

Abstract. Several sufficient conditions are presented for a regular set or context-free language problem to be as hard as testing for emptiness or testing for equivalence to the language $\{0, 1\}^*$. These sufficient conditions provide a unified method for proving undecidability or complexity results and apply to a large number of language problems studied in the literature. Many new nonpolynomial lower complexity bounds and undecidability results follow easily.

The techniques used to prove these sufficient conditions involve reducibilities utilizing simple and efficient encodings by homomorphisms.

Key words. computational complexity, efficient reducibility, decidability, lower bounds, regular sets, regular expressions, context-free languages, and context-free grammars

1. Introduction. By a language predicate, we mean any function from a set of languages into the set {True, False}. Here, we consider the complexity or decidability of language predicates in general. For a given class of language descriptors (such as the context-free grammars or the regular expressions) and a given language predicate on the corresponding set of languages, we consider the complexity or decidability of testing a language descriptor to determine if the language it denotes satisfies the predicate. Two language predicates, whose complexities have been studied for many different kinds of language descriptors, are testing for emptiness and testing for equivalence to the language $\{0, 1\}^*$, which we denote by "$=\varnothing$" and "$=\{0, 1\}^*$", respectively.

We give several sufficient conditions for a regular set or context-free language predicate to be as hard as "$=\varnothing$" or "$=\{0, 1\}^*$". An example of such sufficient conditions is—

1) the predicate is true for $\{0, 1\}^*$; and
2) there exists a context-free language L' that is *not* expressible as $x \backslash L$, where x is a string and L is a language for which the predicate is true.

Any context-free language predicate satisfying these two conditions is as hard as "$=\{0, 1\}^*$", and thus is undecidable when the context-free languages are denoted by context-free grammars or pushdown automata. If the word "regular" is substituted for the word "context-free" in the statement of the two conditions above, then any predicate that satisfies the resulting two conditions is as hard as "$=\{0, 1\}^*$" for the regular sets, and thus is *PSPACE*-hard, when the regular sets are denoted by regular expressions, regular grammars, or nondeterministic finite automata.

The techniques used to prove the sufficiency of the conditions presented here involve showing that there is an efficient (*not* just effective) reduction of the "$=\varnothing$" or "$=\{0, 1\}^*$" problem to any language predicate satisfying them. These reductions utilize simple encoding arguments involving homomorphisms. The encodings are the same for several different language classes including the context-free languages and the regular sets. Thus, for a variety of different language classes $\mathscr{C}$, there is a uniform way of embedding the "$=\varnothing$" or "$=\{0, 1\}^*$" problem for $\mathscr{C}$ into each language predicate on $\mathscr{C}$ satisfying the given sufficient conditions.

* Received by the editors March 11, 1976, and in revised form April 25, 1977.

† Aiken Computation Laboratory, Harvard University, Cambridge, Massachussetts 02138. The research of this author was supported in part by a National Science Foundation graduate fellowship in computer science, by the National Science Foundation under Grant GJ-35570, and by the General Electric Company Corporate Research and Development Center.

‡ General Electric Company, Schenectady, New York, 12345.

In § 2 the undecidability of context-free language predicates is studied in detail. Several general undecidability theorems for context-free language predicates are presented. These theorems extend results in [9], [11], and [13].

In § 3, the complexity of regular set problems is studied. Many regular set predicates studied in the literature are shown to be as hard as the regular expression equivalence problem, the complexity of which is extensively studied in [19] and [28].

Section 4 is a short conclusion.

Several preliminary definitions are needed to read this paper. See [14] for the definitions of a regular set, regular grammar, deterministic finite automaton, nondeterministic finite automaton, pushdown automaton, context-free language (cfl), context-free grammar (cfg), linear bounded automaton, context-sensitive language (csl), context-sensitive grammar (csg), and Turing machine (Tm). A two-way deterministic finite automaton (2dfa) is defined as in [14], except that the input string is enclosed by endmarkers. We abbreviate infinitely often by i.o. All logarithms are to the base 2.

We use λ to denote the empty string and $\varnothing$ to denote the empty set.

DEFINITION 1.1. The set of $(\cup, \cdot, *)$ *regular expressions over* $\{0, 1\}$ is defined recursively as follows:

(a) λ, $\varnothing$, 0 and 1 are $(\cup, \cdot, *)$ regular expressions.

(b) If A and B are $(\cup, \cdot, *)$ regular expressions, then so are

$$(A)\cup(B), \quad (A)\cdot(B), \quad \text{and} \quad (A)^*.$$

(c) Nothing else is a $(\cup, \cdot, *)$ regular expression.

Thus $(\cup, \cdot, *)$ regular expressions are strings over $\{(, \cup, \cdot, *,), \lambda, \varnothing, 0, 1\}$. The sets of $(\cup, \cdot, *, \cap)$, $(\cup, \cdot, *, \sim)$, $(\cup, \cdot, *, \oplus)$, *and* $(\cup, \cdot, *, ^2)$ *regular expressions over* $\{0, 1\}$ are defined analogously. Here $\cap$, $\sim$, $\oplus$, and ^{2} denote intersection, complementation with respect to $\{0, 1\}^*$, exclusive or, and squaring respectively. Thus if $A, B \subseteq \{0, 1\}^*$, then $A \oplus B = \{x \mid x \in A \cap \sim B \text{ or } x \in \sim A \cap B\}$ and $A^2 = \{z \mid$ there exist $x, y \in A$ for which $z = x \cdot y\}$.

The language denoted by a regular expression R is written $L(R)$.

DEFINITION 1.2. The *star height* SH *of a* $(\cup, \cdot, *)$ *regular expression* is defined recursively:

$$\mathrm{SH}(0) = 0, \quad \mathrm{SH}(1) = 0, \quad \mathrm{SH}(\lambda) = 0, \quad \mathrm{SH}(\varnothing) = 0$$

$$\mathrm{SH}((A)\cup(B)) = \max\{\mathrm{SH}(A), \mathrm{SH}(B)\},$$

$$\mathrm{SH}((A)\cdot(B)) = \max\{\mathrm{SH}(A), \mathrm{SH}(B)\},$$

$$\mathrm{SH}((A)^*) = \mathrm{SH}(A) + 1.$$

The *star height of a regular set* L is the minimum of the star heights of any $(\cup, \cdot, *)$ regular expression R such that $L(R) = L$.

DEFINITION 1.3. For cfgs we use $\overset{*}{\Rightarrow}$ to mean derives by a sequence of zero or more steps. Let $G = (N, \Sigma, P, S)$ be a cfg. Then $L(G)$, the language generated by G, equals $\{w \mid w \in \Sigma^* \text{ and } S \overset{*}{\Rightarrow} w\}$. G is said to be *linear* if the right-hand side of each production of G is an element of $\Sigma^* \cup \Sigma^* N \Sigma^*$. G is said to be *nonselfembedding* if for all B in N, $B \overset{*}{\Rightarrow} xBy$ implies x equals λ or y equals λ.

DEFINITION 1.4. A cfg G is said to be *ambiguous* if some string x in $L(G)$ has two distinct leftmost or, equivalently, two distinct rightmost derivations. G is said to be *inherently ambiguous* if all cfgs generating $L(G)$ are ambiguous.[1]

[1] A detailed discussion of derivations and ambiguity can be found in [1].

DEFINITION 1.5. Let k be a positive integer. A cfg G is said to be *ambiguous of degree k* if each string in $L(G)$ has at most k distinct derivations. G is said to be *inherently ambiguous* of degree k if $L(G)$ cannot be generated by any grammar that is ambiguous of degree less than k, but some grammar generating $L(G)$ is ambiguous of degree k.

G is said to be *infinitely ambiguous* if for each positive integer i, there exists a string in $L(G)$ that has at least i distinct leftmost derivations. G is said to be *infinitely inherently ambiguous* if each grammar generating $L(G)$ is infinitely ambiguous.

It is known that for all $k \geqq 2$ there exist inherently ambiguous cfgs of degree k. Similarly it is known that there exist infinitely inherently ambiguous cfgs [24].

In what follows Σ denotes an arbitrary finite nonempty alphabet.

DEFINITION 1.6. Let $A, B \subseteq \Sigma^*$.

$$A \backslash B = \{y \mid \text{there exists } x \in A \text{ for which } x \cdot y \in B\}.$$

$$A / B = \{x \mid \text{there exists } y \in B \text{ for which } x \cdot y \in A\}.$$

$A \backslash B$ is called the *left quotient of B with respect to A*. A/B is called the *right quotient of A with respect to B*.

DEFINITION 1.7. Let $L \subseteq \Sigma^*$. Then L^{rev}, the *reversal of L*, equals $\{a_n \cdot \cdots \cdot a_1 \mid \text{each } a_i \in \Sigma, \text{ and } a_1 \cdot \cdots \cdot a_n \in L\}$.

DEFINITION 1.8. Let $L \subseteq \Sigma^*$. Then

$$\text{Init}(L) = \{x \mid x \in \Sigma^* \text{ and there exists } y \in \Sigma^* \text{ for which } x \cdot y \in L\}.$$

$$\text{Fin}(L) = \{y \mid y \in \Sigma^* \text{ and there exists } x \in \Sigma^* \text{ for which } x \cdot y \in L\}.$$

DEFINITION 1.9. A language $L \subseteq \Sigma^*$ is said to be *bounded* if and only if there exist strings $w_1, \cdots, w_k$ in Σ^* such that $L \subseteq w_1^* \cdots w_k^*$. A language that is not bounded is said to be *unbounded*.

The following properties of unbounded regular sets are used repeatedly in this paper.

LEMMA 1.10. *Let L be any regular set over* $\{0, 1\}$. *Then the following are equivalent*:

1) *L is unbounded*;

2) *there exist strings r, s, x, and y in* $\{0, 1\}^*$ *such that*

$$r\{0x, 1y\}^* s \subseteq L;$$

and

3) *there exist strings r, s, x, and y in* $\{0, 1\}^*$ *such that*

$$r\{x0, y1\}^* s \subseteq L.$$

The proof of the equivalence of 1) and 2) can be found in [13]. The equivalence of 2) and 3) follows easily from the fact that the class of unbounded languages is closed under reversal.

The next definition defines the concept of a predicate and introduces, for such a predicate $\mathcal{P}$, four closely related sets of languages $\mathcal{P}_{\text{left}}$, $\mathcal{P}_{\text{right}}$, $\mathcal{L}(\mathcal{P}, r, s, x, y)$ and $\mathcal{R}(\mathcal{P}, r, s, x, y)$. These sets can be viewed as sets of languages that are generated from the set of languages for which $\mathcal{P}$ is true by several simple linguistic operations. Their importance will become apparent in §§ 2 and 3.

DEFINITION 1.11. A *predicate* $\mathcal{P}$ is a function from a set L into {True, False}. $\mathcal{P}$ is said to be *nontrivial* if there exist a, b in L such that $\mathcal{P}(a) = \text{True}$ and $\mathcal{P}(b) = \text{False}$.

Let $\mathcal{P}$ be nontrivial predicate on a class F of languages over $\{0, 1\}$. Then

(a) $\mathcal{P}_{\text{left}} = \{L' | L' = x\backslash L,\ x \in \{0, 1\}^+,\ \mathcal{P}(L) \text{ is true}\}$ and

(b) $\mathcal{P}_{\text{right}} = \{L' | L' = L/x,\ x \in \{0, 1\}^+,\ \mathcal{P}(L) \text{ is true}\}$.

Let r, s, x, and y be strings over $\{0, 1\}$. Then

(c) $\mathcal{L}(\mathcal{P}, r, s, x, y) = \{L' | L' = h^{-1}[\alpha\backslash[r\backslash L/s]],\ \alpha \in \{0x, 1y\}^+,\ \mathcal{P}(L) \text{ is true}\}$, where $h:\{0, 1\}^* \to \{0, 1\}^*$ is the one to one homomorphism defined by $h(0) = 0x$ and $h(1) = 1y$; and

(d) $\mathcal{R}(\mathcal{P}, r, s, x, y) = \{L' | L' = h^{-1}[[r\backslash L/s]/\alpha],\ \alpha \in \{x0, y1\}^+,\ \mathcal{P}(L) \text{ is true}\}$, where $h:\{0, 1\}^* \to \{0, 1\}^*$ is the one to one homomorphism defined by $h(0) = x0$ and $h(1) = y1$.

The reader should note that $\mathcal{P}_{\text{left}}$, $\mathcal{P}_{\text{right}}$, $\mathcal{L}(\mathcal{P}, r, s, x, y)$ and $\mathcal{R}(\mathcal{P}, r, s, x, y)$ are classes of languages over $\{0, 1\}$. Note also that $\mathcal{L}(\mathcal{P}, \lambda, \lambda, \lambda, \lambda) = \mathcal{P}_{\text{left}}$ and $\mathcal{R}(\mathcal{P}, \lambda, \lambda, \lambda, \lambda) = \mathcal{P}_{\text{right}}$. As an illustration of Definition 1.11(c), consider the predicate $\mathcal{P}$ that is true only for the language

$$L = 011\cdot\{010, 1111, 00\}^*\cdot 10.$$

Let $r = 011$, $s = 10$, $x = 10$, and $y = 1$. Then

$$r\backslash L/s = \{010, 1111, 00\}^*$$

and

$$\{L' | L' = \alpha\backslash[r\backslash L/s],\ \alpha \in \{0x, 1y\}^+\} = \{\{010, 1111, 00\}^*, 11\cdot\{010, 1111, 00\}^*\}.$$

Finally.

$$\mathcal{L}(\mathcal{P}, r, s, x, y) = \{\{0, 11\}^*, 1\cdot\{0, 11\}^*\}.$$

DEFINITION 1.12. 1) P (NP) is the class of all languages over $\{0, 1\}$ accepted by some deterministic (nondeterministic) polynomially time-bounded Tm.

2) *PSPACE* is the class of all languages over $\{0, 1\}$ accepted by some polynomially space-bounded Tm.

3) *DCSL* (*NDCSL*) is the class of all languages over $\{0, 1\}$ accepted by some deterministic (nondeterministic) linear bounded automaton.

DEFINITION 1.13. Let Σ, Δ be finite nonempty alphabets. Let $L \subseteq \Sigma^*$ and $M \subseteq \Delta^*$.

1) We say that L is *p-reducible* to M if there exists a function $f:\Sigma^* \to \Delta^*$ computable by a deterministic polynomially time-bounded Tm such that for all x in Σ^*, x is in L if and only if $f(x)$ is in M. L is said to be *NP-(PSPACE-)hard* if all languages in *NP* (*PSPACE*) are p-reducible to it. L is said to be *NP* (*PSPACE-*)*complete* if it is *NP* (*PSPACE-*)*hard* and is accepted by some nondeterministic polynomially time-bounded (polynomially space-bounded) Tm.

2) A *log-space transducer* T is a deterministic Tm with a two-way read-only input tape, a one-way output tape, and several two-way read-write work tapes such that T given input x always halts with some string y on its output tape, and such that T never uses more than $O(\log |x|)$ tape cells on its work tapes. A function f: $\Sigma^* \to \Delta^*$ is said to be *log-computable* if there exists a log-space transducer T such that T, when given input x in Σ^*, eventually halts with output $f(x)$. If in addition $|f(x)| = O(|x|)$, f is said to be *log-lin computable*. We say that L is *log-reducible* to M if there exists a log-computable function f: $\Sigma^* \to \Delta^*$ such that for all x in Σ^*, x is in L if and only if $f(x)$ is in M.

Let $F(n)$ be the function

$$F(n) = 2^{2^{\cdot^{\cdot^{\cdot^2}}}}\Big\}\, n \text{ levels of exponentiation,}$$

i.e. $F(0) = 1$ and $F(n) = 2^{F(n-1)}$ for $n > 0$.

THEOREM 1.14. 1) *$\{R|R$ is a $(\cup, \cdot, *)$ regular expression over $\{0, 1\}$ and $L(R) \neq \{0, 1\}^*\}$ is PSPACE-complete; requires space greater than n^r i.o., for all r less than 1, on any nondeterministic Tm; and is an element of DCSL if and only if DCSL = NDCSL.*

2) *$\{M|M$ is a 2dfa and $L(M) \neq \{0, 1\}^*\}$ is PSPACE-complete; and requires space greater than n^r i.o., for all r less than 1, on any nondeterministic Tm.*

3) *$\{R|R$ is a $(\cup, \cdot, *, \cap)$ regular expression over $\{0, 1\}$ and $L(R) \neq \{0, 1\}^*\}$ requires space greater than $2^{cn/\log n}$ i.o., for some c greater than 0, on any Tm.*

4) *$\{R|R$ is a $(\cup, \cdot, *, ^2)$ regular expression over $\{0, 1\}$ and $L(R) \neq \{0, 1\}^*\}$ requires space greater than 2^{cn} i.o., for some c greater than 0, on any Tm.*

5) *$\{G|G$ is a nonselfembedding cfg with terminal alphabet $\{0, 1\}$ and $L(G) \neq \{0, 1\}^*\}$ requires space greater than $2^{cn/(\log n)}$ i.o., for some c greater than 0, on any Tm.*

6) *$\{R|R$ is a $(\cup, \cdot, *, \sim)$ regular expression over $\{0, 1\}$ and $L(R) \neq \{0, 1\}^*\}$ requires space greater than $F(c \cdot \log n)$ i.o., for some c greater than 0, on any Tm.*

7) *$\{R|R$ is a $(\cup, \cdot, *, \oplus)$ regular expression over $\{0, 1\}$ and $L(R) \neq \{0, 1\}^*\}$ requires space greater than $F(c \cdot \log n)$ i.o., for some c greater than 0, on any Tm.*

The proofs of 1), 4), and 6) can be found in [28]. The proof of 2) can be found in [15], and the proof of 5) can be found in [19]. A weaker form of 3) appears in [15]; the lower bound in 3) was suggested by L. J. Stockmeyer, and appears in [27]. Conclusion 7) follows easily from conclusion 6) since $L(\sim A) = \{0, 1\}^* \oplus L(A)$ for all $(\cup, \cdot, *, \sim)$ regular expressions A over $\{0, 1\}$.

THEOREM 1.15. 1) *$\{M|M$ is a 2dfa and $L(M) \neq \emptyset\}$ is PSPACE-complete; and requires space greater than n^r i.o., for all r less than 1, on any nondeterministic Tm.*

2) *$\{R|R$ is a $(\cup, \cdot, *, \cap)$ regular expression over $\{0, 1\}$ and $L(R) \neq \emptyset\}$ is PSPACE-hard.*

3) *$\{R|R$ is a $(\cup, \cdot, *, \sim)$ regular expression over $\{0, 1\}$ and $L(R) \neq \emptyset\}$ requires space greater than $F(c \cdot \log n)$ i.o., for some c greater than 0, on any Tm.*

4) *$\{R|R$ is a $(\cup, \cdot, *, \oplus)$ regular expression over $\{0, 1\}$ and $L(R) \neq \emptyset\}$ requires space greater than $F(c \cdot \log n)$ i.o., for some c greater than 0, on any Tm.*

The proofs of 1) and 2) can be found in [15]. The proof of 3) can be found in [28]. Again the lower bound in 4) follows easily from that in 3).

2. Undecidable properties of context-free languages. New general undecidability theorems for predicates on the cfls are presented. Theorem 2.4 shows that any predicate that sufficiently dichotomizes the cfls is undecidable. The conditions for sufficient dichotomy are that the predicate is true for some language with an unbounded regular subset, and that the predicate is false for some language that cannot be obtained from the set of languages for which the predicate is true by several linguistic operations. Our results extend related work in [9], [11], and [13]. Analogous results hold for many subfamilies of the cfls including the linear cfls, the metalinear cfls, the on-line one counter languages, and the least AFL containing $\{a^n b^n | n \geqq 1\}$.

The first theorem is a special but important corollary of Theorem 2.4 below. We present an independent proof of Theorem 2.1 in order to display the ideas behind the proof of Theorem 2.4 in a simple setting.

THEOREM 2.1. *Let $\mathcal{P}$ be any predicate on the cfls over $\{0, 1\}$ such that $\mathcal{P}(\{0, 1\}^*)$ is true and such that $\mathcal{P}_{\text{left}}$ or $\mathcal{P}_{\text{right}}$ is a proper subset of the cfls over $\{0, 1\}$. Then for arbitrary cfg G, the predicate $\mathcal{P}(L(G))$ is undecidable. Similarly for arbitrary pushdown automaton M, the predicate $\mathcal{P}(L(M))$ is undecidable.*

Proof. We prove Theorem 2.1 for cfgs; the proof for pushdown automata is similar. The proof consists of effectively reducing the known undecidable predicate "$L(G) = \{0, 1\}^*$" to the predicate $\mathcal{P}(L(G))$, where $\mathcal{P}$ is any predicate on the cfls over $\{0, 1\}$ satisfying the conditions of the theorem. Let $\mathcal{P}$ be any such predicate. Suppose $\mathcal{P}_{\text{left}}$ is a

proper subset of the cfls over $\{0, 1\}$. Let L_f be a cfl over $\{0, 1\}$ that is not in $\mathscr{P}_{\text{left}}$. Let $h: \{0, 1\}^* \to \{0, 1\}^*$ be the homomorphism defined by $h(0) = 00$ and $h(1) = 01$. (The reader should note that this homomorphism is one to one.) For any cfg G, a cfg H can be constructed effectively such that

$$L(H) = L_1 \cup L_2 \cup L_3$$

and

$$L_1 = h(L(G))\cdot 10\{0, 1\}^*$$

$$L_2 = \{00, 01\}^*\cdot 10 \cdot L_f$$

$$L_3 = \sim[\{00, 01\}^*\cdot 10\cdot\{0, 1\}^*].$$

We claim that $\mathscr{P}(L(H))$ is true if and only if $L(G) = \{0, 1\}^*$.

There are two cases to consider.

Case 1. If $L(G) = \{0, 1\}^*$, then $h(L(G)) = \{00, 01\}^*$ and

$$L_1 = \{00, 01\}^*\cdot 10\cdot\{0, 1\}^*.$$

Thus $L_1 \cup L_3 = \{0, 1\}^*$, so that $L(H) = \{0, 1\}^*$ and $\mathscr{P}(L(H))$ is true.

Case 2. If $L(G)$ is properly contained in $\{0, 1\}^*$, then there exists a string w in $\{0, 1\}^* - L(G)$. This implies that $h(w)$ is in $\{00, 01\}^* - h(L(G))$, and that $h(w)\cdot 10$ is not a prefix of any string in L_1. Also, $h(w)\cdot 10$ is not a prefix of any string in L_3. Thus $h(w)\cdot 10 \backslash L(H) = L_f$. If $\mathscr{P}(L(H))$ were true, L_f would be in $\mathscr{P}_{\text{left}}$. But L_f was selected to be a cfl over $\{0, 1\}$ not in $\mathscr{P}_{\text{left}}$. Therefore $\mathscr{P}(L(H))$ is false.

Thus $\mathscr{P}(L(H))$ is true if and only if $L(G) = \{0, 1\}^*$. Since H is constructed effectively from G, the predicate $\mathscr{P}(L(H))$ is undecidable. The proof when $\mathscr{P}_{\text{right}}$ is a proper subset of the cfls over $\{0, 1\}$ is analogous. □

DEFINITION 2.2 A predicate $\mathscr{P}$ is *preserved by quotient with singletons on the left* if $\mathscr{P}(L)$ true implies $\mathscr{P}(x\backslash L)$ is true for all x in $\{0, 1\}^+$. Similarly $\mathscr{P}$ is *preserved by quotient with singletons on the right* if $\mathscr{P}(L)$ true implies $\mathscr{P}(L/x)$ is true for all x in $\{0, 1\}^+$.

Unlike the results in [9], Theorem 2.1 does not require $\mathscr{P}$ to be preserved by quotient with singletons on either the left or the right. For example the following holds.

PROPOSITION 2.3. *Let $\mathscr{P}$ be the predicate "$L = [L]^{\text{rev}}$". Then $\mathscr{P}$ is not preserved by quotient with singletons on the left or on the right; but $\mathscr{P}$ satisfies the conditions of Theorem* 2.1. *Thus for arbitrary cfg G the predicate "$L(G) = [L(G)]^{\text{rev}}$" is undecidable.*

Proof. Let $L = \{001, 100\}$. Then $L = [L]^{\text{rev}}$; but $0\backslash L = \{0, 1\}$ and $L/0 = \{10\}$. Thus $\mathscr{P}$ is not preserved by quotient with singletons on the left or on the right. Since $\{0, 1\}^* = [\{0, 1\}^*]^{\text{rev}}$, $\mathscr{P}(\{0, 1\}^*)$ is true. Moreover as shown below, $\mathscr{P}_{\text{left}}$ is a proper subset of the cfls over $\{0, 1\}$. Thus $\mathscr{P}$ satisfies the conditions of Theorem 2.1; and for arbitrary cfg G the predicate "$L(G) = [L(G)]^{\text{rev}}$" is undecidable.

Facts 1 and 2 below show that the cfl $1\cdot\{0, 1\}^*$ is not in $\mathscr{P}_{\text{left}}$, so that $\mathscr{P}_{\text{left}}$ is a proper subset of the cfls over $\{0, 1\}$.

FACT 1. *If L is in $\mathscr{P}_{\text{left}}$, then there exists an x in $\{0, 1\}^+$ such that $L/x = [L/x]^{\text{rev}}$.*

FACT 2. *There is no string x in $\{0, 1\}^+$ such that*

$$1\cdot\{0, 1\}^*/x = [1\{0, 1\}^*/x]^{\text{rev}}.$$

To prove Fact 1, observe that if L is in $\mathscr{P}_{\text{left}}$, there is a y in $\{0, 1\}^+$ and L_0 such that $L = y\backslash L_0$ and $L_0 = [L_0]^{\text{rev}}$. But $(y\backslash L_0)/y^{\text{rev}}$ equals $[(y\backslash L_0)/y^{\text{rev}}]^{\text{rev}}$. Therefore letting $x = y^{\text{rev}}$, $L/x = [L/x]^{\text{rev}}$. To prove Fact 2, observe that for a string x in $\{0, 1\}^+$ beginning with 1

$$1\cdot\{0, 1\}^*/x = \{\lambda\} \cup 1\cdot\{0, 1\}^*$$

and for a string x in $\{0, 1\}^+$ beginning with 0

$$1\cdot\{0, 1\}^*/x = 1\cdot\{0, 1\}^*.$$

Thus for all x in $\{0, 1\}^+$, 10 is in $1\cdot\{0, 1\}^*/x$, but 01 is not. □

Next Theorem 2.1 is extended so that $\mathcal{P}$ need only be true for some cfl L_t that contains an unbounded regular subset. The properties of unbounded regular sets that we exploit are those of Lemma1.10. Thus a cfl L_t over $\{0, 1\}$ has an unbounded regular subset if and only if there exists strings r, s, x, and y in $\{0, 1\}^*$ such that

$$r\cdot\{0x, 1y\}^*\cdot s \subseteq L_t \quad \text{or} \quad r\cdot\{x0, y1\}^*\cdot s \subseteq L_t.$$

THEOREM 2.4. *Let $\mathcal{P}$ be any predicate on the cfls over $\{0, 1\}$ for which there exists a cfl L_t and strings r, s, x, and y in $\{0, 1\}^*$ such that*

(a) *$\mathcal{P}(L_t)$ is true,*

(b) *$r\cdot\{0x, 1y\}^*\cdot s \subseteq L_t$, and*

(c) *$\mathcal{L}(\mathcal{P}, r, s, x, y)$ is a proper subset of the cfls over $\{0, 1\}$;*

or

(d) *$\mathcal{P}(L_t)$ is true,*

(e) *$r\cdot\{x0, y1\}^*\cdot s \subseteq L_t$, and*

(f) *$\mathcal{R}(\mathcal{P}, r, s, x, y)$ is a proper subset of the cfls over $\{0, 1\}$.*

Then for arbitrary cfg G the predicate $\mathcal{P}(L(G))$ is undecidable. Similarly for arbitrary pushdown automaton M, the predicate $\mathcal{P}(L(M))$ is undecidable.

Proof. We only prove the theorem for the cfgs. Let $\mathcal{P}$ be any predicate on the cfls over $\{0, 1\}$ satisfying (a), (b), and (c). From (c), there is a cfl L_f over $\{0, 1\}$ that is not in $\mathcal{L}(\mathcal{P}, r, s, x, y)$. Let h_1, h_3: $\{0, 1\}^* \to \{0, 1\}^*$ be the one to one homomorphisms defined by $h_1(0) = 0x$, $h_1(1) = 1y$, $h_3(0) = 0x0x$ and $h_3(1) = 0x1y$. For any cfg G a cfg H can be constructed effectively such that

$$L(H) = L_1 \cup L_2 \cup L_3$$

where

$$L_1 = r\cdot h_3(L(G))\cdot 1y0x\cdot\{0x, 1y\}^*\cdot s$$

$$L_2 = r\cdot\{0x0x, 0x1y\}^*\cdot 1y0x\cdot h_1(L_f)\cdot s$$

$$L_3 = L_t \cap \sim[r\cdot\{0x0x, 0x1y\}^*\cdot 1y0x\cdot\{0x, 1y\}^*\cdot s].$$

But $\mathcal{P}(L(H))$ is true if and only if $L(G) = \{0, 1\}^*$. There are two cases to consider.

Case 1. If $L(G) = \{0, 1\}^*$, then $h_3(L(G)) = \{0x0x, 0x1y\}^*$ and $L_1 \cup L_3 = L_t$. Thus $L(H) = L_t$ and by (a) $\mathcal{P}(L(H))$ is true.

Case 2. If $L(G)$ is properly contained in $\{0, 1\}^*$, then there exists a string w in $\{0, 1\}^* - L(G)$. Thus $h_3(w)$ is in $\{0x0x, 0x1y\}^* - h_3(L(G))$. Let $\alpha = h_3(w)\cdot 1y0x$. Then $r\cdot\alpha$ is not a prefix of any string in L_1. Let

$$A = \{z \mid r\cdot\alpha\cdot z\cdot s \in L_3\}$$

Note that A contains no string in $\{0x, 1y\}^*$.

Now, $\alpha\backslash[r\backslash L(H)/s] = h_1(L_f) \cup A$. Also

$$h_1^{-1}(\alpha\backslash[r\backslash L(H)/s]) = h_1^{-1}h_1(L_f) \cup h_1^{-1}(A).$$

Since h_1 is one-to-one, $h_1^{-1}\, h_1(L_f) = L_f$. Moreover $h_1^{-1}(A) = \varnothing$. This follows since t in $h_1^{-1}(A)$ implies $h_1(t)$, a string in $\{0x, 1y\}^*$, is in A, a contradiction. Thus $h_1^{-1}(\alpha\backslash[r\backslash L(H)/s]) = L_f$. If $\mathcal{P}(L(H))$ were true, L_f would be in $\mathcal{L}(\mathcal{P}, r, s, x, y)$, which is a contradiction. There, $\mathcal{P}(L(H))$ is false.

The remainder of the proof closely follows the above and is left to the reader. □

The reader should note the following facts about Theorem 2.4. First, (a) and (b), and (d) and (e) assert that $\mathcal{P}$ is true for some cfl L_t with an unbounded regular subset. However, L_t need not be regular. Second, there is only one homomorphism involved in the statement of (c) or (f). Third, the homomorphisms in (c) and (f), depend only upon the particular decomposition of L_t used to determine r, s, x, and y in (b) and (e), respectively. Thus if $r\{0, 1\}^* s \subseteq L_t$ for some strings r and s, then the homomorphisms and inverse homomorphisms in Definition 1.11 can be set equal to the identity on $\{0, 1\}^*$. Fourth, $\mathcal{P}$ need not be preserved under quotient with singletons on the left or on the right, or under inverse homomorphisms. All that is required is that these operations, when applied as in (c) or (f) to the set of cfls over $\{0, 1\}$ for which $\mathcal{P}$ is true, do not generate all the cfls over $\{0, 1\}$. Thus let $\mathcal{P}$ be some nontrivial predicate on the cfls over $\{0, 1\}$; and let $\text{True}(\mathcal{P}) = \{L | \mathcal{P}(L) \text{ is true}\}$. If $\text{True}(\mathcal{P})$ is closed under quotient with singletons on both the left and the right and under inverse one-to-one homomorphisms, then any predicate $\mathcal{P}'$ such that $\text{True}(\mathcal{P}') \subseteq \text{True}(\mathcal{P})$ and such that $\mathcal{P}'$ is true for some cfl with an unbounded regular subset is undecidable. This is true regardless of the closure properties of $\text{True}(\mathcal{P}')$. We illustrate these observations with several corollaries of Theorem 2.4.

PROPOSITION 2.5. *Any predicate satisfying the conditions of Theorem* 2.1 *satisfies the conditions of Theorem* 2.4.

Proof. Let $\mathcal{P}$ be any predicate on the cfls over $\{0, 1\}$ satisfying the conditions of Theorem 2.1. Then $\mathcal{P}(\{0, 1\}^*)$ is true and $\mathcal{P}_{\text{left}}$ or $\mathcal{P}_{\text{right}}$ is a proper subset of the cfls over $\{0, 1\}$. L_t of (a) and (d) of Theorem 2.4 can be set equal to $\{0, 1\}^*$; and r, s, x, and y of (b) or (e) of Theorem 2.4 can be set equal to λ, the empty string. Thus the homomorphisms of Definition 1.11 are equal to the identity function on $\{0, 1\}^*$. Thus $\mathcal{L}(\mathcal{P}, r, s, x, y) = \mathcal{L}(\mathcal{P}, \lambda, \lambda, \lambda, \lambda) = \{L' | L' = \alpha \backslash L, \quad \alpha \in \{0, 1\}^+, \quad \mathcal{P}(L) \text{ is true}\} = \mathcal{P}_{\text{left}}$. Similarly $\mathcal{R}(\mathcal{P}, r, s, x, y) = \mathcal{R}(\mathcal{P}, \lambda, \lambda, \lambda, \lambda) = \{L' | L' = L/\alpha, \alpha \in \{0, 1\}^+, \mathcal{P}(L) \text{ is true}\} = \mathcal{P}_{\text{right}}$. □

Thus Theorem 2.4 is an extension of Theorem 2.1. Our next result shows that it is a proper extension.

COROLLARY 2.6. *Let L_0 be any cfl over $\{0, 1\}$ such that L_0 contains an unbounded regular subset. Then for arbitrary cfg G the predicate $\mathcal{P}$ equal to "$L(G) = L_0$" is undecidable.*

Proof. Since L_0 has an unbounded regular subset, there exist strings r, s, x, and y in $\{0, 1\}^*$ such that $r\{0x, 1y\}^* s \subseteq L_0$. Therefore $[r \backslash L_0 / s] \supseteq \{0x, 1y\}^*$. For any α in $\{0x, 1y\}^+$,

$$\alpha \backslash [r \backslash L_0 / s] \supseteq \{0x, 1y\}^*.$$

Let $h: \{0, 1\}^* \to \{0, 1\}^*$ be the one-to-one homomorphism defined by $h(0) = 0x$ and $h(1) = 1y$. Then

$$h^{-1}[\alpha \backslash [r \backslash L_0 / s]] = \{0, 1\}^*.$$

But

$$\mathcal{L}(\mathcal{P}, r, s, x, y) = \{h^{-1}(L') | L' = \alpha \backslash [r \backslash L_0 / s], \ \alpha \in \{0x, 1y\}^+\}.$$

Thus $\mathcal{L}(\mathcal{P}, r, s, x, y) = \{\{0, 1\}^*\}$. □

Corollary 2.6. appears in [19], and extends results in [13], where it is shown that "$L(G) = L_0$" is undecidable for all unbounded regular sets L_0. The reader should note that, for all L_0 not equal to $\{0, 1\}^*$, both of the predicates "$L(G) = L_0$" and "$L(G) \neq L_0$" do not satisfy the conditions of Theorem 2.1.

Next we show that the undecidability of many known undecidable predicates on

the cfls is a special case of the closure properties of the finitely inherently ambiguous cfls. We need one technical lemma.

LEMMA 2.7. 1) *The finitely inherently ambiguous cfls are closed under inverse homomorphisms.*

2) *The finitely inherently ambiguous cfls are closed under quotient with singletons on the left and on the right.*

Proof. The proofs of 1) and 2) are very similar and fairly easy. We sketch the proof of 1). Let L be any cfl of finite degree of inherent ambiguity. Then there exists a pushdown automaton M such that $L = L(M)$; and there exists a positive integer k such that each string x accepted by M is accepted by at most k distinct sequences of moves of M. Let h be any homomorphism. The pushdown automaton M' described below accepts $h^{-1}(L)$. For all inputs y to M', M' applies h to y one character at a time in its finite control. M' simulates M on $x = h(y)$; and it accepts y if and only if M accepts x. Thus for each y in $h^{-1}(L)$ the number of distinct accepting sequences of moves of M' on y equals the number of distinct accepting sequences of moves on $x = h(y)$. Thus $h^{-1}(L) = L(M')$ is of finite degree of inherent ambiguity. □

THEOREM 2.8. *Let $\mathcal{S}$ be any subset of the finitely inherently ambiguous cfls over $\{0, 1\}$ such that there exists a language L_t in $\mathcal{S}$, where L_t has an unbounded regular subset. Then for arbitrary cfg G the predicate "$L(G)$ is a member of $\mathcal{S}$" is undecidable.* □

Proof. By Lemma 2.7 the finitely inherently ambiguous cfls over $\{0, 1\}$ are closed under all inverse homomorphisms and quotient with singletons on both the left and on the right. Thus for all such $\mathcal{S}$, $\mathcal{L}(\mathcal{P}, r, s, x, y)$ is a subset of the cfls over $\{0, 1\}$ of finite degree of inherent ambiguity. Since there are infinitely inherently ambiguous cfls over $\{0, 1\}$, the theorem follows immediately from Theorem 2.4. □

The applicability and power of Theorems 2.4 and 2.8 is shown by the following corollary of Theorem 2.8.

THEOREM 2.9. *The following classes $\mathcal{S}$ of cfls over $\{0, 1\}$ satisfy the conditions of Theorem 2.8. Thus for arbitrary cfg G, the predicate "$L(G)$ is a member of $\mathcal{S}$" is undecidable.*

1) *the finitely inherently ambiguous cfls;*
2) *for all positive integers k greater than 1 the cfls of degree of inherent ambiguity equal to, or less than or equal to k;*
3) *the unambiguous cfls;*
4) *the one-one linear cfls;*
5) *the RPP languages;*
6) *the LR regular languages;*
7) *the $LR(1, \infty)$ languages;*
8) *the full SPM parsable languages;*
9) *the FPFAP languages;*
10) *the BCP languages;*
11) *the deterministic cfls;*
12) *for all k greater than or equal to 1, the $ELC(k)$ languages;*
13) *the ELC languages, i.e. $\mathcal{S} = \{L \mid L$ is an $ELC(k)$ language for some $k\}$;*
14) *for all k greater than or equal to 1, the $LL(k)$ languages;*
15) *the LL languages, i.e. $\mathcal{S} = \{L \mid L$ is an $LL(k)$ language for some $k\}$;*
16) *the strict deterministic languages;*
17) *the real-time strict deterministic languages;*
18) *the $LR(0)$ languages;*
19) *the s-languages;*
20) *the simple precedence languages;*

21) *the operator precedence languages*;

22) *the regular sets*; *and*

23) *the reversal of any of the above language classes.*

Proof. For the definitions of the above language classes, see [2] for 4); [29] for 5), 7), and 9); [6] for 6); [7] for 8); [30] for 10); [1] for 11), 18), 19), 20), and 21); [4] for 12) and 13); [26] for 14) and 15); and [10] for 16) and 17).

The proofs that classes 1) through 22) satisfy the conditions of Theorem 2.8 follow easily from their definitions and known inclusion properties. The result for 23) follows since the class of cfls over $\{0, 1\}$ of finite degree of inherent ambiguity is closed under reversal. □

Theorems 2.8 and 2.9 show why the Post's correspondence problem was used almost identically to prove that several of the classes of cfls mentioned in Theorem 2.9 are undecidable. We hope that future proofs that subclasses $\mathscr{C}$ of the finitely inherently ambiguous cfls are undecidable will only consist of verifications that some element of $\mathscr{C}$ has an unbounded regular subset. Results for classes of context-free grammars rather than context-free languages appear in [20].

Analogues of Theorems 2.1, 2.4, and 2.6 hold for many other families of languages as well (see [18]). For example, analogues of Theorems 2.1, 2.4, 2.6, 2.8, and 2.9 hold for the linear cfls, the metalinear cfls, the online one counter languages, and the least AFL containing $\{a^n b^n | n \leqq 1\}$. (See [3] and [11] for properties of the last two families of languages.) We present one such theorem for the linear cfls.

THEOREM 2.10. *Let $\mathscr{P}$ be any predicate on the linear cfls over $\{0, 1\}$ for which there exists a linear cfl L_t and strings r, s, x, and y in $\{0, 1\}^*$ such that*

(a) *$\mathscr{P}(L_t)$ is true,*

(b) *$r\cdot\{0x, 1y\}^*\cdot s \subseteq L_t$, and*

(c) *$\mathscr{L}(\mathscr{P}, r, s, x, y)$ is a proper subset of the linear cfls over $\{0, 1\}$;*

or

(d) *$\mathscr{P}(L_t)$ is true*

(e) *$r\cdot\{x0, y1\}^*\cdot s \subseteq L_t$, and*

(f) *$\mathscr{R}(\mathscr{P}, r, s, x, y)$ is a proper subset of the linear cfls over $\{0, 1\}$.*

Then for arbitrary linear cfg G the predicate $\mathscr{P}(L(G))$ is undecidable.

The proof of Theorem 2.10 is almost identical to that of Theorem 2.4 and is left to the reader.

Finally, we present one possible analogue of Theorem 2.1 for the context-sensitive languages.

THEOREM 2.11. *Let $\mathscr{P}$ be any nontrivial predicate on the csls over $\{0, 1\}$ such that $\mathscr{P}_{\text{left}}$ or $\mathscr{P}_{\text{right}}$ is a proper subset of the csls over $\{0, 1\}$. Then for arbitrary csg G, the predicate $\mathscr{P}(L(G))$ is undecidable. Similarly for arbitrary linear bounded automaton M, the predicate $\mathscr{P}(L(M))$ is undecidable.*

Proof. We only prove the theorem when $\mathscr{P}_{\text{left}}$ is a proper subset of the csls over $\{0, 1\}$. There are two cases to consider.

Case 1. There exists a csl L_t and a string r in $\{0, 1\}^*$ such that $r\cdot\{0, 1\}^* \subseteq L_t$ and $\mathscr{P}(L_t)$ is true.

Case 2. For no string r in $\{0, 1\}^*$ and csl L for which $r\cdot\{0, 1\}^* \subseteq L$ is $\mathscr{P}(L)$ true.

To prove Case 1 let L_f be a csl over $\{0, 1\}$ that is not an element of $\mathscr{P}_{\text{left}}$. Let $h: \{0, 1\}^* \to \{0, 1\}^*$ be the one-to-one homomorphism defined by $h(0) = 00$ and $h(1) = 01$. For any csg G a csg H can be constructed effectively such that

$$L(H) = L_1 \cup L_2 \cup L_3$$

where

$$L_1 = r \cdot h(L(G)) \cdot 10 \cdot \{0, 1\}^*$$

$$L_2 = r \cdot \{00, 01\}^* \cdot 10 \cdot L_f$$

$$L_3 = L_t \cap \sim [r \cdot \{00, 01\}^* \cdot 10 \cdot \{0, 1\}^*].$$

As in the proofs of Theorems 2.1 and 2.4, $\mathcal{P}(L(H))$ is true if and only if $L(G) = \{0, 1\}^*$. Since the predicate "$L(G) = \{0, 1\}^*$" is undecidable for the csgs, so is $\mathcal{P}(L(G))$.

To prove Case 2 for any csg G, a csg H can be constructed effectively such that $L(H) = L(G)\cdot\{0, 1\}^* \cup L_t$, where $\mathcal{P}(L_t)$ is true. But $\mathcal{P}(L(H))$ is true if and only if $L(G) = \varnothing$. This follows since if $L(G) = \varnothing$, then $L(H) = L_t$. If $L(G) \neq \varnothing$, there is a string r in $L(G)$; and, hence, $r \cdot \{0, 1\}^* \subseteq L(H)$.

Since the predicate "$L(G) = \varnothing$" is undecidable for the csgs, so is $P(L(G))$. □

One immediate corollary is the following.

COROLLARY 2.12. *Let L_0 be any csl over $\{0, 1\}$. Then for arbitrary csg G the predicate "$L(G) = L_0$" is undecidable.*

Proof. There are two cases to consider.

Case 1. There exists $r \in \{0, 1\}^*$ such that $r \cdot \{0, 1\}^* \subseteq L_0$. Then, for all $y \in \{0, 1\}^*$, L_0/y is infinite. Hence, every language in $\mathcal{P}_{\text{right}}$ is infinite.

Case 2. For no string $r \in \{0, 1\}^*$ is $r \cdot \{0, 1\}^* \subseteq L_0$. Then $\{0, 1\}^* \notin \mathcal{P}_{\text{left}}$. □

3. Hard problems of the regular sets. Many of the undecidable properties that satisfy the conditions of Theorems 2.1, 2.4, and 2.11 become "hard" but decidable when restricted to the regular sets. In fact almost identical analogues of these theorems hold for the regular sets.

Our first theorem is a regular set analogue of Theorem 2.1.

THEOREM 3.1. *Let $\mathcal{P}$ be any predicate on the regular sets over $\{0, 1\}$ such that $\mathcal{P}(\{0, 1\}^*)$ is true and such that $\mathcal{P}_{\text{left}}$ or $\mathcal{P}_{\text{right}}$ is a proper subset of the regular sets over $\{0, 1\}$. Then the following hold.*

1) *$\{R | R$ is a $(\cup, \cdot, *)$ regular expression over $\{0, 1\}$ and $\mathcal{P}(L(R))$ is false$\}$ is PSPACE-hard; requires space greater than n^r i.o., for all r less than 1, on any nondeterministic Tm; and is an element of DCSL only if DCSL = NDCSL.*

2) *$\{M | M$ is a 2dfa and $P(L(M))$ is false$\}$ is PSPACE-hard; and requires space greater than n^r i.o., for all r less than 1, on any nondeterministic Tm.*

3) *$\{R | R$ is a $(\cup, \cdot, *, \cap)$ regular expression over $\{0, 1\}$ and $\mathcal{P}(L(R))$ is false$\}$ requires space greater than $2^{cn/(\log n)}$ i.o., for some c greater than 0, on any Tm.*

4) *$\{R | R$ is a $(\cup, \cdot, *, ^2)$ regular expression over $\{0, 1\}$ and $\mathcal{P}(L(R))$ is false$\}$ requires space greater than 2^{cn} i.o., for some c greater than 0, on any Tm.*

5) *$\{G | G$ is a nonselfembedding cfg with terminal alphabet $\{0, 1\}$ and $\mathcal{P}(L(G))$ is false$\}$ requires space greater than $2^{cn/(\log n)}$ i.o., for some c greater than 0, on any Tm.*

6) *$\{R | R$ is a $(\cup, \cdot, *, \sim)$ regular expression over $\{0, 1\}$ and $\mathcal{P}(L(R))$ is false$\}$ requires space greater than $F(c \cdot \log n)$ i.o., for some c greater than 0, on any Tm.*

7) *$\{R | R$ is a $(\cup, \cdot, *, \oplus)$ regular expression over $\{0, 1\}$ and $\mathcal{P}(L(R))$ is false$\}$ requires space reater than $F(c \cdot \log n)$ i.o., for some c greater than 0, on any Tm.*

Proof. The proof closely follows that of Theorem 2.1. From Theorem 1.14 the conclusions of 1) through 7) hold when $\mathcal{P}$ is "$L(R)$, $L(M)$, or $L(G) = \{0, 1\}^*$." Let $\mathcal{P}_{\text{left}}$ be a proper subset of the regular sets over $\{0, 1\}$ and let L_f be a regular set over $\{0, 1\}$ that is not in $\mathcal{P}_{\text{left}}$. Let $h: \{0, 1\}^* \to \{0, 1\}^*$ be the one-to-one homomorphism defined by $h(0) = 00$ and $h(1) = 01$. Let

$$L_1 = [\{00, 01\}^* \cdot 10 \cdot L_f] \cup [\sim[\{00, 01\}^* \cdot 10 \cdot \{0, 1\}^*]].$$

From any regular set descriptor R of type 1) through 7), a set descriptor S of the same type can be constructed deterministically using only linear space and polynomial time in $|R|$ such that

$$L(S) = h(L(R))\cdot 10\cdot\{0, 1\}^* \cup L_1.$$

Moreover, if R is a descriptor of the types of 1) or 2), then it is easily seen that S is log computable from R. The construction of S from R is only complicated for the 2dfa. The construction in this case is based on having the 2dfa for $L(S)$ first simulate a fixed 2dfa for L_1 that always halts. (Note that L_1 is independent of R). If the simulated 2dfa rejects the input string, the 2dfa for S then simulates R using 10 as a right endmarker, and the encoding h of 0 and 1.

$\mathcal{P}(L(S))$ is true if and only if $L(R) = \{0, 1\}^*$. There are two cases to consider.

Case 1. If $L(R) = \{0, 1\}^*$, then $L(S) = \{0, 1\}^*$. By assumption $\mathcal{P}(L(S))$ is true.

Case 2. If $L(R) \subsetneq \{0, 1\}^*$, then there exists a string w in $\{0, 1\}^* - L(R)$. As in the proof of Theorem 2.1, $h(w)\cdot 10 \backslash L(S) = L_f$. Since L_f is a regular set over $\{0, 1\}$ not in $\mathcal{P}_{\text{left}}$, $\mathcal{P}(L(S))$ is false.

Thus $(L(S))$ is true if and only if $L(R) = \{0, 1\}^*$ as claimed and the conclusions of 1) through 7) follow. The proof, when $\mathcal{P}_{\text{right}}$ is a proper subset of the regular sets over $\{0,1\}$, is left to the reader. □

Our next theorem is an analogue of Theorem 2.4.

THEOREM 3.2. *Let $\mathcal{P}$ be any predicate on the regular sets over $\{0, 1\}$ such that there exists a regular set L_t and strings r, s, x, and y in $\{0, 1\}^*$ for which*

(a) $\mathcal{P}(L_t)$ *is true,*

(b) $r\cdot\{0x, 1y\}^*\cdot s \subseteq L_t$, *and*

(c) $\mathcal{L}(\mathcal{P}, r, s, x, y)$ *is a proper subset of the regular sets over* $\{0, 1\}$;

or

(d) $\mathcal{P}(L_t)$ *is true,*

(e) $r\cdot\{x0, y1\}^*\cdot s \subseteq L_t$, *and*

(f) $\mathcal{R}(\mathcal{P}, r, s, x, y)$ *is a proper subset of the regular sets over* $\{0, 1\}$.

Then the following hold:

1) $\{R|R$ *is a* $(\cup, \cdot, *)$ *regular expression over* $\{0, 1\}$ *and* $\mathcal{P}(L(R))$ *is false*$\}$ *is PSPACE-hard; requires space greater than* n^r *i.o., for all* r *less than* 1, *on any nondeterministic Tm; and is an element of DCSL only if DCSL = NDCSL.*

2) $\{R|R$ *is a* $(\cup, \cdot, *, \cap)$ *regular expression over* $\{0, 1\}$ *and* $\mathcal{P}(L(R))$ *is false*$\}$ *requires space greater than* $2^{cn/(\log n)}$ *i.o., for some* c *greater than* 0, *on any Tm.*

3) $\{R|R$ *is a* $(\cup, \cdot, *, ^2)$ *regular expression over* $\{0, 1\}$ *and* $\mathcal{P}(L(R))$ *is false*$\}$ *requires space greater than* 2^{cn} *i.o., for some* c *greater than* 0, *on any Tm.*

4) $\{G|G$ *is a nonselfembedding cfg with terminal alphabet* $\{0, 1\}$ *and* $\mathcal{P}(L(G))$ *is false*$\}$ *requires space greater than* $2^{cn/(\log n)}$ *i.o., for some* c *greater than* 0, *on any Tm.*

Proof. The proof is almost identical to that of Theorem 2.4 and is left to the reader.

The third theorem is an analogue of Theorem 2.11.

THEOREM 3.3. *Let $\mathcal{P}$ be any nontrivial predicate on the regular sets over $\{0, 1\}$ such that $\mathcal{P}_{\text{left}}$ or $\mathcal{P}_{\text{right}}$ is a proper subset of the regular sets over $\{0, 1\}$. Then the following hold:*

1) $\{M|M$ *is a 2dfa and* $\mathcal{P}(L(M))$ *is false*$\}$ *is PSPACE-hard and requires space greater than* n^r *i.o., for all* r *less than* 1, *on any nondeterministic Tm.*

2) $\{R|R$ *is a* $(\cup, \cdot, *, \cap)$ *regular expression over* $\{0, 1\}$ *and* $\mathcal{P}(L(R))$ *is false*$\}$ *is PSPACE-hard.*

3) $\{R|R$ *is a* $(\cup, \cdot, *, \sim)$ *regular expression over* $\{0, 1\}$ *and* $\mathcal{P}(L(R))$ *is false*$\}$ *requires space greater than* $F(c\cdot\log n)$ *i.o., for some* c *greater than* 0, *on any Tm.*

4) $\{R|R$ *is a* $(\cup, \cdot, *, \oplus)$ *regular expression over* $\{0, 1\}$ *and* $\mathcal{P}(L(R))$ *is false*$\}$ *requires space greater than* $F(c\cdot\log n)$ *i.o., for some* c *greater than* 0, *on any Tm.*

Proof. The proof of Theorem 3.3 is similar to that of Theorem 2.11 once "$=\varnothing$" and "$=\{0, 1\}^*$" are known to have the corresponding complexities (Theorems 1.14 and 1.15). The major difference in the proof is that for Case 2, we let $h:\{0, 1\}^* \to \{0, 1\}^*$ be the one-to-one homomorphism defined by $h(0)=00$ and $h(1)=01$, and let

$$L(H)=h(L(G))\cdot 10\cdot\{0, 1\}^*\cdot \cup L_t.$$

This encoding ensures that when G is a 2dfa, H can be obtained from G (by using 10 as an endmarker). The details of the proof are left to the reader. □

We illustrate the power and applicability of Theorems 3.1, 3.2, and 3.3.

THEOREM 3.4. *The following predicates on the regular sets over* $\{0, 1\}$ *satisfy the conditions of Theorem* 3.2 *and of Theorem* 3.3.

1) *For all unbounded regular sets* L_0, "$=L_0$";

2) *L is a star event, i.e.* $L=(L)^*$;

3) *L is a code event, i.e. there exist strings* $w_1, \ldots, w_k$ *in* $\{0, 1\}^*$ *such that* $L=\{w_1, w_2, \cdots, w_k\}^*$;

4) *For all* $k \geqq 1$, *L is a k-parsable event*; *and L is a locally parsable event*;

5) *L is an ultimate definite event, reverse ultimate definite event, or generalized ultimate definite event*;

6) *L is a comet event, reverse comet event, or generalized comet event*;

7) $L=\gamma(L)$, *where* $\gamma(L)=\{y|$ *there exists x in L such that* $|y|=|x|\}$;

8) *L is prefix closed*; *i.e.,* $L=\{x|$ *there exists y in* $\{0, 1\}^*$ *and* $x\cdot y \in L\}$;

9) *L is suffix closed, i.e.,* $L=\{y|$ *there exists x in* $\{0, 1\}^*$ *and* $x\cdot y \in L\}$;

10) *L is infix closed, i.e.,* $L=\{y|$ *there exist x, z in* $\{0, 1\}^*$ *and* $x\cdot y\cdot z \in L\}$;

11) *L is cofinite*;

12) *For all* $k \geqq 1$, *L is a k-definite event, k-reverse definite event, or k-generalized definite event*;

13) *L is a definite, reverse definite, or generalized definite event*;

14) *For all* $k \geqq 1$, *L is a k-testable event*;

15) *For all* $k \geqq 1$, *L is k-testable in the strict sense*;

16) *L is locally testable in the strict sense*;

17) *L is locally testable*;

18) *L is a loop-free or FOL event*;

19) *L is a star-free, noncounting, group-free, permutation-free, or LTO event*;

20) *For all* $k > 2$, *L is a CMk event*;

21) *For all* $k \geqq 1$, *L is of star height equal to, or less than or equal to, k*;

22) *L is accepted by some strongly connected deterministic finite automaton*;

23) *L is accepted by some permutation automaton*;

24) *L is a pure group event*;

25) $L=[L]^{\text{rev}}$; *and*

26) *L is dot-free, i.e. L is denoted by some* $(\cup, \cdot, *, \sim)$ *regular expression over* $\{0, 1\}$ *with no occurrence of* "$\cdot$".

Moreover, predicates 2) *through* 17), 19), 20), *and* 22) *through* 26) *also satisfy the conditions of Theorem* 3.1.

Proof. The definitions of the classes of regular sets of 3), 4), and 12) through 21) can be found in [23]. The definition of 5) can be found in [25], 6) in [5], 22) in [12], 23) in [31], and 24) in [22].

The proof for each of the above predicates consists of two parts. The first part consists of observing that the predicates we claim satisfy Theorem 3.1 are true for $\{0, 1\}^*$, and that each of the remaining predicates is true for some unbounded regular set. The second part of the proof consists of showing that $\mathscr{P}_{\text{left}}$ or $\mathscr{P}_{\text{right}}$ is a proper subset

of the regular sets over $\{0, 1\}$; and for predicates 1), 18), and 21), that for the appropriate r, s, x, and y in $\{0, 1\}^*$, $\mathscr{L}(\mathscr{P}, r, s, x, y)$ is a proper subset of the regular sets over $\{0, 1\}$.

We now consider the second part of the proof for each predicate.

1): From Lemma 1.10 a regular set L_0 over $\{0, 1\}$ is unbounded if and only if there exist strings r, s, x, and y in $\{0, 1\}^*$ such that $r\cdot\{0x, 1y\}^*\cdot s \subseteq L_0$. As in the proof of Corollary 2.6, $\mathscr{L}(\mathscr{P}, r, s, x, y) = \{\{0, 1\}^*\}$. Next, we note that an argument analogous to that in the proof of Corollary 2.11 shows that $\mathscr{P}_{\text{left}}$ or $\mathscr{P}_{\text{right}}$ is a proper subset of the regular sets over $\{0, 1\}$.

The proofs of 2) through 6) are similar. We only prove 6). A regular set L is a comet event if and only if there exist regular sets L_1 and L_2 such that $L = L_1 \cdot L_2$, $L_1 = L_1^*$, and $L_1 \neq \{\lambda\}$. Let $\mathscr{P}$ be the predicate "L is a comet event." Then $\mathscr{P}_{\text{right}} \subseteq \{\varnothing\} \cup \{L | L \text{ is an infinite regular set over } \{0, 1\}\}$. This follows since for all x in $\{0, 1\}^*$, either $L/x = \varnothing$ or there exists a string y in L and w in L/x such that $y/x = w$. Suppose L/x is not empty. Then there exists a nonnull string z in L_1. Hence for all $k \geqq 0$, $z^k \cdot y$ is in L and $z^k \cdot y/x = z^k \cdot w$. Thus L/x is infinite.

7): $L = \gamma(L)$ implies for all x in $\{0, 1\}^+$ that $L/x = \gamma(L/x)$. This follows since if y is an element of L/x, then $y \cdot x$ is in L. Thus $L = \gamma(L)$ implies for all strings z such that $|z| = |y \cdot x|$, z is in L. Thus for all strings z' such that $|z'| = |y|$, $z' \cdot x$ is in L.

The proofs of 8), 9), and 10) are similar. We only prove 8). $L = \{x |$ there exists y in $\{0, 1\}^*$ such that $x \cdot y \in L\}$ implies for all x in $\{0, 1\}^*$ that $x \backslash L$ is prefix closed. This follows since y in $x \backslash L$ implies $x \cdot \text{Init}(y) \subseteq \text{Init}(L) = L$. Thus for all y in $x \backslash L$, $\text{Init}(y) \subseteq x \backslash L$.

11): L is cofinite implies for all x in $\{0, 1\}^*$ that $x \backslash L$ is cofinite. The proof is left to the reader.

The proofs of 12) through 18) all follow from that of 19), since any regular set, satisfying one of the predicates of 12) through 18), satisfies the predicate of 19). We note that McNaughton and Papert [23] have shown that the classes of star-free, noncounting, group-free, permutation-free, and LTO events are the same.

By definition a regular set L is a noncounting event over $\{0, 1\}$ if and only if for some $n \geqq 1$, for all positive integers j, and for all strings u, v, and w in $\{0, 1\}^*$, $u \cdot v^{n+j} \cdot w$ is in L if and only if $u \cdot v^n \cdot w$ is in L. Let L be a noncounting event. Let z be any string over $\{0, 1\}$. Then there exists an $n \geqq 1$, such that for all positive integers j, and for all strings u, v, and w in $\{0, 1\}^*$, $z \cdot u \cdot v^{n+j} \cdot w$ is in L if and only if $z \cdot u \cdot v^n \cdot w$ is in L. Hence for all strings z in $\{0, 1\}^*$, L noncounting implies that $z \backslash L$ is noncounting.

The proof of 20) is similar to that of 19) noting that for all $k \geqq 2$, there are regular sets which are elements of $CM(k+1)$ but not elements of CMk, e.g. $(0^{k+1})^*$. By definition L is in CMk if and only if there exists an $n \geqq 1$ such that for all positive integers j and strings u, v, and w over $\{0, 1\}$, $u \cdot v^{n+kj} \cdot w$ is in L if and only if $u \cdot v^n \cdot w$ is in L.

21): The proof consists of the following two facts:

(a) there are regular sets over $\{0, 1\}$ of star height k for all $k \geqq 0$ and

(b) quotient with single string does not raise star height.

22): If L is accepted by some strongly connected deterministic finite automaton, then for x in $\{0, 1\}^+$ either $x \backslash L$ is empty or infinite. Let $M = (K, \{0, 1\}, \delta, q_0, F)$ be some strongly connected automaton which accepts L. Then $x \backslash L \neq \varnothing$ implies that there exists y in $\{0, 1\}^*$ such that $\delta(q_0, x \cdot y)$ is in F. But M strongly connected implies that there exists z in $\{0, 1\}^+$ such that $\delta(\delta(q_0, x \cdot y), z) = q_o$. Hence for all $k \geqq 0$, $x \cdot y \cdot (z \cdot x \cdot y)^k$ is in L.

23): If L is accepted by a permutation automaton, then for all x in $\{0, 1\}^+$, $x \backslash L$ is either empty or infinite. The proof is similar to that of (22) and is left to the reader.

24): By definition L is a pure-group event if the syntactic monoid of L, denoted by $\text{Syn}(L)$, is a group. The elements of $\text{Syn}(L)$ are the congruence classes $[x]$ defined by $[x] = \{y |$ for all u, v in $\{0, 1\}^*$, $u \cdot y \cdot v$ is in L if and only if $u \cdot x \cdot v$ is in $L\}$. Again for all x in

$\{0, 1\}^+$, $x\backslash L$ is either empty or infinite. Suppose $x\backslash L \neq \varnothing$. Let y be an element of $x\backslash L$. Since $\mathrm{Syn}(L)$ is a group, for all x in $\{0, 1\}^*$ there exists x' in $\{0, 1\}^*$ such that $[x \cdot x'] = [x] \cdot [x'] = [\lambda]$. Thus there exists z in $\{0, 1\}^*$ such that $[x \cdot y] \cdot [z] = [x \cdot y \cdot z] = [\lambda]$. Thus for all $k \geqq 0$, $([x \cdot y \cdot z])^k \cdot [x \cdot y] = [x \cdot y]$. Hence for all $k \geqq 0$, $(x \cdot y \cdot z)^k \cdot x \cdot y$ is in L and for all $k \geqq 1$, $(y \cdot z) \cdot (x \cdot y \cdot z)^{k-1} \cdot x \cdot y$ is in $x\backslash L$.

25): The proof is identical to that in Proposition 2.3.

26): If L is dot-free, then $L = [L]^{\mathrm{rev}}$ as is easily seen by induction on the depth of nesting of regular operators appearing in some dot-free $(\cup, \cdot, *, \sim)$ regular expression R for which $L = L(R)$. Thus the proof of Proposition 2.3 applies. □

THEOREM 3.5. *The following predicates satisfy the conditions of Theorem* 3.3:

1) *for all bounded regular sets L over* $\{0, 1\}$, "$=L$";

2) *L is finite*;

3) *L is commutative, i.e. for all x, y in L,* $x \cdot y = y \cdot x$; *and*

4) *L is bounded.*

Proof. 1), 2), and 4) are obvious. 3) follows since L is commutative if and only if $L \subseteq w^*$ for some word w (see [8].) Predicates 1) through 4) do not satisfy the conditions of Theorems 3.1 or 3.2. □

COROLLARY 3.6. *For all unbounded regular sets* L_0 *over* $\{0, 1\}$ *the following hold.*

1) $\{R | R$ *is a* $(\cup, \cdot, *)$ *regular expression over* $\{0, 1\}$ *and* $L(R) \neq L_0\}$ *is PSPACE-complete*; *requires space greater than* n^r *i.o., for all r less than* 1, *on any nondeterministic Tm*; *and is an element of DCSL if and only if DCSL = NDCSL.*

2) $\{R | R$ *is a* $(\cup, \cdot, *, \cap)$ *regular expression over* $\{0, 1\}$ *and* $L(R) \neq L_0\}$ *requires space greater than* $2^{cn/(\log n)}$ *i.o., for some c greater than* 0, *on any Tm.*

3) $\{R | R$ *is a* $(\cup, \cdot, *, ^2)$ *regular expression over* $\{0, 1\}$ *and* $L(R) \neq L_0\}$ *requires space greater than* 2^{cn} *i.o., for some c greater than* 0, *on any Tm.*

4) $\{G | G$ *is a nonselfembedding cfg with terminal alphabet* $\{0, 1\}$ *and* $L(G) \neq L_0\}$ *requires space greater than* $2^{cn/(\log n)}$ *i.o., for some c greater than* 0, *on any Tm.*

Proof. 1) through 4) follow immediately from Theorem 3.2 and Theorem 3.4 part 1). The statement of 1) reflects the fact that the predicate in 1) is a member of *NDCSL* [28]. □

COROLLARY 3.7. *For all regular sets* L_0 *over* $\{0, 1\}$ *the following hold.*

1) $\{M | M$ *is a 2dfa and* $L(M) \neq L_0\}$ *is PSPACE-complete and requires space greater than* n^r *i.o., for all r less than* 1, *on any nondeterministic Tm.*

2) $\{R | R$ *is a* $(\cup, \cdot, *, \sim)$ *regular expression over* $\{0, 1\}$ *and* $L(R) \neq L_0\}$ *requires space greater than* $F(c \cdot \log n)$ *i.o., for some c greater than* 0, *on any Tm.*

3) $\{R | R$ *is a* $(\cup, \cdot, *, \oplus)$ *regular expression over* $\{0, 1\}$ *and* $L(R) \neq L_0\}$ *requires space greater than* $F(c \cdot \log n)$ *i.o., for some c greater than* 0, *on any Tm.*

Proof. 1) through 3) follow immediately from Theorem 3.3, if we note that the equivalence problem for *2dfa* is decidable by a polynomially space-bounded Tm. This follows from the construction in [14], given an arbitrary 2dfa, of an equivalent deterministic finite automaton. □

Corollaries 3.6 and 3.7 provide a whole class of new provably hard sets and nonpolynomial lower time or space complexity bounds.

4. Conclusion. We have studied the complexity of a variety of problems on the regular sets and the context-free languages. The main technique used was that of embedding "$=\{0, 1\}^*$" and "$=\varnothing$" into other predicates. In § 2 the undecidability of "$=\{0, 1\}^*$" for the context-free languages was exploited to provide sufficient conditions for the undecidability of predicates on the context-free languages. In § 3 the same techniques were applied to the regular sets. Many predicates studied in the literature satisfy the conditions of our theorems. Related results appear in [18] and [19].

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling*, vol. 1 and 2, Prentice-Hall, Engelwood Cliffs, NJ, 1972 and 1973.

[2] V. AMAR AND G. PUTZOLU, *On a family of linear grammars*, Information and Control, 7 (1964), pp. 283–291.

[3] B. S. BAKER AND R. V. BOOK, *Reversal bounded multi-pushdown machines*, IEEE Conf. Record 13th Annual Symp. on Switching and Automata Theory (Oct. 1972), pp. 207-211.

[4] B. M. BROSGOL, *Deterministic translation grammars*, Ph.D. thesis, Harvard Univ., Cambridge, MA, 1974.

[5] J. A. BRZOZOWSKI AND R. COHEN, *On the decomposition of regular events*, J. Assoc. Comput. Mach., 18 (1969), pp. 4–18.

[6] K. CULIK, II AND R. COHEN, *LR-regular grammars—An extension of LR(k) grammars*, J. Comput. System Sci., 7 (1973), pp. 66–96.

[7] C. N. FISCHER, *On parsing context-free languages in parallel environments*, Ph.D. thesis, Cornell Univ., Ithaca, NY, 1975.

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

[9] S. A. GREIBACH, *A note on undecidable properties of formal languages*, Math. Systems Theory, 2(1968), pp. 1–6.

[10] M. A. HARRISON AND I. V. HAVEL, *Real-time strict deterministic languages*, this Journal, 1 (1972), pp. 333–349.

[11] J. HARTMANIS AND J. E. HOPCROFT, *What makes some language theory problems undecidable*, J. Comput. System Sci., 4 (1970), pp. 368–376.

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

[13] J. E. HOPCROFT *On the equivalence and containment problems for context-free languages*, Math. Systems Theory, 3 (1969), pp. 119–124.

[14] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[15] H. B. HUNT, III, *On the time and tape complexity of languages*, Ph.D. thesis, Cornell Univ. Ithaca, NY, 1973.

[16] ———, *On the complexity of finite, pushdown, and stack automata*, Math. Systems Theory, 10 (1976), pp. 33–52.

[17] ———, *On the complexity of regular set descriptors*, to appear.

[18] H. B. HUNT, III AND D. J. ROSENKRANTZ, *On equivalence and containment problems for formal languages*, J. Assoc. Comput. Mach., 24 (1977), pp. 387–396.

[19] H. B. HUNT, III, D. J. ROSENKRANTZ AND T. G. SZYMANSKI, *On the equivalence, containment, and covering problems for the regular and context-free languages*, J. Comput. System Sci., 12 (1976), pp. 222–268.

[20] H. B. HUNT, III AND T. G. SZYMANSKI, *Complexity metatheorems for context-free grammar problems*, Ibid., 13 (1976), pp. 318–334.

[21] A. J. KORENJAK AND J. E. HOPCROFT, *Simple deterministic languages*, IEEE Conf. Record 7th Annual Symp. on Switching and Automata Theory (Oct. 1966), pp. 36–46.

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

[23] R. MCNAUGHTON AND S. PAPERT, *Counter-Free Automata*, MIT Press, Cambridge, MA, 1971.

[24] W. F. OGDEN, *A helpful result for proving inherent ambiguity*, Math. Systems Theory, 2 (1968), pp. 191–194.

[25] A. PAZ AND B. PELEG, *Ultimate-definite and symmetric definite events and automata*, J. Assoc. Comput. Mach., 12 (1965), pp. 399–410.

[26] D. J. ROSENKRANTZ AND R. E. STEARNS, *Properties of deterministic top-down grammars*, Information and Control, 17 (1970), pp. 226–256.

[27] L. J. STOCKMEYER, *The complexity of decision problems in automata theory and logic*, Rep. TR-133, Project MAC, Mass. Inst. of Tech., Cambridge, MA, 1974.

[28] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time: Preliminary report*, Proc. Fifth Annual ACM Symposium on Theory of Computing (May 1973), pp. 1–9.

[29] T. G. SZYMANSKI, *Generalized bottom-up parsing*, Ph.D. thesis, Cornell Univ., Ithaca, NY 1973.

[30] T. G. SZYMANSKI AND J. H. WILLIAMS, *Noncanonical parsing*, IEEE Conf. Record 14th Annual Symp. On Switching and Automata Theory (Oct. 1973), pp. 122–129.

[31] G. THIERRIN, *Permutation automata*, Math. Systems Theory, 2 (1968), pp. 83–90.

SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

THE TIME MEASURE OF ONE-TAPE TURING MACHINES DOES NOT HAVE THE PARALLEL COMPUTATION PROPERTY*

JOACHIM BISKUP†

Abstract. J. Hartmanis conjectured that the time measure of one-tape Turing machines does not have the parallel computation property. We prove this conjecture by constructing one-tape Turing machines M_1 and M_2 such that no one-tape Turing machine can simulate them in parallel. This is shown using a result of F. C. Hennie that nonregular sets cannot be accepted in linear time.

Key words. abstract complexity measure, parallel computation property, one-tape Turing machines, time measure, recognizers, nonregular sets

L. H. Landweber and E. L. Robertson [7] defined that an (abstract) complexity measure $\langle(\varphi_i)_{i\in\mathbb{N}}, (C_i)_{i\in\mathbb{N}}\rangle$ in the sense of M. Blum [2] has the *parallel computation property* iff there exists a recursive function h such that for all i and j, and for all x

$$\varphi_{h(i,j)}(x) = \begin{cases} \varphi_i(x) & \text{if } C_i(x) \leqq C_j(x), \\ \varphi_j(x) & \text{otherwise} \end{cases} \tag{1}$$

and

$$C_{h(i,j)}(x) = \min [C_i(x), C_j(x)]. \tag{2}$$

J. Hartmanis [5] conjectured that the complexity measure, defined by the number of steps taken by one-tape Turing machines, does *not* have the parallel computation property. In this note we prove this conjecture for Turing machines recognizing sets. A similar proof for Turing machines computing word functions is given in a technical report [1]. For unexplained notations and further background the reader is referred to [3].

We consider the class $\mathfrak{M}$ of *one-tape Turing machines recognizing sets* over some fixed finite alphabet Σ with cardinality $c(\Sigma) \geqq 2$ by off-line computations as explained below (cf. [6], [4]). Any machine $M \in \mathfrak{M}$ has one tape the left end of which is indicated by the special marker $\square$ on tape square 0 and which is unbounded on the right. Prior to the start of a computation an input word $w = \sigma_1 \cdots \sigma_n \in \Sigma^*$ is written on the first n tape squares, and the remainder of the tape squares are left blank (denoted by the special symbol $\sqcup$). Furthermore the reading head is positioned on the tape square 1 (the first input symbol), and the machine is placed in a designated *starting state*. Then on each operation the machine M scans the tape square under the reading head, and depending on the symbol scanned and the present internal state it prints a new symbol on that tape square, moves the reading head one tape square to the right or to the left, and enters a new internal state. The machine stops if the new state is a halting state, which must be either an *accepting state* or a *rejecting state*. Formally a Turing machine $M \in \mathfrak{M}$ can be defined by a finite set of instructions of the form (present state, symbol scanned, new symbol, direction of moving, new state), such that no two different instructions are identical both in the first and the second component. Furthermore there are designated exactly one starting state and a set of accepting states and a set of rejecting states.

A Turing machine $M \in \mathfrak{M}$ *computes* a possibly partial 0-1-*valued function* $|M|: \Sigma^* \to \{0, 1\}$ in the following way. Let M be started with input w. If M eventually stops by entering an accepting state then $|M|(w) := 1$; if M eventually stops by entering

* Received by the editors May 6, 1977.

† Lehrstuhl für Angewandte Mathematik, insbesondere Informatik, RWTH Aachen, D-5100 Aachen, Germany.

a rejecting state then $|M|(w) := 0$; otherwise, if M does not stop, $|M|(w)$ is undefined. Interpreting $|M|$ as a characteristic function, we say that *M recognizes the set* $\{w | w \in \Sigma^* \text{ and } |M|(w) = 1\}$.

With each Turing machine $M \in \mathfrak{M}$ we associate a possibly partial *step counting function* $T_M: \Sigma^* \to \mathbb{N}$ such that $T_M(w)$ is the number of operations performed by M in processing the input word w. $T_M(w)$ is undefined if machine M does not stop when started with input word w.

Under an appropriate arithmetization of Σ^* and $\mathfrak{M}$ we can consider $\langle (|M|)_{M \in \mathfrak{M}}, (T_M)_{M \in \mathfrak{M}} \rangle$ as a complexity measure (for all computable 0-1-valued function) in the sense of M. Blum [2].

THEOREM. *The complexity measure* $\langle (|M|)_{M \in \mathfrak{M}}, (T_M)_{M \in \mathfrak{M}} \rangle$, *i.e. the time measure of one-tape Turing machines recognizing sets, does not have the parallel computation property.*

Proof. Below we shall construct Turing machines $M_1 \in \mathfrak{M}$ and $M_2 \in \mathfrak{M}$ with the following properties:

(A) The step counting functions T_{M_1} and T_{M_2} are linearly bounded by the lengths of the input words.

(B) The function

$$g(w) := \begin{cases} |M_1|(w) & \text{if } T_{M_1}(w) \leqq T_{M_2}(w), \\ |M_2|(w) & \text{otherwise} \end{cases} \tag{3}$$

is the (total) characteristic function of a *nonregular* context-free set.

Now assume that there exists a Turing machine $M \in \mathfrak{M}$ computing g with step counting function $T_M(w) = \min [T_{M_1}(w), T_{M_2}(w)]$. By (A), T_M is also linearly bounded by the lengths of the input words. Thus, by a result of F. C. Hennie [6, Thm. 3], M recognizes a *regular* set. But this is a contradiction to (B). Hence there cannot exist a function h with the properties (1), (2) required by the parallel computation property.

M_1 is defined by the following set of instructions:

$$(S, a, a, \text{right}, S) \quad (X, a, a, \text{right}, Y) \quad (Y, a, a, \text{right}, Y)$$

$$(S, b, b, \text{right}, X) \quad (X, b, b, \text{right}, X) \quad (Y, b, b, \text{right}, Y)$$

$$(S, \sqcup, \sqcup, \text{right}, A) \quad (X, \sqcup, \sqcup, \text{right}, A) \quad (Y, \sqcup, \sqcup, \text{right}, R)$$

where S is starting state, A is accepting state, and R is rejecting state.

The machine M_1 scans the input word from left to right exactly once recognizing the set $\{a^m b^n | m \geqq 0 \text{ and } n \geqq 0\}$. Thus we have

$$|M_1|(w) = \begin{cases} 1 & \text{if } w = a^m b^n \text{ with } m, n \geqq 0, \\ 0 & \text{otherwise,} \end{cases}$$

$$T_{M_1}(w) = \text{length } (w) + 1.$$

M_2 is defined by the following set of instructions:

$$(S, a, a, \text{right}, S) \quad (X, a, a, \text{left}, X)$$

$$(S, b, b, \text{left}, X) \quad (X, \square, \square, \text{right}, R)$$

$$(S, \sqcup, \sqcup, \text{right}, A)$$

where S is starting state, A is accepting state , and R is rejecting state.

The machine M_2 first scans the input word from left to right. If it finds the symbol b, it moves the reading head back to the tape square 1 and rejects the input word.

Otherwise, if the input word does not contain the symbol b, the machine accepts it after detecting its end. Thus we have

$$|M_2|(w) = \begin{cases} 1 & \text{if } w \in \{a\}^*, \\ 0 & \text{otherwise,} \end{cases}$$

$$T_{M_2}(w) = \begin{cases} \text{length}(w)+1 & \text{if } w \in \{a\}^*, \\ 2m+2 & \text{otherwise, that is if } w = a^m b\tilde{w} \\ & \text{with } m \geqq 0,\ \tilde{w} \in \{a, b\}^*. \end{cases}$$

We have to show that M_1 and M_2 satisfy the conditions (A) and (B). Condition (A) obviously holds. In order to prove (B) we look at the function g defined by (3) in detail:

w	$T_{M_1}(w)$	$T_{M_2}(w)$	$g(w)$
a^m with $m \geqq 0$	$m+1$	$m+1$	$\|M_1\|(w) = 1$
$a^m b^n$ with $m \geqq 0$, $n \geqq 1$	$m+n+1$	$2m+2$	$\|M_1\|(w) = 1$ if $m+n+1 \leqq 2m+2$ $\|M_2\|(w) = 0$ otherwise
$a^m b^n a\tilde{w}$ with $m \geqq 0$, $n \geqq 1$, $\tilde{w} \in \{a, b\}^*$	length $(w)+1$	$2m+2$	$\|M_1\|(w) = 0$ if length $(w)+1 \leqq 2m+2$ $\|M_2\|(w) = 0$ otherwise

Hence we have

$$g(w) = \begin{cases} 1 & \text{if } w = a^m b^n \text{ and } n \leqq m+1, \\ 0 & \text{otherwise.} \end{cases}$$

But it is well-known that $\{a^m b^n \mid n \leqq m+1\}$ is a *nonregular* context-free set. This proves property (B).

REFERENCES

[1] J. BISKUP, *A note on the time measure of one-tape Turing machines*, Schriften zur Informatik und Angewandten Mathematik, Bericht Nr. 32, RWTH Aachen, May 1977.

[2] M. BLUM, *A machine-independent theory of the complexity of recursive functions*, J. Assoc. Comput. Mach., 14 (1967), pp. 322–336.

[3] W. S. BRAINERD AND L. H. LANDWEBER, *Theory of Computation*, Wiley–Interscience, New York, 1974.

[4] J. HARTMANIS, *Computational complexity of one-tape Turing machine computations*, J. Assoc. Comput. Mach., 15 (1968), pp. 325–339.

[5] ———, *On the problem of finding natural computational complexity measures*, Proceedings of an International Symposium and Summer School on Mathematical Foundations of Computer Science, High Tatras, Sept. 3–8, 1973, pp. 95–103; Tech. Rep. TR 73–175, Dept. of Computer Sci., Cornell Univ., Ithaca, NY.

[6] F. C. HENNIE, *One-tape, off-line Turing machine computations*, Information and Control, 8 (1965), pp. 553–578.

[7] L. H. LANDWEBER AND E. L. ROBERTSON, *Recursive properties of abstract complexity classes*, J. Assoc. Comput. Mach., 19 (1972), pp. 296–308.

SIAM J. COMPUT.
Vol. 7, No. 1, February 1978

ERRATUM: A FAST MONTE-CARLO TEST FOR PRIMALITY*

R. SOLOVAY† AND V. STRASSEN‡

Allan Borodin has pointed out a slight error in the justification of our algorithm. We assert that if n is an odd composite integer then the Jacobi symbol, (a/n), is unequal to 1 for some a prime to n. But in fact (a/n) is identically equal to 1 when n is a perfect square (and only in that case).

Here is a repair for our analysis. Let n be odd and composite and suppose

$$a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n} \tag{1}$$

for all $a \in Z$ relatively prime to n. We shall derive a contradiction. First, suppose that n is not square free. Say $n = p^e \cdot q$ where p is an odd prime, $e > 1$, and q is relatively prime to p. It follows from (1) that $a^{n-1} \equiv 1 \pmod{n}$ for all a relatively prime to n. By the Chinese remainder theorem, $a^{n-1} \equiv 1 \pmod{p^e}$ for all a relatively prime to p. But $Z_{p^e}^x$ is cyclic of order $p^{e-1} \cdot (p-1)$. So $p^{e-1} \cdot (p-1)$ divides $n-1$. Since $e > 1$, p divides $n-1$ as well as n. This absurdity shows n is square free.

But if n is square free, the argument in our paper applies to show that since n has two distinct prime factors $(a/n) \equiv 1$ for a prime to n. Whence a is a perfect square and also a square free composite. Contradiction!

* This Journal, 6 (1977), pp. 84–85. Received by the editors, June 2, 1977.

† IBM Watson Laboratory, Yorktown Heights, New York. Now at Department of Mathematics, University of California, Berkeley, Berkeley, California 94720.

‡ Seminar für Angewandte Mathematik, Universität Zurich, 8032 Zurich, Switzerland.

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

ON STRUCTURE PRESERVING REDUCTIONS*

NANCY LYNCH† AND RICHARD J. LIPTON‡

Abstract. The concept of reduction between problems is strengthened. Certain standard problems are shown to be complete in the new and stronger sense. Applications to the number of solutions of particular problems are presented.

Key words. reductions, polynomial time, logspace, complete sets

1. Introduction. One of the most striking features of a large number of the known reductions of one problem to another [3] is that they often preserve a great deal more than they have to. More precisely, suppose that A is many-to-one polynomial time reducible to B where A and B are, as usual, subsets of Σ^* for some finite alphabet Σ. Then all that is required in the usual definition is that

$$(*) \qquad \forall x \in \Sigma^*, \quad x \in A \text{ if and only if } f(x) \in B,$$

where $f(x)$ is some polynomial time computable function. Essentially (*) states that x has a solution exactly when $f(x)$ has a solution. It appears, however, that quite often x and $f(x)$ are more closely related than this.

This imprecise intuitive feeling that reductions often preserve additional structure is the subject of this paper. We introduce a new kind of reduction and prove that some standard complete problems are also complete in our strong sense.

The notion that reduction preserves additional structure also appears in Simon [7]. His main result is that a number of problems are still equivalent when (*) is strengthened to:

$$x \text{ has the same number of solutions as } f(x).$$

He calls reducibilities preserving the number of solutions "parsimonious". There is a difficulty with these results, however; it is not clear what it means for x to have k solutions when A is an arbitrary set. Clearly, either x is in A or it is not. Simon avoids this difficulty by working only with well known and specific problems. In these cases it is reasonable to assume that "x has k solutions" is a meaningful concept. We take an alternative approach. The main virtue of this approach is that it allows us to work with arbitrary problems, and thus we can prove the existence of complete sets.

2. New definition of reduction. The key idea of the new reduction is a focus on relations rather than on sets. Roughly, suppose that

$$\forall x \in \Sigma^*, \quad x \in A \text{ if and only if } \exists y R(x, y).$$

The intuitive concept that "x is an instance of A with k solutions" can be more precisely rendered by "there are k y's such that $R(x, y)$ is true." There are, however, several interesting difficulties in making this rough idea work correctly. In this direction the next definition is the key.

* Received by the editors February 28, 1977, and in revised form July 25, 1977. This work was supported in part by the National Science Foundation under Grant DCR 92373. Part of this work was carried out while the authors were visiting IBM Thomas J. Watson Research Laboratory in June, 1975.

† School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332.

‡ Department of Computer Science, Yale University, New Haven, Connecticut 06520. The work of this author was supported in part by the U.S. Office of Naval Research under Grant N00014-75-C-0752.

DEFINITION. A *combination machine* is a Turing machine with two read-only input tapes with end-markers, the first 2-way and the second 1-way, a 2-way read-write worktape and a 1-way write-only output tape. A combination machine is *logspace* (*polynomial time*) if it always halts and runs within worktape space logarithmic (within time polynomial) in the length of the first input.

As remarked in [5], a set A is in $\mathcal{NL}$, the class of nondeterministic logspace sets ($\mathcal{NP}$, the class of nondeterministic polynomial time sets) if and only if there exist a polynomial p and a relation R such that

$$x \in A \Leftrightarrow (\exists y)[|y| \leqq p(|x|) \wedge R(x, y)],$$

and R is computable by a logspace (polynomial time) combination machine.

Now let R and S be arbitrary binary relations, and let r and s be polynomials. Then we will define reducibility $\leqq^{\mathcal{L}}$ ($\leqq^{\mathcal{P}}$) between (R, r) and (S, s) as follows:

$(R, r) \leqq^{\mathcal{L}} (\leqq^{\mathcal{P}}) (S, s)$ provided there exists an f and g such that

1. f is a function computable by a deterministic logspace (polynomial time) transducer 2-way on its input;
2. g is a function computable by a logspace (polynomial time) combination machine;
3. $\forall x, y \in \Sigma^*$,

$$[R(x, y) \wedge |y| \leqq r(|x|)] \quad \text{implies} \quad [S(f(x), g(x, y)) \wedge |g(x, y)| \leqq s(|f(x)|)];$$

4. g is 1–1 in the sense that $\forall x, y_1, y_2 \in \Sigma^*$,

$$[R(x, y_1) \wedge R(x, y_2) \wedge |y_1| \leqq r(|x|) \wedge |y_2| \leqq r(|x|) \wedge g(x, y_1) = g(x, y_2)] \quad \text{implies} \quad y_1 = y_2;$$

5. g is onto in the sense that $\forall x, z \in \Sigma^*$,

$$[S(f(x), z) \wedge |z| \leqq s(|f(x)|)] \quad \text{implies} \quad [\exists y |y| \leqq r(|x|) \wedge R(x, y) \wedge g(x, y) = z].$$

This definition, while at first appearing to be complex, is actually a natural extension of the usual one. In order to see this, observe that the usual definition states that A is logspace (polynomial time) reducible to B for A and B which are expressed by

$$A = \{x | \exists y |y| \leqq r(|x|) \wedge R(x, y)\}$$

and

$$B = \{x | \exists y |y| \leqq s(|x|) \wedge S(x, y)\}$$

provided

$$[\exists y |y| \leqq r(|x|) \wedge R(x, y)] \Leftrightarrow [\exists y |y| \leqq s(|f(x)|) \wedge S(f(x), y)],$$

where f is some logspace (polynomial time) transduction. Our main idea is to strengthen this condition by putting into 1–1 correspondence specific witnesses to the two existential quantifiers. Note that according to our definitions, if $(R, r) \leqq^{\mathcal{L}} (\leqq^{\mathcal{P}}) (S, s)$ via f and g, then for any x,

$$|\{y | |y| \leqq r(|x|) \wedge R(x, y)\}| = |\{y | |y| \leqq s(|f(x)|) \wedge S(f(x), y)\}|.$$

PROPOSITION 1. $\leqq^{\mathcal{L}}$ ($\leqq^{\mathcal{P}}$) *is transitive.*

Proof. We consider $\leqq^{\mathscr{L}}$. We need only show that if $(R, r) \leqq^{\mathscr{L}} (S, s)$ via f, g and $(S, s) \leqq^{\mathscr{L}} (T, t)$ via f', g' then $(R, r) \leqq^{\mathscr{L}} (T, t)$ via

$$f'' = \lambda x[f'(f(x))] \quad \text{and} \quad g'' = \lambda x, y[g'(f(x), g(x, y))].$$

First, f'' is a logspace transduction; this follows by standard arguments [8]. We show that g'' is computable by a logspace combination machine as follows:

Simulate g'. The only difficulty is in obtaining the appropriate bits of the inputs to g' as needed. The first input is easy: in order to compute the ith bit of $f(x)$ we need only simulate $f(x)$ from the beginning until it outputs the ith bit. This works since x is 2-way; a counter must be maintained for i. The second input is more difficult. In order to compute the ith bit of $g(x, y)$ we simply simulate $g(x, y)$ until it outputs the ith bit. The key is that g' asks for these bits in the same order as produced in the computation of $g(x, y)$; thus, in the simulation of $g(x, y)$ it suffices to have y on a 1-way tape. No counter need be maintained for i in this case.

Now verification of properties 3–5 is reasonably straightforward. Transitivity of $\leqq^{\mathscr{P}}$ is obvious.

DEFINITION. (S, s) is *$\mathscr{L}$-complete* (*$\mathscr{P}$-complete*) if S is a relation computable by a logspace (polynomial time) combination machine, s is a polynomial, and for all (R, r), where R is computable by a logspace (polynomial time) combination machine and r is a polynomial,

$$(R, r) \leqq^{\mathscr{L}} (\leqq^{\mathscr{P}}) (S, s).$$

It is not difficult to show that if (S, s) is $\mathscr{L}$-complete ($\mathscr{P}$-complete), then

$\{x \mid \exists y |y| \leqq s(|x|) \wedge S(x, y)\}$ is complete in $\mathscr{NL}$ ($\mathscr{NP}$) according to the more usual definitions.

Let SAT(x, y) be "x is a conjunctive normal form Boolean formula and y is an assignment of true or false to the variables of x making x true" [3]. SAT is clearly computable by a polynomial time combination machine. Let $s(n) = n$.

PROPOSITION 2. *(SAT, s) is $\mathscr{P}$-complete.*

Proof. Let R be a relation computable by a polynomial time combination machine M and let r be a polynomial. We will construct a nondeterministic Turing machine M' from M and r as follows:

M' on input x simulates M on inputs of the form (x, y) by guessing bits of y (including guessing when the end of y has been reached) when M needs them. M' also keeps a counter and if M tries to read more than $r(|x|)$ bits of y, then M' will reject the input x for this series of guesses. If M rejects (x, y), then M' will also reject x on the corresponding computation path. If M accepts then M' will continue to guess bits of y, and it accepts when it guesses that the end has been reached; again if it tries to exceed $r(|x|)$ total bits of y then it rejects.

We also require that every guess made by M' be actually "written down" when made (at least temporarily) so that distinct values of y satisfying $R(x, y)$ and $|y| \leqq r(|x|)$ will cause M' to follow distinct computation paths. Clearly there is a polynomial q such that M' on any input x and any computation path, halts in at most $q(|x|)$ steps; by standard techniques, we can actually assume that any computation of M' that accepts x halts in exactly $q(|x|)$ steps.

Now M' is coded into (SAT, s) as in Simon [7]. In order to show $(R, r) \leqq$ (SAT, s) we obtain the required mappings as follows:

1. $f(x)$ is the Boolean formula obtained by coding the computations of M' on the input x;

2. $g(x, y)$ is anything we like if $|y| > r(|x|)$; otherwise, let M' operate on input x and guess precisely the input y when simulating the machine M; then let $g(x, y)$ be the Boolean assignment to the variables of the formula $f(x)$ which describes this computation.

We may now assert that these functions have the required properties. Properties 1 and 2 of the definition of $\leqq^{\mathscr{P}}$ are clear. For 3, we must show that

$$[R(x, y) \wedge |y| \leqq r(|x|)] \quad \text{implies} \quad [S(f(x), g(x, y)) \wedge |g(x, y)| \leqq s(|f(x)|)].$$

But this should be clear from the construction of M', $f(x)$ and $g(x, y)$. To see 4, suppose that $R(x, y_1)$, $R(x, y_2)$, $|y_1| \leqq r(|x|)$, $|y_2| \leqq r(|x|)$, and $g(x, y_1) = g(x, y_2)$ are all true. Since M' writes down all its guesses, it must be the case that $y_1 = y_2$.

Finally we will show property 5. Suppose that $S(f(x), z)$ and $|z| \leqq s(|f(x)|)$ are true. Since z encodes the guesses of M', there must be an input y (since M' only guesses the second input) such that $R(x, y)$ with $|y| \leqq r(|x|)$ and by construction $g(x, y) = z$. Thus g is onto, and hence (SAT, s) is $\mathscr{P}$-complete. □

We could, of course, extend Proposition 2 to a collection of other polynomial-computable relations. Rather than pursuing such results, we turn our attention to relations computable by logspace combination machines. Let GAP (x, y) be "x encodes an acyclic directed graph (Savitch [6]) with the property that the total order of nodes induced by the node numbering is a topological ordering of the partial ordering induced by the edge directions, and y encodes a directed path from start to finish." Again let $s(n) = n$. Clearly, GAP is computable by a logspace combination machine.

PROPOSITION 3. (GAP, s) *is* $\mathscr{L}$*-complete.*

Proof. Since this is almost identical to the proof of Proposition 2 we will only sketch it. Let R be a relation computable by a logspace combination machine M, and let r be a polynomial. Define M' as follows:

M' on input x simulates M on inputs of the form (x, y) by guessing bits of y when M needs them. (Note since M is 1-way on this input M' does *not* have to remember all of y). M' then operates just as in Proposition 2.

Since guesses need only be written down temporarily, there is no difficulty with the space bound. Clearly M' will be a nondeterministic logspace Turing machine which we can assume halts for any computation in exactly $q(|x|)$ steps, for some polynomial q.

M' is encoded into GAP as in [6]. Then f and g are obtained as follows:

1) $f(x)$ is the graph obtained by encoding of M' on x.
2) $g(x, y)$ is anything we like if $|y| > r(|x|)$. Otherwise, let M' on input x with guesses y yield the path $g(x, y)$ through the graph $f(x)$.

Then $(R, r) \leqq^{\mathscr{L}} (\text{SAT}, s)$ via this f and g. A key again is that the computation of M' encodes the actual y it guesses so that $g(x, y)$ will be 1–1. □

We note that Proposition 3 is also extendible to a variety of other logspace-computable relations.

3. Size of solution sets for relations. We consider $\mathscr{L}$-complete ($\mathscr{P}$-complete) (R, r). We wish to give a complexity classification for

$$(R, r)_k = \{x \mid \exists \text{ exactly } k \text{ values of } y,\ |y| \leqq r(|x|) \wedge R(x, y)\}$$

for various values of k. We will first examine specific problems and then we will use the results of §2 to obtain generalizations. In the following, we let $\underset{\mathscr{M}}{\leqq}^{\mathscr{P}}$, $\underset{\mathscr{M}}{\leqq}^{\mathscr{L}}$, $\underset{\mathscr{T}}{\leqq}^{\mathscr{P}}$ and $\underset{\mathscr{T}}{\leqq}^{\mathscr{L}}$ represent the reducibilities used by Karp [3], Jones and Laaser [2], Cook [1] and Ladner and Lynch [4], respectively. For convenience, we let SAT1 (x, y) be "x is an arbitrary form Boolean formula and y is an assignment of true or false to the variables of x making x true." Let $s(n) = n$ as before. Then it is clear that (SAT1, s) is $\mathscr{P}$-complete.

PROPOSITION 4. *For any fixed* $k \geqq 1$,

(a) $$(\mathrm{GAP}, s)_k \underset{\mathcal{M}}{\equiv^{\mathcal{L}}} (\mathrm{GAP}, s)_1,$$

and

(b) $$(\mathrm{SAT1}, s)_k \underset{\mathcal{M}}{\equiv^{\mathcal{P}}} (\mathrm{SAT1}, s)_1.$$

Proof. (a) We first show

$$(\mathrm{GAP}, s)_k \underset{\mathcal{M}}{\leqq^{\mathcal{L}}} (\mathrm{GAP}, s)_1.$$

Assume $k \geqq 2$. If x is not an acyclic directed graph satisfying the given consistency condition, then let $f(x) = x$. Otherwise, let $p\!\!\!/: N^k \times \{0, 1\}^{k-1}$ (where N is the set of natural numbers) be a natural 1–1 $(2k-1)$-tupling function with the property that

$$\left[\bigwedge_{i=1}^{k} (x_i \leqq y_i) \wedge \bigvee_{i=1}^{k} (x_i < y_i)\right]$$
$$\text{implies} \quad p\!\!\!/(x_1, \cdots, x_k, a_1, \cdots, a_{k-1}) < p\!\!\!/(y_1, \cdots, y_k, b_1, \cdots, b_{k-1}).$$

The *start node* of $f(x)$ is

$$p\!\!\!/\ (\underbrace{n_s, \cdots, n_s}_{k}, \underbrace{0, \cdots, 0}_{k-1}),$$

where n_s is the (number of the) start node of x. The *goal node* of $f(x)$ is

$$p\!\!\!/\ (\underbrace{n_G, \cdots, n_G}_{k}, \underbrace{1, \cdots, 1}_{k-1}),$$

where n_G is the (number of the) goal node of x. The edges of graph $f(x)$ will be defined by tupling together edges of x as follows:

$$(p\!\!\!/(x_1, \cdots, x_k, a_1, \cdots, a_{k-1}),\ p\!\!\!/(y_1, \cdots, y_k, b_1, \cdots, b_{k-1}))$$

will be an edge of $f(x)$ exactly if:

1) $(x_1, \cdots, x_k, a_1, \cdots, a_{k-1}) \neq (n_G, \cdots, n_G, 1, \cdots, 1)$, and
2) for all i, $1 \leqq i \leqq k-1$,
   (a) either (x_i, y_i) is an edge of x, or $x_i = y_i = n_G$, and
   (b) one of (b1)–(b3) holds:
      (b1) $a_i = 0$ and $x_i = x_{i+1} \neq n_G$ and $b_i = 0$ and $y_i = y_{i+1} \neq n_G$,
      (b2) $a_i = 0$ and $x_i = x_{i+1} \neq n_G$ and $b_i = 1$ and $y_i < y_{i+1}$,
      (b3) $a_i = 1$ and $b_i = 1$.

The reader may verify that $f(x)$ is an acyclic directed graph satisfying the given consistency condition, and that $f(x)$ is computable from x in logspace. Intuitively, a single path through $f(x)$ corresponds to parallel simulation of k distinct paths through x, where a flag is changed from 0 to 1 to indicate that two "adjacent" paths have just been discovered to diverge with the path at the left preceding the path at the right lexicographically. Padding is used for shorter paths in x. With this intuition, the reader should be able to verify that x has exactly k solutions iff $f(x)$ has exactly 1 solution.

We now must show $(\mathrm{GAP}, s)_1 \underset{\mathcal{M}}{\leqq^{\mathcal{L}}} (\mathrm{GAP}, s)_k$. But this is straightforward by a construction which adds $k-1$ disjoint "dummy paths" to a graph of the appropriate type. □

(b) We first show

$$(\mathrm{SAT1}, s)_k \underset{\mathcal{M}}{\leqq^{\mathcal{P}}} (\mathrm{SAT1}, s)_1.$$

If x is not a Boolean formula, define $f(x)=x$. Otherwise, assume x is of the form $a(x_1, \cdots, x_n)$. Let $f(x)$ be the formula obtained by selecting new disjoint sets of variables $\{x_{i1}, \cdots, x_{in}\}_{i=1}^{k}$ and expanding into the appropriate form the expression:

$$[a(x_{11}, \cdots, x_{1n}) \wedge a(x_{21}, \cdots, x_{2n}) \wedge \cdots \wedge a(x_{k1}, \cdots, x_{kn}) \wedge (x_{11}x_{12}\cdots x_{1n} < x_{21}x_{22}\cdots x_{2n} < \cdots < x_{k1}x_{k2}\cdots x_{kn})].$$

The last line of inequalities is intended to indicate lexicographic ordering of the given strings. f is computable in polynomial time, and x has exactly k solutions iff $f(x)$ has exactly 1 solution.

Next we show

$$(\mathrm{SAT1}, s)_1 \underset{\mathscr{M}}{\leqq}^{\mathscr{P}} (\mathrm{SAT1}, s)_k.$$

If $k=1$ there is nothing to prove, so assume $k \geqq 2$. If x is not a Boolean formula, define $f(x)=x$. Otherwise, assume x is of the form $a(x_1, \cdots, x_n)$. Let $f(x)$ be the formula

$$\left[a(x_1, \cdots, x_n) \wedge \bigwedge_{i=1}^{k-1} \overline{x_{n+1}}\right] \vee \bigvee_{i=1}^{k-1} \left[\left(\bigwedge_{j=1}^{n+i} x_j\right) \wedge \left(\bigwedge_{j=n+i+1}^{n+k-1} \overline{x_j}\right)\right].$$

f essentially adds $k-1$ dummy solutions to x, and so x has exactly 1 solution iff $f(x)$ has exactly k solutions. □

We now note that a result similar to Proposition 4 must hold for *all* complete (R, r). That is, addition of dummy solutions and collapsing of several solutions to one are constructions which work for all complete problems. In fact, all such problems must be equivalent to each other:

PROPOSITION 5. *For any fixed* $k \geqq 1$,

(a) *if* (R, r) *is* $\mathscr{L}$*-complete, then*

$$(R, r)_k \underset{\mathscr{M}}{\equiv}^{\mathscr{L}} (\mathrm{GAP}, s)_1,$$

and

(b) *if* (R, r) *is* $\mathscr{P}$*-complete, then*

$$(R, r)_k \underset{\mathscr{M}}{\equiv}^{\mathscr{P}} (\mathrm{SAT1}, s)_1.$$

Proof. By Proposition 4 and the fact that our reducibilities $\leqq^{\mathscr{L}}$ and $\leqq^{\mathscr{P}}$ preserve cardinality of solution sets (as noted immediately prior to Proposition 1). □

Now that we know that all size problems lie in a common complexity class, we would like to be able to say more about the location of this class. The only such information we have so far arises as a result of the following proposition. Here, let $G(S)$ be $\mathscr{NL}$-complete ($\mathscr{NP}$-complete) in the usual sense.

PROPOSITION 6. (a) *If*

$$\bar{G} \underset{\mathscr{M}}{\leqq}^{\mathscr{L}} A \underset{\mathscr{T}}{\leqq}^{\mathscr{L}} G,$$

then $A \in \mathscr{NL}$ *iff* $\mathscr{NL}$ *is closed under complement, and* $A \in \mathscr{L}$ *iff* $\mathscr{L} = \mathscr{NL}$, *and*

(b) *If*

$$\bar{S} \underset{\mathscr{M}}{\leqq}^{\mathscr{P}} A \underset{\mathscr{T}}{\leqq}^{\mathscr{P}} S,$$

then $A \in \mathscr{NP}$ *iff* $\mathscr{NP}$ *is closed under complement, and* $A \in \mathscr{P}$ *iff* $\mathscr{P} = \mathscr{NP}$.

Proof. The arguments are all standard Turing machine constructions, of the type found in [2] or [4], for example. □

Finally, we can conclude:

PROPOSITION 7. *For any fixed* $k \geqq 1$,

(a) *if* (R, r) *is* $\mathcal{L}$*-complete, then* $(R, r)_k \in \mathcal{NL}$ *iff* $\mathcal{NL}$ *is closed under complement, and* $(R, r)_k \in \mathcal{L}$ *iff* $\mathcal{L} = \mathcal{NL}$, *and*

(b) *if* (R, r) *is* $\mathcal{P}$*-complete, then* $(R, r)_k \in \mathcal{NP}$ *iff* $\mathcal{NP}$ *is closed under complement, and* $(R, r)_k \in \mathcal{P}$ *iff* $\mathcal{P} = \mathcal{NP}$.

Proof. (a) It suffices to show

$$\bar{G} \underset{\mathcal{M}}{\leqq}^{\mathcal{L}} (\mathrm{GAP}, s)_1 \underset{\mathcal{T}}{\leqq}^{\mathcal{L}} G,$$

where $G = \{x \mid \exists y |y| \leqq s(|x|)$ and GAP $(x, y)\}$.

The first reduction follows by adding a single "dummy path" to a graph.

To see the second reduction, note that $(\mathrm{GAP}, s)_k = A \cap \bar{B}$, where

$$A = \{x \mid \exists \text{ at least } k \text{ values of } y,\ |y| \leqq s(|x|) \wedge \mathrm{GAP}\ (x, y)\},$$

$$B = \{x \mid \exists \text{ at least } k+1 \text{ values of } y,\ |y| \leqq s(|x|) \wedge \mathrm{GAP}\ (x, y)\}.$$

Clearly A, B are both in $\mathcal{NL}$, so that

$$A \underset{\mathcal{M}}{\leqq}^{\mathcal{L}} G \quad \text{and} \quad B \underset{\mathcal{M}}{\leqq}^{\mathcal{L}} G.$$

Then a Turing machine with a G-oracle may easily be constructed to decide membership in $(\mathrm{GAP}, s)_1$.

(b) It suffices to show

$$\bar{S} \underset{\mathcal{M}}{\leqq}^{\mathcal{P}} (\mathrm{SAT1}, s)_1 \underset{\mathcal{T}}{\leqq}^{\mathcal{P}} S.$$

To see the first reduction, we use a special case of construction for Proposition 4(b): if x is not a Boolean formula, define $f(x) = x$. Otherwise, if x is the form $a(x_1, \cdots, x_n)$ let $f(x)$ be the formula $[a(x_1, \cdots, x_n) \wedge \overline{x_{n+1}}] \vee [x_1 \wedge \cdots \wedge x_n \wedge x_{n+1}]$. x has no solutions iff $f(x)$ has exactly one solution.

The second reduction follows from the same argument as in (a). □

Of course, the given complexity classification is still very incomplete; further work remains to be done. For example, is $(\mathrm{SAT}, s)_1$ in $\mathcal{NP}$?

We would expect that there are other interesting properties of solution sets which are preserved by strong reducibilities such as ours. Finding the appropriate strength reducibilities needed to preserve constructions such as those used for approximation, or for finding a *particular* solution when existence is known, seems to be an interesting area for further study.

REFERENCES

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

[2] N. JONES AND W. LAASER, *Complete problems for deterministic polynomial time*, Proceedings of Sixth Annual ACM Symposium on Theory of Computing, April–May 1974, pp. 40–46.

[3] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[4] R. LADNER AND N. A. LYNCH, *Relativization of questions about log space computability*, Math. Systems Theory, 10 (1976), pp. 19–32.

[5] R. LIPTON AND N. A. LYNCH, *A quantifier characterization for nondeterministic log space*, SIGACT News, 7 (1975), no. 4, pp. 24–26.

[6] W. SAVITCH, *Relations between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 14 (1970), pp. 177–192.

[7] J. SIMON, *On some central problems in computational complexity*, TR 75-224, Cornell University, Ithaca, NY, 1975.

[8] L. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time: Preliminary report*, 5th Annual ACM Symposium on Theory of Computing, 1973, pp. 1–9.

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

AN ALGORITHM FOR TRANSITIVE CLOSURE WITH LINEAR EXPECTED TIME*

C. P. SCHNORR†

Abstract. An algorithm for transitive closure is described with expected time $O(n+m^*)$ where n is the number of nodes and m^* is the expected number of edges in the transitive closure.

Key words. transitive closure, average time, random access machines

1. Introduction. Algorithms that construct the transitive closure of a given directed graph have obtained considerable attention. Roy [5] and later on Warshall [8] proposed an algorithm that runs in $O(n^3)$ steps, where n is the number of nodes. In 1970 four Russians published an algorithm with time bound $O(n^3/\log n)$. M. Fischer and A. R. Meyer [4] applied Strassen's fast matrix multiplication and obtained an algorithm for transitive closure with time bound $O(n^{2.81})$.

Recently Bloniarz, Fischer and Meyer [2] proposed an algorithm for transitive closure with average time $O(n^2 \log n)$. We propose an improved algorithm with expected time $O(n+m^*)$ where m^* is the expected number of edges in the transitive closure. Moreover the event that the running time exceeds cn^2 is less than 2^{-n} for some fixed $c>0$ and all n.

These bounds hold for all those graph distributions where the probability of a graph only depends on its number of edges and on its number of nodes. This, for instance, is true whenever the n^2 possible edges of the random graph with n nodes (i.e. whether there is an edge $i \to j$ or not) are identical and independent random variables. The above class of graph distributions is only slightly smaller than the one which is used by Bloniarz, Fischer and Meyer. They require that the probability of a graph only depends on the set of outdegrees and the number of nodes. This latter class of graph distributions is not preserved under the transformation which associates to each graph its edge reversal (the edge $i \to j$ is replaced by $j \to i$ by this transformation). Requiring that the class of graph distributions is closed with respect to the transformation of edge reversal leads to our smaller class of distributions.

2. The algorithm. Our time analysis is valid for an implementation of the algorithm on the type of storage modification machines which has been proposed by Schönhage [7]. This model of machine is linear time equivalent to the RAM machine with unit costs and addition/substraction by 1; see Schnorr [6] for the equivalence proof.

We suppose that the input graph has node set $\{1, 2, \cdots, n\}$ and is given by its adjacency lists $L_i = \{j | \exists$ edge from i to $j\}$ for $i = 1, \cdots, n$. We assume that there are no repetitions in the lists L_i, which ensures that the input is not too big. No particular ordering of the input lists L_i is assumed; these lists will be linearly ordered in Stage 0 of the algorithm. Let $m = \sum_{i=1}^{n} \|L_i\|$ be the number of edges in the given graph. Let $L_i^r = \{j | \exists$ edge from j to $i\}$ $i = 1, \cdots, n$, be the adjacency lists of the edge reserved graph. j is called a *successor* of i if there is a path from i to j. In this case, i is called a *predecessor* of j.

* Received by the editors September 20, 1976, and in final revised form September 26, 1977.
† Fachbereich Mathematik, Universität Frankfurt, 6 Frankfurt am Main, West Germany.

Informal description of the algorithm. The algorithm consists of the following stages:

Stage 0. The lists L_j^r of the edge reserved graph are constructed in linear order by inserting i into L_j^r for all $j \in L_i$ in the order of succession $i = 1, 2, \cdots n$. By a second application of this process we obtain the lists $(L_i^r)^r$ which are linearly ordered permutations of the lists L_i. Substitute $(L_i^r)^r$ for L_i.

Stage 1. Associate in a breadth first search manner to each node i a list S_i of successors such that either (i) or (ii) holds:

(i) $\|S_i\| < \lfloor n/2 \rfloor + 1$ and S_i is the complete list of successors of i.

(ii) $\|S_i\| = \lfloor n/2 \rfloor + 1$.

Stage 2. Apply Stage 1 to the edge reversed graph, i.e. associate to each node i a list P_i of predecessors such that either (i) or (ii) holds with P_i substituted for S_i.

Stage 3. For $i = 1, 2, \cdots, n$ form the adjacency lists

$$L_i^* = S_i \cup \{j | i \in P_j\} \cup \{j | \|S_i\| = \|P_j\| = \lfloor n/2 \rfloor + 1\}$$

of the transitive closure as unions of three lists each.

Stage 4. Permute L_i^* into the linearly ordered lists $(L_i^*)^{rr}$ using two applications of the algorithm for forming the edge reserved graph (compare Stage 0). Replace L_i^* by $(L_i^*)^{rr}$ and remove multiplicities by one pass over the ordered lists L_i^*.

Clearly Stages 0, 3 and 4 can be implemented on a storage modification machine as to run in linear time on their respective input data. Stage 2 is symmetric to Stage 1. Therefore it will be sufficient for the time analysis of the algorithm to give a detailed description of Stage 1 which constructs S_i, $i = 1, \cdots, n$, in a breadth first search manner. We use two auxiliary storages, *queue* and *stack*. The lists Q_j, $j = 1, \cdots, n$, are supposed to be initially empty.

Stage 1 of the algorithm.

```
for i = 1 step 1 until n do
begin S_i := queue := {i}, stack := ∅
      counter := 1, mark i
      while queue ≠ ∅ and counter < ⌊n/2⌋ + 1 do
            j := top node of queue
            remove j from top of queue
            push j onto the top of stack
            while counter < ⌊n/2⌋ + 1 and L_j ≠ ∅ do
                  a := first node of L_j
                  L_j := L_j − {a}, Q_j := Q_j ∪ {a}
                  if a is not yet marked then
                  [push a to the bottom of queue,
                  mark a, S_i := S_i ∪ {a},
                  counter := counter + 1]
            end
      end
      for all j on stack
      [unmark j, L_j := L_j ∪ Q_j, Q_j := ∅]
end
```

Comments on the algorithm. Let i be fixed. Then the algorithm constructs S_i in a breadth first search since we push a at the bottom of *queue* when a is visited for the

first time (while in a depth first search a must be pushed to the top of *queue*) and within the inner **while**-loop, the adjacency list L_j of the previous top node j of *queue* is exhausted. Before examining L_j, j is removed from *queue* and pushed onto *stack*.

3. Correctness and analysis of the algorithm.

THEOREM 1. *The complete algorithm correctly computes the transitive closure.*

Proof. The outer **while**-loop in Stage 1 either finishes because *queue* $= \varnothing$ or because *counter* $= \lfloor n/2 \rfloor + 1$. If it finishes because *queue* $= \varnothing$, then S_i is the complete list of successors of node i. If the **while**-loop finishes with *queue* $\neq \varnothing$, then *counter* $= \lfloor n/2 \rfloor + 1$, and this means $\|S_i\| = \lfloor n/2 \rfloor + 1$. This proves that Stage 1 associates to each node i a set of successors S_i such that either (i) or (ii) above hold. Stage 2 of the algorithm works exactly as Stage 1 with L_i replaced by L_i^r and S_i replaced by P_i.

It remains to prove that for each edge $i \to j$, $i \to j$ is in the transitive closure of the input graph if and only if

$$[j \in S_i \text{ or } i \in P_j \text{ or } \|S_i\| = \|P_j\| = \lfloor n/2 \rfloor + 1]$$

"$\Leftarrow$": Obviously, if $j \in S_i$ or $i \in P_j$, then the edge $i \to j$ is in the transitive closure. If $\|S_i\| = \|P_j\| = \lfloor n/2 \rfloor + 1$ then $S_i \cap P_j$ cannot be empty since this would imply $\|S_i \cup P_j\| > n$. However, $S_i \cap P_j \neq \varnothing$ implies that $i \to j$ is in the transitive closure.

"$\Rightarrow$": If $j \to i$ is in the transitive closure and either $\|S_i\| < \lfloor n/2 \rfloor + 1$ or $\|P_j\| < \lfloor n/2 \rfloor + 1$ then the correctness conditions (i), (ii) of Stages 1 and 2 of the algorithm imply that S_i is the complete set of successors of i provided $\|S_i\| < \lfloor n/2 \rfloor + 1$ and that P_j is the complete set of predecessors of j provided $\|P_j\| < \lfloor n/2 \rfloor + 1$. □

THEOREM 2. *Suppose that the probability of a graph is a function of its number of nodes n and edges m. Then the expected running time of the algorithm is* $O(n + m^*)$, *where* m^* *is the expected number of edges in the transitive closure.*

Proof. Observe that we do not suppose that the input lists L_j are permuted at random. This would be unnatural since usually these lists are ordered in some way. However, the lists L_i will be in linear order at the end of Stage 0 and the following time analysis of Stage 1 will use this fact. Since we do not suppose that the lists L_j are permuted at random and since the first element $a \in L_j$ in the inner **while**-loop of Stage 1 of the algorithm is chosen deterministically and not at random, it is clear that the sequence $(a_t | t = 1, 2, \cdots)$ of nodes $a_t \in \{1, \cdots, n\}$ which are examined in the inner **while**-loop during the construction of S_i is not a sequence of independent random variables. Here a_t is the node which is examined within the tth pass through the inner **while**-loop during the construction of S_i for some fixed i.

A main point is that the breadth first search structure of our algorithm ensures some global random properties of the sequence $(a_t | t = 1, 2, \cdots)$. Let $L_{\sigma(\nu)} = (a_t | h(\nu) < t \leqq h(\nu + 1))$ be the νth adjacency list under examination within the construction of S_i. Here h is a function that depends on the input lists L_j and the start vertex i. Let $h(\nu + 1)$ be defined if and only if the νth adjacency list $L_{\sigma(\nu)}$ is exhausted during the construction of S_i. Let $\Delta L_{\sigma(\nu)} = \{a_t | h(\nu) < t \leqq h(\nu + 1)\}$ be the set of elements in the list $L_{\sigma(\nu)}$ and let $S_i(t) = \{a_1, \cdots, a_t\}$.

In the following average time analysis of Stage 1 let n and m be arbitrarily fixed. Under this condition we have Prob $(j \in \Delta L_i) = m/n^2$ for all nodes j, i. In particular all n-tuples of ordered lists $(L_1, \cdots, L_n)$ with $m = \sum_{i=1}^{n} \|\Delta L_i\|$ are equally probable. Since no list L_j is examined twice during the construction of S_i, we have

FACT 1. (i) *The sets* $\Delta L_{\sigma(\nu)}$, $\nu = 1, 2, \cdots$, *are (mutually) independent;*

(ii) $\Delta L_{\sigma(\nu)}$ *is uniformly distributed, i.e., for all* $A \subset \{1, \cdots, n\}$,

$$\text{Prob}\,(\Delta L_{\sigma(\nu)} = A) = (m/n^2)^{\|A\|}(1 - m/n^2)^{n - \|A\|}.$$

The independence of the sets $\Delta L_{\sigma(\nu)}$ will serve as a substitute for the independence of the sequence $(a_t|t=1,2,\cdots)$. We like to bound the expected value of $\min\{t|\,\|S_i(t)\|>k\}$ with $k\leqq\lfloor n/2\rfloor$. Observe that $\min\{t|\,\|S_i(t)\|>k\}$ up to a constant factor bounds the number of steps that are carried out in the construction of S_i until the $(k+1)$st element of S_i has been found. We shall compare $(a_t|t=1,2,\cdots)$ to a sequence of independent random variables.

LEMMA 1. *Let $A\subset\{1,\cdots,n\}$ with $\|A\|=k$ be fixed and let $(\bar{a}_t|t=1,2,\cdots)$ be independent random variables which are uniformly distributed over $\{1,2,\cdots,n\}$. Then* $\min\{t|\bar{a}_t\notin A\}$ *has expected value* $1+k/(n-k)$.

Proof. $\text{Prob}\,[\bar{a}_1,\cdots,\bar{a}_r\in A \text{ and } \bar{a}_{r+1}\notin A]=(k/n)^r(1-k/n)$. Hence the expected value of $\min\{t|\bar{a}_t\notin A\}$ is

$$1+\sum_{r=1}^{\infty} r(k/n)^r(1-k/n).$$

Obviously $\sum_{r=0}^{\infty} r\alpha^{r-1}=(1/(1-\alpha))'=1/(1-\alpha)^2$. Hence $\sum_{r=1}^{\infty} r(k/n)^r(1-k/n)=(k/n)/(1-k/n)=k/(n-k)$. This proves Lemma 1. □

LEMMA 2. *Let $(\bar{a}_t|t=1,2,\cdots)$ be independent random variables which are uniformly distributed over $\{1,2,\cdots,n\}$. Then for $k\leqq n/2$,* $\min\{t|\,\|\{\bar{a}_1,\bar{a}_2,\cdots,\bar{a}_t\}\|>k\}$ *has expected value* $\leqq 1.5(k+1)$.

Proof. According to Lemma 1 the expected value of $\min\{t|\,\|\{\bar{a}_1,\bar{a}_2,\cdots,\bar{a}_t\}\|>k\}$ is bounded by

$$\begin{aligned}\sum_{\nu=0}^{k}(1+\nu/(n-\nu))&\leqq k+1+\sum_{\nu=1}^{k}\nu/(n/2)\\&\leqq k+1+k(k+1)/n\\&\leqq 1.5k+1.5.\ \square\end{aligned}$$

Let the sequence of random variables $(a'_t|t=1,2,\cdots)$ be obtained from $(a_t|t=1,2,\cdots)$ by permuting the segments $(a_t|h(\nu)<t\leqq h(\nu+1))$ at random for $\nu=1,2,\cdots$. $(a'_t|t=1,2,\cdots)$ is "almost" a sequence of independent random variables. The unique dependence between the variables a'_t is that the variables within each segment $(a'_t|h(\nu)<t\leqq h(\nu+1))$ take pairwise different values. Let $S'_i(t)=\{a'_1,\cdots,a'_t\}$.

LEMMA 3. *For $k\leqq n/2$ the expected value of* $\min\{t|\,\|S'_i(t)\|>k\}$ *is* $\leqq 1.5(k+1)$.

Proof. Since the sets $\Delta L_{\sigma(\nu)}$ are independent and $\Delta L_{\sigma(\nu)}$ is uniformly distributed according to Fact 1(ii), it follows from the construction of $(a'_t|t=1,2,\cdots)$ that this sequence is distributed as a sequence of independent random variables $(\bar{a}_t|t=1,2,\cdots)$ under the condition that the variables within each segment $(\bar{a}_t|h(\nu)<t\leqq h(\nu+1))$ take pairwise different values for $\nu=1,2,\cdots$. However, the $1.5k+1.5$ bound of Lemma 2 holds a fortiori under the condition that the variables within each segment $(\bar{a}_t|h(\nu)<t\leqq h(\nu+1))$ take pairwise different values, since it should be obvious that the expected value in Lemma 2 must increase under the converse condition. □

LEMMA 4. $(\|S_i(t)\|\ t=1,2,\cdots)$ *and* $(\|S'_i(t)\|\ t=1,2,\cdots)$ *are equally distributed sequences of random variables.*

Proof. Let $t=h(\nu)+r\leqq h(\nu+1)$. Since $S_i(h(\nu))=S'_i(h(\nu))$ we have

$$S'_i(t)=S_i(h(\nu))\cup A(\tau,r),$$

$$S_i(t)=S_i(h(\nu))\cup A(<,r)$$

where $A(\tau, r)$ ($A(<, r)$, resp.) are the first r elements of $\Delta L_{\sigma(\nu)}$ with respect to a random ordering τ (the linear ordering $<$, resp.) It follows from Fact 1 that $S_i(h(\nu))$ is uniformly distributed. Therefore the distribution of $\|S_i(h(\nu)) \cup A(\tau, r)\|$ does not depend on τ. This proves that $\|S_i'(t)\|$ and $\|S_i(t)\|$ are equally distributed. □

Let s be the expected and let $\|S_i\|$ be the actual size of S_i upon completion of Stage 1. Obviously $s = \sum_{k=1}^{\lfloor n/2 \rfloor + 1} k \cdot \text{Prob}(\|S_i\| = k)$. Let q be the expected number of passes through the inner **while**-loop during the construction of S_i. Then

$$q \leqq \sum_{k=0}^{\lfloor n/2 \rfloor + 1} \text{Prob}(\|S_i\| = k) 1.5k$$

where $1.5k$ according to Lemmas 3 and 4 bounds the expected value of $\min\{t | \|S_i(t)\| > k - 1\}$. Hence $q \leqq 1.5s$. Since $ns \leqq m^*$, this implies

LEMMA 5. *The expected total number of passes through the inner* **while**-loop *during Stage* 1 *and Stage* 2 *of the algorithm is* $\leqq 3m^*$.

Observe that the bound in Lemma 5, which was proved under the condition that n and m are arbitrarily fixed, now holds uniformly for all n and m. It remains to bound the number of steps in those parts of the algorithm which are not affected by the distribution. Stage 0 takes $O(n+m)$ steps with $m = \sum_{i=1}^{n} \|\Delta L_i\|$ and Stage 4 takes $O(n + \sum_{i=1}^{n} \|\Delta L_i^*\|)$ steps. Next we consider Stage 3 of the algorithm, which does the following:

construct $P_i^r := \{j | i \in P_j\}$, $\quad i = 1, \cdots, n$,

$L_i^* := S_i \cup P_i^r$, $\quad i = 1, \cdots, n$,

$S := \{i | \|S_i\| = \lfloor n/2 \rfloor + 1\}$,

$P := \{j | \|P_j\| = \lfloor n/2 \rfloor + 1\}$,

for all $i \in S, j \in P$: $\quad L_i^* := L_i^* \cup \{j\}$.

This yields the adjacency lists L_i^* of the transitive closure and Stage 3 can clearly be done within worst case running time $O(n + \sum_{i=1}^{n} \|\Delta L_i^*\|)$. This finishes the proof of Theorem 1. □

Our algorithm not only has a linear expected running time but for some $c > 0$ the event that the running time exceeds cn^2 has probability less than 2^{-n}. We shall use the following fact from probability theory; see e.g. Erdös and Spencer [3].

LEMMA 6. *Let* $\bar{X}_t$, $t = 1, \cdots, k$, *be independent random variables with* $\text{Prob}(\bar{X}_t = 1) = \text{Prob}(\bar{X}_t = -1) = \frac{1}{2}$. *Then* $\text{Prob}(\sum_{t=1}^{k} \bar{X}_t > \lambda) \leqq e^{-2\lambda^2/k}$.

Let $A_{n,q}$ be the event that the number of passes through the inner **while**-loop within Stage 1 exceeds $n^2 q$. $A_{n,q}$ implies that the number of passes through the inner **while**-loop during the construction of some S_i exceeds nq. Fix i and let $(a_t | t = 1, 2, \cdots)$ be the sequence of nodes which are examined during the construction of S_i. Set $X_t = 1$ if $a_t \in S_i(t-1) = \{a_1, \cdots, a_{t-1}\}$ and $X_t = -1$ if $a_t \notin S_i(t-1)$. Set $X_t = -1$ if less than t nodes are examined during the construction of S_i.

In the same way we associate to $(a_t' | t = 1, 2, \cdots)$ the random variables $(X_t' | t = 1, 2, \cdots)$. Obviously $X_t = 1$ iff $\|S_i(t)\| = \|S_i(t-1)\|$ and $X_t' = 1$ iff $\|S_i'(t)\| = \|S_i'(t-1)\|$. By Lemma 4, $(X_t | t = 1, 2, \cdots)$ and $(X_t' | t = 1, 2, \cdots)$ are equally distributed. Now let $(\bar{X}_t | t = 1, 2, \cdots)$ be a sequence of independent random variables with $\text{Prob}(\bar{X}_t = \mu) = \text{Prob}(X_t = \mu)$ for $\mu = -1, 1$. Then

$$\text{Prob}\left(\sum_{t=1}^{k} X_t > \lambda\right) = \text{Prob}\left(\sum_{t=1}^{k} X_t' > \lambda\right)$$

by Lemma 4, and

$$\text{Prob}\left(\sum_{t=1}^{k} X_t' > \lambda\right) \leqq \text{Prob}\left(\sum_{t=1}^{k} \bar{X}_t > \lambda\right)$$

by the argument that underlies Lemma 3.

Moreover, Lemma 6 applies to $(\bar{X}_t | t = 1, 2, \cdots)$ since the $\bar{X}_t$ are independent and $\text{Prob}(\bar{X}_t = 1) = \text{Prob}(X_t = 1) \leqq \frac{1}{2}$. This altogether yields

$$(*) \qquad \text{Prob}\left(\sum_{t=1}^{k} X_t > \lambda\right) \leqq e^{-2\lambda^2/k}.$$

Now suppose that the number of passes through the inner **while**-loop during the construction of S_i exceeds nq and call this event $A^i_{n,q}$. It follows for at least $nq - n$ different $t \leqq nq$: $a_t \in S_i(t-1)$ and therefore $X_t = 1$. This implies $\sum_{t=1}^{nq} X_t \geqq n(q-2)$. We conclude from (*):

$$\text{Prob}(A^i_{n,q}) \leqq \text{Prob}\left(\sum_{t=1}^{nq} X_t \geqq n(q-2)\right) \leqq e^{-2(q-2)^2 n/q}.$$

Hence

$$\text{Prob}(A_{n,q}) \leqq \sum_{i=1}^{n} \text{Prob}(A^i_{n,q}) \leqq n\, e^{-2(q-2)^2 n/q},$$

and for $q = 4$,

$$\text{Prob}(A_{n,4}) \leqq n\, e^{-2n}.$$

Now consider the total running time of the algorithm which is composed by the running time of the different stages. For some c_1 the running time of Stages 0, 3, 4 is bounded by $c_1 n^2$ in total. The number of passes through the inner **while**-loop of Stage 1 bounds the total running time of Stage 1 up to some constant factor c_2. Since Stage 2 has the same average time behavior as Stage 1 it follows that the event that the algorithm takes more than $(8c_2 + c_1)n^2$ steps has probability less than $n\, e^{-2n}$. This yields a constant $c := 8c_2 + c_1$, which satisfies the following

THEOREM 3. *For some $c > 0$, the event that the algorithm takes more than cn^2 steps has probability less than 2^{-n} for all n.*

Finally we remark that our average time analysis no longer holds if the algorithm is operated in a depth first search manner. In this case the examination of an adjacency list will be successively interrupted. Therefore the sequence $(a_t | t = 1, 2, \cdots)$ of visited nodes during the construction of S_i can no longer be partitioned into independent parts. Unless the input lists L_i are permuted at random, there will be a trend that those nodes which come first in the linear order and which therefore stand at the very beginning of the input lists are visited more frequently than other nodes. Consequently it takes longer to find those nodes which come last in the linear ordering. For example, in the case of the complete graph with n nodes which is given for $m = n^2$, the depth first search algorithm takes at least $\Omega(n^3)$ steps. Hence Theorems 1 and 2 no longer hold.

Acknowledgment. I thank M. Fischer and the referee for their valuable suggestions that greatly helped to improve the paper.

REFERENCES

[1] V. L. ARLAZAROV, E. A. DINIC, M. A. KRONROD AND I. A. FARADZEV, *On economical construction of the transitive closure of an oriented graph*, Soviet Math. Dokl., 11 (1970), pp. 1209–1210.

[2] P. A. BLONIARZ, M. J. FISCHER AND A. R. MEYER, *A note on the average time to compute transitive closures*, Automata Languages and Programming, Michaelson and Milner, eds., Edinburgh University Press, Edinburgh, 1976.

[3] P. ERDÖS AND J. SPENCER, *Probabilistic Methods in Combinatorics.* Academic Press, New York, 1974.

[4] M. J. FISCHER AND A. R. MEYER, *Boolean matrix multiplication and transitive closure*, Twelfth Annual IEEE Symp. on Switching and Automata Theory, (East Lansing, MI, 1971), pp. 129–131.

[5] M. B. ROY, *Transitivité et connexité*, Comptes Rendus, 249 (1959), pp. 216–218.

[6] C. P. SCHNORR, *Rekursive Funktionen und ihre Komplexität*, Teubner, Stuttgart, W. Germany, 1974.

[7] A. SCHÖNHAGE, *Universelle Turingspeicherung*, Automatentheorie und formale Sprachen, Dörr und Hotz, eds., Bibliographisches Institut, Mannheim, W. Germany, 1970.

[8] S. WARSHALL, *A theorem on Boolean matrices*, J. Assoc. Comput. Mach., 9 (1962), pp. 11–12.

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

OBSERVATIONS ON THE COMPLEXITY OF GENERATING QUASI-GRAY CODES*

MICHAEL L. FREDMAN†

Abstract. The purpose of this paper is to develop a decision tree-like model for defining and measuring the on-line complexity of algorithms for generating combinatorial objects. For the purpose of illustration, we consider the problem of generating Gray codes and simple generalizations of Gray codes. We include some results pertaining to the generation of certain special codes and, in addition, we present a trade-off theorem. Our model is information theoretical and we emphasize two aspects of complexity; the amount of information that must be gathered and the amount of data structure update required to generate the successor to a given codeword.

Key words. on-line complexity, Gray codes, decision trees, combinatorial generation

1. Introduction. The purpose of this paper is to develop an information theory oriented model for defining and measuring the complexity of algorithms that generate combinatorial objects. For the purpose of illustration, we will focus our attention on what we shall call quasi-Gray codes. The model will be used to establish theoretical bounds on the efficiency of optimal algorithms that generate various types of quasi-Gray codes. Among other things, we shall demonstrate a provocative trade-off between two complexity measures: the amount of information that must be gathered, and the amount of data structure update that must be performed in order to generate the successor to a given codeword.

We define a quasi-Gray code (QGC) of the dimension n to be a cyclic sequence of n dimensional binary vectors, $v_1, v_2, \cdots, v_L, v_1$, the first L of which are distinct, and satisfying the condition that two consecutive vectors differ in exactly one component. We refer to the number L of distinct vectors as the length of the code. Clearly, $L \leqq 2^n$. The extremal QGC's with $L = 2^n$ are called *Gray codes*. The class of algorithms which we shall use to generate these codes will be called *decision assignment trees* (DAT).

A DAT is a binary tree whose internal nodes are each labeled with the name of a binary variable, and whose leaves each contain a sequence of one or more assignment statements of the form $x \leftarrow 0$ (or 1), where x is a variable name. The execution of a DAT begins with control at the root node. In general, control passes from a given internal node labeled with variable x to its left son if currently $x = 0$, and to the right son if $x = 1$. Upon reaching a leaf, the list of assignment statements contained in that leaf are performed, and the execution terminates. Given that the set of variables appearing in a particular decision assignment tree T is $\{x_1, \cdots, x_m\}$, we let $\bar{x}$ denote the binary vector whose jth component is the current value of x_j, and we refer to $\bar{x}$ as the current value vector. The execution of a DAT typically changes the values of certain of its variables by virtue of its executed assignment statements. If initially $\bar{x} = \omega$, and after an execution of the tree T we have $\bar{x} = \nu$, then we write $\nu = T(\omega)$ to denote the function relationship holding between ω and ν, as defined by T. Given an m dimensional vector ω_1, consider the infinite sequence $\omega_1, \omega_2, \omega_3, \cdots$ such that $\omega_{i+1} = T(\omega_i)$, and let p_n, $n \leqq m$, denote the projection mapping an m dimensional vector ω into the n dimensional vector $p_n(\omega)$, consisting of the first n components of

* Received by the editors November 9, 1976, and in revised form August 2, 1977.

† Department of Applied Physics and Information Science, University of California, San Diego, La Jolla, California 92093. This work was supported in part by the National Science Foundation under Grant MCS-76-08543.

ω. We say that the tree T generates the QGC $v_1, \cdots, v_L$ of dimension n if and only if there is a way to choose ω_1 so that for each $k \geqq 1$, $p_n(\omega_k) = v_j$ when $k \equiv j$ (modulo L).

Our DAT algorithms are intended to model random access machines with small word size, one bit words to be precise. The current value vectors represent the contents of memory. When used to generate a QGC, under our interpretation the first n bits of memory comprise the most recently generated code vector. The remaining $m-n$ bits represent the dynamical state of a data structure employed to facilitate efficient generation. Given a decision assignment tree, we let I denote the maximum number of internal nodes along any path from the root to a leaf, U denote the maximum number of assignment statements in any leaf, and T denote the maximum number of internal nodes plus assignment statements in any path from the root. Observe that $\max(I, U) \leqq T \leqq I + U$. Given that the tree generates a specified QGC, we can, in principle, prune from the tree any redundant tests and any nodes that are never reached in the generation process, so that I, U, and T can be given the following respective worst case on-line complexity interpretations: the amount of information gathered per code vector generation, the amount of data structure update or modification, and the total time per generation. More precisely, one of the assignment statements enumerated by U specifies a change to the QGC vector; the remaining specify changes to the data structure. The quantity I represents the number of memory probes required to determine the appropriate set of assignments.

In the sequel, we will concern ourselves with the following questions concerning inherent complexity. Among all DAT's that generate a specified code, how small can T, I or U be? We will observe that I can generally be traded against U.

A few comments about the DAT as a computational model are in order. Given a QGC, the best value for T from any DAT generating the code provides a lower bound on the RAM complexity of the problem under the logarithmic cost criterion, roughly speaking (see [9, p. 12]). It is not reasonable to expect, however, to be able to always attain these bounds on the RAM in a practical sense. In particular, when defining a class of codes with the code vector dimension being a parameter, the problem of uniformity rears up.

2. Summary of results. A few of our results pertain to the so-called binary reflected Gray codes. For each $n \geqq 1$, there is a reflected Gray code G_n of length $L = 2^n$ consisting of code vectors of dimension n. The code G_1 is given by the vector sequence (0), (1). The code G_{n+1} is described in terms of G_n as follows. If $v_1, v_2, \cdots, v_L$ denotes the vector sequence for the code G_n, the code G_{n+1} of length $2L$ is given by the sequence $(0, v_1), \cdots, (0, v_L), (1, v_L), \cdots, (1, v_1)$.

In the literature, there are algorithms (e.g. see [1]) that generate G_n, which, when formulated as DAT's, satisfy $I, U, T = O(\log n)$ and $I, U, T = \Omega(\log n)$.[1] Since, as will be seen, $I = \Omega(\log(\log L))$ for any DAT which generates a QGC of length L, it follows that these algorithms are near optimal in terms of T and I. But what happens if we try to decrease U? Can we reduce U without significantly increasing I? Yes; we shall construct an algorithm with $I = O(\log n)$ and $U = 7$. If we impose the extreme constraint $U = 1$, however, then we can show that $I = n$ for any such DAT generating the reflected Gray code. Thus we see slight evidence of a trade-off phenomenon between I and U, but one which does not manifest itself until a rather extreme constraint on U is imposed.

[1] Logarithms are taken to the base 2. The expression $f(n) = \Omega(g(n))$ denotes the relationship $g(n) = O(f(n))$.

If we consider that the constraint $U = 1$ altogether eliminates the possibility of utilizing a data structure in generating a QGC, it is not unreasonable to conjecture that $U = 1$ and $I = O(\log(\log L))$ is an impossible condition to meet when generating any infinite class of QGC's, not merely the reflected Gray codes. But, in fact, we shall construct a class of QGC's satisfying these constraints.

To find an interesting trade-off phenomenon between I and U, it suffices to consider codes that are relatively difficult to generate. Given any small $\varepsilon > 0$, we will show that there exist QGC's for each dimension $n \geqq n_0(\varepsilon)$, such that for each α in the interval $\frac{1}{2}+\varepsilon \leqq \alpha \leqq 1$, a DAT generating the QGC with $I \leqq \alpha n$ and $U \leqq 2^{(1-\alpha+\varepsilon)n}$ exists. Moreover, for any DAT generating these codes and satisfying $I \leqq \alpha n$, the lower bound $U \geqq 2^{(1-\alpha-\varepsilon)n}$ also holds. At first glance this seems absurd; more memory must be updated than would ever need to be probed. But we can explain this by remembering that we are considering worst case values. When averaged over all code vectors, it turns out that a very small amount of update takes place per code vector generation. Only on extremely rare occasions is substantial update required.

While the results presented in this paper pertain strictly to the generation of quasi-Gray codes, it is clear that the DAT model of computation can be adapted for the purpose of studying the complexity of algorithms which generate other kinds of combinatorial configurations; e.g. combinations, partitions, permutations, etc. In [4], Ehrlich introduces the notion of a loopless algorithm. Roughly speaking, an algorithm for generating combinatorial objects is called *loopless* if the transition from a particular object to its successor is computed in constant time (on a conventional machine), independently of the size or dimension of the objects being generated. The DAT model provides a precise and more stratified notion of cost, in terms of which the so-called loopless algorithms appearing in [1]–[8] actually require $O(\log n)$ time (n = object size). Moreover, Lemma 1 (below) suggests that $O(\log n)$ is the best possible.

3. An information theoretic lower bound.

LEMMA 1. *For any DAT which generates a QGC of length L, we have $I \geqq \log(\log L)$.*

Proof. Let $x_1, x_2, \cdots, x_n$ be the variables in the DAT that represent the vector components of the generated QGC. (In the sequel, the words "variable" and "component" will be used interchangeably, exploiting the correspondence between variables and components of the current value vectors.) We shall call one of these variables *active* if it takes on both values 0 and 1 over the entire code sequence. Clearly, the number n_A of active variables satisfies $n_A \geqq \log L$ if the code has length L. Because a QGC is cyclic, if x_j is an active variable, then both assignment statements, $x_j \leftarrow 0$ and $x_j \leftarrow 1$, must appear in the DAT. Because two consecutive code vectors differ in only one component, it follows that the DAT must have at least $2n_A$ leaves, and the lemma follows immediately.

4. Some special DAT constructions. Our first construction shows that we can come close to achieving the bound in Lemma 1, even when imposing the constraint $U = 1$. We begin with some definitions and notation.

Given a decision assignment tree T, we let $X(T)$ denote the set of components (variables) appearing in T, and we let $X_I(T)$ denote the subset of $X(T)$ consisting of those components that label the internal nodes of T. Let ω_1 be the zero vector of dimension $|X(T)|$, and as in the Introduction, define the sequence $\omega_1, \omega_2, \omega_3, \cdots$, by the rule $\omega_{i+1} = T(\omega_i)$. Let ω_j^I denote the projection of ω_j consisting of those components in $X_I(T)$. Each vector ω_j^I determines a path through T from the root to

some leaf. Given a particular leaf $\mathscr{L}$ of T, we let $\phi(\mathscr{L})$ denote the set

$$\{\omega_j^I | \mathscr{L} \text{ is the leaf reached by input } \omega_j^I\}.$$

We say that T is *parity preserving* if and only if for each leaf $\mathscr{L}$ in T, all vectors in $\phi(\mathscr{L})$ have either an even number of 1's, or all have an odd number of 1's. Moreover, we say that $\mathscr{L}$ has even parity, or $\mathscr{L}$ has odd parity in accordance with the parity of the vectors in $\phi(\mathscr{L})$.

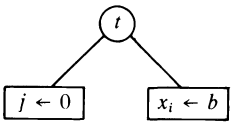
Let T and $\bar{T}$ be two DAT's. We say that T and $\bar{T}$ are *isomorphic* if and only if they are isomorphic as ordered binary trees, and there is a one-to-one mapping between the sets $X(T)$ and $X(\bar{T})$ preserving node labels and assignment statements. In particular, if T and $\bar{T}$ are isomorphic, then the induced functions $T(\omega)$ and $\bar{T}(\omega)$ are identical. We will have occasion to notationally distinguish certain components in $X(T)$ by parenthetically listing them following the name T, e.g. $T(x, y, z)$. When we state that $T(x, y, z)$ and $\bar{T}(\bar{x}, \bar{y}, \bar{z})$ are isomorphic, it is to be understood that $\bar{x}$, $\bar{y}$, and $\bar{z}$ are the respective isomorphic images of x, y, and z.

Given a parity preserving DAT with distinguished components, $T(x, \cdots, z)$, which satisfies $U = 1$, we define the extended DAT, $T^e(x, \cdots, z, t, j)$ where t and j are not in $X(T)$, as follows. If

(1) $\boxed{x_i \leftarrow b}$ $\quad b = 0$ or 1

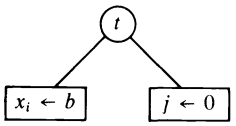
is a leaf of even parity, we replace it by

(2E)



On the other hand, if (1) is a leaf of odd parity, we replace it by

(2O)



Observe that U remains 1 and I increases by 1 when "extending" the DAT. In the sequel it will become apparent that this extending construction fits into a mechanism for implementing subroutine calls. The $\boxed{j \leftarrow 0}$ leaf acts as a transfer instruction, and t is inverted before the return from the subroutine takes place.

Next, we define Flip(u, v, t, k) to be the following DAT shown in Fig. 1. Observe that if initially $u = v = 1$, then after repeated application of Flip, we ultimately reach the leaf $\boxed{k \leftarrow 0}$, with $u = v = 0$, and t complemented in value; hence the name Flip.

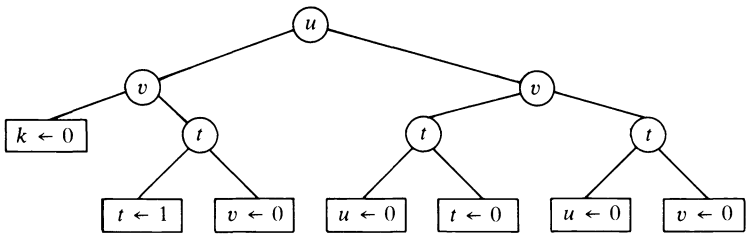


FIG. 1

Using the above definitions and notation, we define a sequence of trees $T_n(j, l, f, z)$, $n \geqq 0$. As will be indicated below, these trees are parity preserving, hence they can be extended as described above.

$T_0(j, l, f, z) = T_1(j, l, f, z) =$

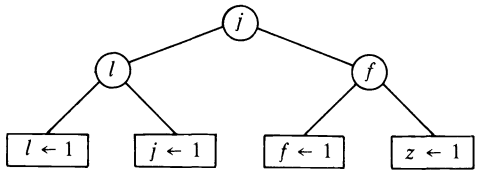


FIG. 2

First, we define T_0 and T_1 in Fig. 2. We define T_{n+1} in terms of T_n and T_{n-1} as follows. Let $T_n(a, b, c, k)$ and $T_{n-1}(d, e, f, l)$ be DAT'S isomorphic to $T_n(j, l, f, z)$ and $T_{n-1}(j, l, f, z)$ respectively, such that the variable sets, $X(T_n(a, b, c, k))$ and $X(T_{n-1}(d, e, f, l))$ are disjoint. Moreover, assume that j, u, v, t and z do not appear in either of these variable sets. Then $T_{n+1}(j, l, f, z)$ is given by Fig. 3. Finally, we define $\hat{T}_n$ to be the DAT shown in Fig. 4. Observe that the DAT's $\hat{T}_n$ satisfy $U = 1$.

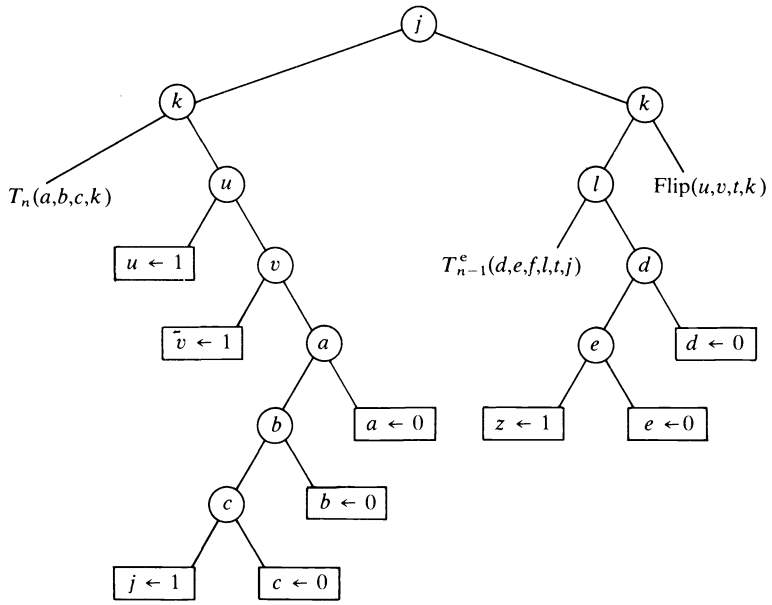


FIG. 3

THEOREM 1. *The DAT $\hat{T}_n$, $n \geqq 0$, generates a QGC of length $L = L_n$ and dimension $d = d_n$ where*

$$L_n = \begin{cases} 8 & \text{if } n = 0 \text{ or } 1, \\ (L_{n-1}+8)(L_{n-2}-4)+6 & \text{if } n \geqq 2. \end{cases}$$

$$d_n = \begin{cases} 4 & \text{if } n = 0 \text{ or } 1, \\ d_{n-1}+d_{n-2}+5 & \text{if } n \geqq 2. \end{cases}$$

Also,

$$I = I_n = \begin{cases} 8 & \text{if } n = 1, \\ 2n+8 & \text{if } n = 0 \text{ or } n \geqq 2. \end{cases}$$

Remark. In terms of L_n, we have $I_n = 2{\cdot}88 \log(\log L_n) + 0(1)$. Hence, these DAT's have a value I which comes to within a constant factor of the theoretical lower bound given in Lemma 1.

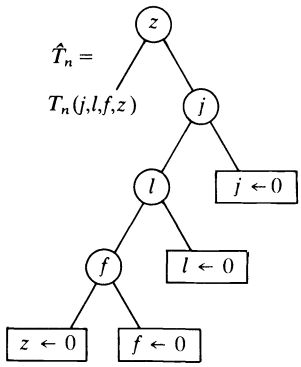


FIG. 4

Informal proof of Theorem 1. The motivation behind the DAT construction in Fig. 3 is provided by the following overview of its operation. Beginning with the zero vector of dimension d_{n+1}, repeated application of the induced T_{n+1} function yields a long sequence of vectors, ultimately leading to the vector which defines the path reaching the leaf, $\boxed{z \leftarrow 1}$. For this vector, it will be the case that $j = l = f = 1$, and all of its remaining components will equal zero. We call this fact the *exit condition*, and it is proven by induction on n.

Describing the T_{n+1} iterations of the zero vector in greater detail, we observe that these iterations define paths into the T_n subtree until the leaf $\boxed{k \leftarrow 1}$ is reached. In accordance with the exit condition induction hypothesis, for the vector reaching this leaf, we have $a = b = c = 1$, and the other components in $X(T_n)$ equal zero. The next six iterations lead to the vector which reaches the leaf $\boxed{j \leftarrow 1}$, and for this vector, $u = v = k = 1$, and its remaining components in $X(T_n)$ equal zero. The next four iterates define paths into the Flip subtree, the last of which, ω (say), reaches the leaf $\boxed{k \leftarrow 0}$ in the Flip subtree. Consider now $\omega' = T_{n+1}(\omega)$; all of its components in $X(T_n)$ equal 0, $u = v = 0$, $t = 1$ (by virtue of Flip), and ω' defines a path into $T^{e}_{n-1}(d, e, f, l, t, j)$. Noting that $X(T^{e}_{n-1}) - \{t, j\} = X(T_{n-1})$, we observe that all components of ω' in $X(T_{n-1})$ equal 0, and therefore, the totality of these components has even parity. Consequently, the situation depicted in (2E) applies and ω' reaches the $\boxed{x_i \leftarrow b}$ leaf. For $\omega'' = T_{n+1}(\omega')$, it follows that the totality of its components in $X(T_{n-1})$ has odd parity, and ω'' reaches a $\boxed{j \leftarrow 0}$ leaf, since the situation depicted in (2O) applies. Hence, when considering $\omega''' = T_{n+1}(\omega'')$, we conclude that $j = 0$ and all components in the left subtree of T_{n+1} equal 0. From the "perspective" of the left subtree of T_{n+1}, the generation process seemingly starts over from the beginning.

In general, the sequence of iterations has a cyclical structure. The first vector in each cycle has zero values for the components in the left sub-tree of T_{n+1}, and $j = 0$, as is the case with ω'''. The last two vectors in a cycle are analogous to ω' and ω'' above.

Denoting by ν_p and ν'_p, respectively, the last two vectors of the pth cycle, we now argue by induction on p that ν'_p leads to a $\boxed{j \leftarrow 0}$ leaf in T^e_{n-1}, with the exception of the last cycle. Assume that ν'_{p-1} leads to a $\boxed{j \leftarrow 0}$ leaf. The next cycle begins with vectors leading into the left subtree of T_{n+1}. By the time ν_p is generated, t has changed in value due to the action of Flip, and so instead of reaching the same leaf $\boxed{j \leftarrow 0}$ in T^e_{n-1} that ν'_{p-1} reaches, ν_p reaches its brother leaf $\boxed{x_i \leftarrow b}$. We have two cases to consider: (a) the leaf reached by ν_p is not the $\boxed{l \leftarrow 1}$ leaf, and (b) it is the $\boxed{l \leftarrow 1}$ leaf. For case (a), we observe that $\nu'_p = T_{n+1}(\nu_p)$ has opposite parity within $X(T_{n-1})$ than has ν_p. Hence ν'_p reaches a $\boxed{j \leftarrow 0}$ leaf in T^e_{n-1}, followed by the commencement of a new iteration cycle, as promised. For case (b), we shall observe that this pth cycle is the final cycle, and we write $p = p_{\text{final}}$. By the exit hypothesis applied to T_{n-1}, for the vector $\nu_{p\text{final}}$ we have $d = e = f = 1$, and all other components in $X(T_{n-1})$ equal 0. We also have that $t = 0$, since for vectors of the form ν_p, the totality of components in $X(T_{n-1}) \cup \{t\}$ has odd parity, as can be readily proven by induction on p. The third iteration beyond $\omega_{p\text{final}}$ reaches the $\boxed{z \leftarrow 1}$ leaf, and for this vector, $j = l = f = 1$, and all other components in $X(T_{n+1})$ equal 0, verifying the exit condition for T_{n+1}.

We now see that the variable t, the subtree Flip, and the parity considerations establish a control structure for implementing "subroutine calls" into the left subtree of T_{n+1}; the $\boxed{j \leftarrow 0}$ leaves constitute the transfer mechanism. By an induction on n left to the reader, taking into account the operation of T_n as described above, it can be verified that T_n is parity preserving. The "trimmings" added to T_n that make up $\hat{T}_n$ are necessary to satisfy the cycle constraint that QGC's must satisfy. The expressions for L_n, d_n and I_n in Theorem 1 are readily derived from the above discussion.

We turn next to the subject of generating G_n, the binary reflected Gray code of dimension n. The following constructions culminate in a DAT for generating G_n with $I = O(\log n)$ and $U = 7$. It is readily verified that if the constraint $U = 1$ is imposed, then for any such DAT generating G_n, $I = n$.

First, we describe an algorithm which can be expressed as a DAT with $I = O(\log n)$ and $U = O(\log n)$. Then we describe the necessary modifications to reduce U. Our algorithm will be based on the following well-known observation (see [7, § 5.2.1]). In generating the code G_n, the transition from the kth vector to its successor is achieved by inverting the lth component (from the right), where k and l are related as follows. When adding 1 to the binary radix representation of $k-1$, the carry propagates to the lth digit; that is, digits 1 through $l-1$ are reset to zero, and digit l is set to 1. This rule is readily proven by induction on n. Our algorithm will involve a representation for numbers which is sufficiently similar to the binary radix representation, that when performing the operation on the representation corresponding to adding 1 (modulo 2^n), the extent of carry propagation is effectively determined. Moreover, in terms of our DAT costs, this operation is efficiently performed.

Our representation involves what we shall call *tree numbers*. Let T denote an extended binary tree with n leaves. (An extended binary tree satisfies the property that each node has either zero or two sons, leaves defined as nodes with zero sons. The reader should not confuse this use of the word "extended" with its previous use in connection with parity preserving DAT's.) Consider labeling each node of T either 0 or 1. We say that the labeling is *proper* if and only if the following condition is satisfied.

(P) If a node is labeled 1, then its brother and descendants are all labeled 0. (It follows that its ancestors are also labeled 0.)

A labeling of T (proper or otherwise) represents an n digit binary radix number as follows.

(R) The kth digit (from the right) is 1 if and only if one of the nodes in T on the path from the root of T to the kth leaf (from the right) is labeled 1.

Figure 5 shows a properly labeled tree and the binary radix number which it represents.

LEMMA 2. *Given an n digit binary radix number α and a tree T with n leaves, there exists a unique proper labeling of T which represents α in accordance with* (R).

This lemma is readily proven by induction on the size of T, or equivalently, on n.

A *tree number* is defined to be a properly labeled tree. The *skeleton* of a tree number is the underlying unlabeled binary tree. We use the notation l_T to denote a tree number with skeleton T, and we let $\alpha(l_T)$ denote the integer represented by l_T as defined by (R). Given l_T, we let l'_T denote the tree number such that $\alpha(l'_T) \equiv 1 + \alpha(l_T)$ (modulo 2^n), where n is the number of leaves in T.

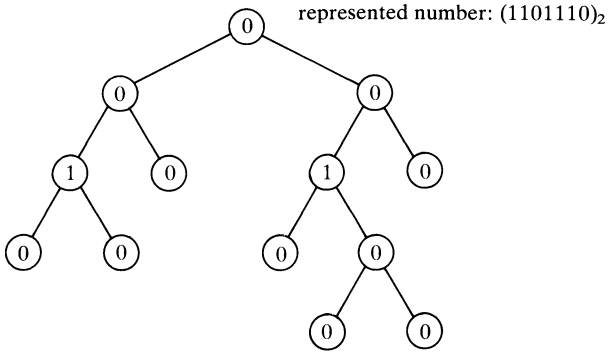


FIG. 5

We present an algorithm, named Increment, which takes as input a tree number l_T, and relabels T to yield as output l'_T. The algorithm utilizes the following notation. Root(T) denotes the root of T, RS(n) denotes the right son of node n. The left brother of node n is denoted by LB(n), and Father(n) denotes the father of n. Finally, Label(n) denotes the 0, 1 label of node n.

Increment

I1. Set $n \leftarrow$ Root(T). If Label(n) = 1, then reset Label(n) $\leftarrow$ 0 and Halt.

I2. (Now Label(n) = 0). If n is a leaf, then go to step I4; otherwise replace $n \leftarrow$ RS(n) and go to step I3.

I3. (Node n is the right son of a node.) If Label(n) = 1, then reset Label(n) $\leftarrow$ 0, replace $n \leftarrow$ LB(n), and go to step I2; otherwise directly return to step I2.

I4. If n is the right son of a node and Label(LB(n)) = 1, then reset Label(LB(n)) $\leftarrow$ 0, replace $n \leftarrow$ Father(n) and repeat this step; otherwise set Label(n) $\leftarrow$ 1 and Halt.

The following comments explain in part how this algorithm works.

(a) In step I1, if Label(n) = 1, then $\alpha(l_T) = 2^n - 1$. Resetting Label(n) $\leftarrow$ 0 yields the tree number l'_T such that $\alpha(l'_T) = 0$.

(b) Consider the labeled subtree of node n in step I3. If ${}^lT_n$ denotes the corresponding tree number, the algorithm effectively adds 1 to this tree number. In particular, if Label(n) = 1, the assignment, Label(n) $\leftarrow$ 0, resets ${}^lT_n$ so that $\alpha({}^lT_n) = 0$, and the assignment $n \leftarrow$ LB(n) effectively propagates the carry.

(c) The explanation of why Label(n) = 0 at the beginning of step I2 is as follows. The first time step I2 is executed, we have $n \leftarrow$ Root(T) and Label(n) = 0 from step I1. Thereafter, transfer to step I2 takes place from step I3. If Label(n) = 0 at the beginning of step I3, then the transfer to step I2 is immediate, and indeed Label(n) = 0 upon entering step I2. On the other hand, if Label(n) = 1 at the beginning of step I3, then (P) implies that Label(LB(n)) = 0; and before transfer to step I2 takes place, the replacement, $n \leftarrow$ LB(n) takes place.

(d) If the step I4 was to be replaced simply by

I4′. Label(n) $\leftarrow$ 1,

then the resulting labeling would represent the appropriate number as defined by (R), but the labeling might not be proper. The repetition of step I4 as prescribed by the algorithm ensures that the resulting labeling is proper. This is accomplished by finding the highest ancestor n_0 of n such that all of the binary digits of $\alpha(l'_T)$ corresponding to the leaves of the subtree of n_0 are 1, and then setting Label(n_0) $\leftarrow$ 1, as well as resetting to 0 the labels of the descendants of n_0.

To modify the Increment algorithm so that it generates successive vectors of G_n, we must first define a second label, $d(n)$, for each of the leaves of T. Given that n is the jth leaf (from the right) in T, regard $d(n)$ as the jth component (from the right) of the currently generated code vector in G_n. If $n' =$ Root(T), then identify $d(n')$ with $d(n)$, where n is the leftmost leaf of T. Now we modify Increment as follows. Just before the Halt in step I1, we insert, set "$d(n) \leftarrow 0$," and at the beginning of step I4, insert "$d(n) \leftarrow 1 - d(n)$."

In assessing the implied DAT costs of this algorithm, we do not include the costs of operations or tests on the static structure of T; these operations and tests are reflected in the DAT structure, and serve as a descriptive convenience for constructing the DAT. The quantity I reflects the cumulative number of tests on Label(n) values, plus 1 for the operation $d(n) \leftarrow 1 - d(n)$, should it be performed; and U reflects the cumulative number of assignments, Label(n) $\leftarrow$ 0 (or 1), plus 1 for the operation $d(n) \leftarrow 1 - d(n)$ in step I4, or the operation $d(n) \leftarrow 0$ in step I1. It is clear that both I and $U = O(h)$ where h is the height of T. Because a tree with n leaves need only have height $\log n$, our algorithm, Increment, modified as described above, lends itself to the construction of a DAT for generating G_n with $I = 0(\log n)$ and $U = O(\log n)$.

An analysis of Increment would show that the average amount of update per code vector generation is bounded by a constant independent of n. This suggests the possibility of an algorithm that amortizes over subsequent code vector generations the infrequent large amounts of update required by Increment. We illustrate how this works for incrementing radix numbers, and then sketch how it could be incorporated within Increment to produce a DAT which generates G_n and satisfies $I = O(\log n)$ and $U = 7$.

Consider the operation of adding 1 to a binary radix number. On the average, only a bounded number of digits need to be changed. We describe a number system similar to the binary radix system, but which enjoys the property that only a uniformly bounded number of digits need ever change when adding 1. The numbers in this system, which we shall call *contagious numbers*, are represented by strings over the three-symbol alphabet, $\{0, \theta, 1\}$. The symbol θ acts as a "contagious" zero. The conversion of the contagious number $a_n a_{n-1} \ldots a_1$ to its equivalent binary radix number, $(b_n b_{n-1} \ldots b_1)_2$ is defined by the following rule.

$$\text{(C)} \qquad b_j = \begin{cases} 0 & \text{if } a_j = 0 \text{ or } \theta, \text{ or for some } i < j,\ a_i = \theta \text{ and } a_{i+1} = \cdots = a_j = 1, \\ 1 & \text{otherwise.} \end{cases}$$

For example, $111011\theta1001\theta1 = (1110000100001)_2$. Observe that if the contagious number s has no occurrences of θ, then s is its own binary radix equivalent. For every contagious number s, we let $v(s)$ denote the equivalent binary radix number defined by rule (C). Assume $s = a_n \ldots a_1$. The following algorithm, Add1, specified changes to some of the a_i's, resulting in a contagious number s' such that $v(s') = v(s) + 1$.

Add 1
A1. Find the least i such that $a_i = 0$ or θ.
A2. If $a_i = \theta$ and $a_{i+1} = 1$, then set $a_{i+1} \leftarrow \theta$.
A3. Set $a_i \leftarrow 1$.
A4. If $i > 1$, then set $a_{i-1} \leftarrow 0$.
A5. If $i > 2$, then set $a_1 \leftarrow \theta$.
A6. Halt.

For example, $1 + 1011\theta1111 = 101\theta10111\theta$. The purpose of step A4 is to set up a barrier between the newly established 1 in the ith digit and the digits to its right, which are converted to zeros when applying rule (C). Observe that no more than four digits are ever changed in a single application of Add1, and that the value of i in step A1 is the extent of carry propagation that takes place when adding 1 to $v(s)$. This allows us to incorporate contagious numbers into a scheme for generating G_n with U uniformly bounded. However, $I = \Omega(n)$.

These techniques can be generalized to construct *contagious tree numbers*. We describe an algorithm, named CIncrement, which is a modification of Increment for handling contagious tree numbers. The quantities Label(ν) referred to in algorithm Increment can take on three values, 0, θ, and 1, in CIncrement. Other modifications are as follows. Let $\nu_1, \nu_2, \cdots, \nu_k$ be the nodes satisfying the condition, Label(ν_j) = 1, $1 \leqq j \leqq k$, in step I3 in the order in which they are encountered. Instead of resetting Label(ν_j) $\leftarrow 0$ for each j, $1 \leqq j \leqq k$, as indicated in step I3, CIncrement only resets Label(ν_k) $\leftarrow 0$, and also, if $k > 1$, sets Label(ν_1) $\leftarrow \theta$; this being reminiscent of the algorithm, Add1. Likewise, let $\nu_1, \nu_2, \cdots, \nu_l$ be the nodes satisfying Label(LB(ν_j)) = 1, in step I4. Instead of resetting Label(LB(ν_j)) $\leftarrow 0$ for each j, as indicated in step I4, CIncrement only resets Label(LB(ν_l)) $\leftarrow 0$, and also, if $l > 1$ sets Label(LB(ν_1)) $\leftarrow \theta$. Before setting Label(ν) $\leftarrow 1$, which takes place at the conclusion of step I4, CIncrement first checks to see if Label(ν) = θ, and if so, "propagates the θ" in analogy with step A2 of Add1. Specifically, if ν is the left son of a node and LB(Father(ν)) = 1, then CIncrement sets LB(Father(ν)) $\leftarrow \theta$. On the other hand, if ν is the right son of a node, CIncrement examines the sequence of nodes, $\nu_1 = \mathrm{RS}(\mathrm{LB}(\nu))$ $\nu_{j+1} = \mathrm{RS}(\nu_j)$, $j \geqq 1$, until it finds a node in this list such that Label(ν_j) = 1, or determines that there is no such node. Should such a node ν_j be discovered, CIncrement sets Label(ν_j) $\leftarrow \theta$. In every case, the algorithm eventually sets Label(ν) $\leftarrow 1$, as takes place at the conclusion of step I4.

As we were able to do with Increment, we can incorporate labels $d(\nu)$ into CIncrement to generate the code G_n. If we interpret 00, 10, and 11 to be binary representations for the labels, 0, 1, and θ, respectively, then this leads to a construction for a DAT which generates G_n with $I = O(\log n)$ and $U = 7$.

5. A trade-off between I and U. In this section we present a theorem which provides an example of an inherent trade-off phenomenon between I and U.

THEOREM 2. *Given $\varepsilon > 0$, there exists for each $n \geqq n_0(\varepsilon)$ a QGC of dimension n such that for each α in the interval $\frac{1}{2} + \varepsilon \leqq \alpha \leqq 1$, a DAT generating this QGC and*

satisfying $I \leqq \alpha n$ *and* $U \leqq 2^{(1-\alpha+\varepsilon)n}$ *exists. Moreover, for any DAT generating this QGC and satisfying* $I \leqq \alpha n$, *the lower bound* $U > 2^{(1-\alpha-\varepsilon)n}$ *also holds.*

Remark. The reader should observe that for any QGC of dimension n, there exists a DAT with $I = n$ and $U = 1$ which generates the QGC. The following lemmas are used in the proof of Theorem 2.

LEMMA 3. *For any QGC of dimension n and any* α *such that* $\frac{1}{2} \leqq \alpha \leqq 1$, *there exists a DAT generating the QGC and satisfying*

$$I = \max\left(\tfrac{1}{2}n + O(\log n), \lceil \alpha n \rceil + 1\right), \tag{3}$$

$$U = O((\log n)2^{(1-\alpha)n} + n). \tag{4}$$

Proof. Let Q denote a QGC of dimension n, and let $x_1, x_2, \cdots, x_n$ denote the n components of the code vectors of Q. Let $A(j)$, $1 \leqq j \leqq L$ (L = length of Q), denote the appropriate assignment statement $x_i \leftarrow 0$ (or 1) which generates the successor to the jth code vector. As there are $2n$ possible assignment statements, $x_i \leftarrow 0$ (or 1), involving these n variables, it is possible to encode a statement $A(j)$ using $O(\log n)$ bits. We let $c(A(j))$ denote such an encoding. What follows is an algorithm schema, algorithm G, for generating Q. Algorithm G uses two counter variables, C_1 and C_2. Counter C_1 counts from 1 to $L_1 = 2^{n-\lceil \alpha n \rceil} - 1$, and C_2 counts from 1 to $L_2 = L/(L_1 + 1)$. (For the sake of simplicity of presentation, we can assume that L is a power of 2.) Algorithm G also employs an array $R(j)$, $1 \leqq j \leqq L_1$, of $O(\log n)$ bit words, as well as a binary variable, Load. We define the initial memory configuration ω_1 as follows: $(x_1, x_2, \cdots, x_n)$ = first code vector, $C_1 = C_2 = 1$, Load = "off", and $R(j) = v(A(j))$ for $1 \leqq j \leqq L_1$. The configuration ω_1 specifies the complete contents of memory (including data structure) for representing the first code vector. Observe that $n - \lceil \alpha n \rceil$ bits suffice to represent all possible values of C_1, and because $L \leqq 2^n$, $\lceil \alpha n \rceil$ bits suffice to represent C_2. The steps of algorithm G follow.

ALGORITHM G

G1. If Load = "on", then go to step G4.
G2. Perform the assignment statement encoded by $R(C_1)$.
G3. If $C_1 = L_1$, then set Load $\leftarrow$ "on" and Halt; otherwise set $C_1 \leftarrow C_1 + 1$ and halt.
G4. Perform the assignment statement, $A(C_2(L_1 + 1))$.
G5. Perform the appropriate collection (depending on C_2) of single bit assignment statements to set up the array R so that $R(j) = c(A(C_2(L_1 + 1) + j))$ for $1 \leqq j \leqq L_1$.
G6. If $C_2 < L_2$ then set $C_2 \leftarrow C_2 + 1$; otherwise set $C_2 \leftarrow 1$.
G7. Set $C_1 \leftarrow 1$, set Load $\leftarrow$ "off", and Halt.

This algorithm can be formulated as a DAT satisfying (3) and (4). In particular if Load = "off" in step G1, then steps G2 and G3 are performed. Reading C_1 and $R(C_1)$ require reading $n - \lceil \alpha n \rceil + O(\log n)$ bits, which we bound by the expression, $\frac{1}{2}n + O(\log n)$, in (3). On the other hand, if Load = "on" in step G1, then steps G4, G5, G6, and G7 are performed. Reading Load and C_2 contributes the term $\lceil \alpha n \rceil + 1$ to (3). The assignment $C_1 \leftarrow 1$ in step G7 involves the appropriate setting of the $n - \lceil \alpha n \rceil$ bits of C_1, contributing to U (but not to I). The assignment, $C_2 \leftarrow C_2 + 1$, in step G6 (or $C_2 \leftarrow 1$) similarly contributes $\lceil \alpha n \rceil$ to U (the contribution to I involved in reading C_2 already accounted for). The assignments in step G5 contribute the remainder of the expression for U in (4).

LEMMA 4. *There are at least* $2^{2^{n-3}}$ *QGC's of dimension* $n \geqq 2$.

Proof. Given a set of n objects, $S = \{s_1, s_2, \cdots, s_n\}$, we define a J_n *sequence*, a sequence over S consisting of $2^n - 1$ terms, inductively as follows.

(5) For a one element set, $S = \{s_1\}$, the one term sequence, s_1, is a J_1 sequence over S.

(6) For an n element set, $S = \{s_1, \cdots, s_n\}$, $n > 1$, the sequence $s_{i_1}, s_{i_2}, \cdots, s_{i_l}, s_j, s_{m_1}, s_{m_2}, \cdots, s_{m_l}$, $l = 2^{n-1} - 1$, is a J_n sequence over S provided that $s_{i_1}, s_{i_2}, \cdots, s_{i_l}$ and $s_{m_1}, s_{m_2}, \cdots, s_{m_l}$ are J_{n-1} sequences over $S - \{s_j\}$.

Letting Z_n denote the number of J_n sequences over S, we clearly have that

$$Z_1 = 1, \tag{7}$$

$$Z_n = nZ_{n-1}^2 \quad \text{for } n > 1. \tag{8}$$

We define a one-to-one mapping from J_n sequence over $\{1, \cdots, n\}$ into QGC's of dimension $n+1$ as follows. Let γ be the J_n sequence $m_1, m_2, \cdots, m_l$ over $\{1, 2, \cdots, n\}$, and let B_i, $1 \leqq i \leqq n+1$, denote the $n+1$ dimensional binary vector whose ith component equals 1, and whose other components are zero. We define the QCG $Q(\gamma) = v_1, v_2, v_3, \cdots, v_L$ by the rules

$$v_i = \begin{cases} \sum_{j=1}^{i-1} B_{m_j} & \text{for } 1 \leqq i \leqq 2^n, \\ & \text{(summation modulo 2)} \\ B_{n+1} + v_{2^{n+1}+1-i} & \text{for } 2^n < i \leqq 2^{n+1}. \end{cases} \tag{9}$$

It is readily verified from (5), (6) and (9) that $Q(\gamma)$ defines a one-to-one mapping from J_n sequences into QGC's of dimension $n+1$, and Lemma 4 follows from (7) and (8).

LEMMA 5. *The number of distinct QGC's of dimension n that can be generated by DAT's with* $I \leqq r$ *and* $U \leqq s$ *is no greater than*

$$(1 + 2(2^r + n))^{(1+s)2^r}. \tag{10}$$

Proof. Assume T is a DAT with $I \leqq r$ and $U \leqq s$ that generates Q, a QGC of dimension n. We can assume without loss of generality that x_i is the variable in T designating the values of the ith component of the code vectors of Q. Since T has at most 2^r internal nodes, we can also assume that the variables which label the internal nodes of T are contained in the set, $X = \{x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_{2^r}\}$. Moreover, without loss of generality we can assume, for each assignment statement $z \leftarrow 0$ (or 1) appearing in a leaf of T, that either z labels one of the internal nodes, or $z = x_i$ for some i; in other words, $z \in X$. For if $z \notin X$, removal of any assignment statements involving z would not change the fact that T (so modified) generates Q.

Now consider the number of ways to construct a DAT consisting of a complete binary tree T_r of height r, with each internal node labeled with a variable in X and each leaf containing at most s assignment statements, $z \leftarrow 0$ (or 1), with $z \in X$. On the basis of the above discussion, this number provides a bound on the number of QGC's considered in Lemma 5. The number of ways of labeling the internal nodes of T_r is bounded by $|X|^{2^r}$. By introducing a new label Λ which denotes the null statement, we can assume that each leaf of T_r contains exactly s statements, and it follows that there are $\leqq (1 + 2|X|)^{s2^r}$ ways to structure the leaves of T_r. The bound expressed in (10) follows immediately.

Proof of Theorem 2. Given $\varepsilon > 0$, we show that for sufficiently large n, there exists a QGC of dimension n, Q (say), such that:

(11) For any DAT generating Q and satisfying $I \leqq \alpha n$ where $0 \leqq \alpha \leqq 1$, the inequality $U > 2^{(1-\alpha-\varepsilon)n}$ holds.

Let $N(r, s)$ denote the number of QGC's of dimension n which can be generated by DAT's satisfying $I \leqq r$ and $U \leqq s$. We estimate the sum,

$$S_\varepsilon(n) = \sum_{r=1}^{\lfloor n-\varepsilon n \rfloor} N(r, 2^{n-r-\varepsilon n})$$

and show that when n is sufficiently large, $S_\varepsilon(n) < 2^{2^{n-3}}$. The QGC's not enumerated by $S_\varepsilon(n)$ are those satisfying (11). (By virtue of the remark following the statement of Theorem 2, there is no need to consider DAT's with $I > n$.) Since the total number of QGC's is at least $2^{2^{n-3}}$, as stated by Lemma 4, this inequality for $S_\varepsilon(n)$ implies the existence of QGC's satisfying (11). From Lemma 5, when $1 \leqq r \leqq n - \varepsilon n$,

$$N(r, 2^{n-r-\varepsilon n}) \leqq (1 + 2(2^r + n))^{(1+2^{n-r-\varepsilon})2^r} \leqq 2^{2^{n-\varepsilon n + O(\log n)}}$$

Hence, for n sufficiently large, $S_\varepsilon(n) < 2^{2^{n-3}}$ as claimed. The upper bound on U stated in Theorem 2 follows from Lemma 3, completing the proof.

COROLLARY. *Almost all QGC's of dimension n can only be generated by DAT's satisfying* $I \geqq \frac{1}{2}n - O(\log n)$.

Proof. Because there is no purpose served by having a given variable appear in two assignment statements within the same leaf, we can reason as in Lemma 5 to conclude that $U \leqq 2^I + n$. Therefore, the number of QGC's that can be generated by DAT's with $I \leqq r$ is no more than

$$(1 + 2(2^r + n))^{(1+2^r+n)2^r} \leqq 2^{2^{(2r+O(\log n))}}$$

Reasoning as in the proof of Theorem 2, we conclude the corollary.

Using a construction similar to Algorithm G in the proof of Lemma 3, one can show that any QGC can be generated by a DAT such that, when averaged over all code vectors in the QGC, $T = O(\log n)$.

Acknowledgment. The author wishes to thank Professor S. G. Williamson for several stimulating discussions.

REFERENCES

[1] J. R. BITNER, G. EHRLICH, and E. M. REINGOLD, *Efficient generation of the binary reflected Gray code and its applications*, Comm. ACM, 19 (1976), pp. 517–521.

[2] N. DERSHOWITZ, *A simplified loop-free algorithm for generating permutations*, BIT, 15 (1975), pp. 158–164.

[3] G. EHRLICH, *Algorithm 466, four combinatorial algorithms*, Comm. ACM, 16 (1973), pp. 690–691.

[4] ———, *Loopless algorithms for generating permutations, combinations and other combinatorial configuration*, J. Assoc. Comput. Mach., 20 (1973), pp. 500–513.

[5] S. EVEN, *Algorithmic Combinatorics*, Macmillan, New York, 1973.

[6] A. NIJENHUIS AND H. S. WILF, *Combinatorial Algorithms*, Academic Press, New York, 1975.

[7] E. M. REINGOLD, J. NIVERGELT AND N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[8] M. B. WELLS, *Elements of Combinatorial Computing*, Pergamon Press, New York, 1971.

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

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

SELECTING THE KTH ELEMENT IN $X+Y$ AND $X_1+X_2+\cdots+X_m$*

DONALD B. JOHNSON† AND TETSUO MIZOGUCHI‡

Abstract. An algorithm is given which selects the Kth element in $X+Y$ in $O(n \log n)$ time and $O(n)$ space, where $X+Y$ is the multiset $\{x_i+y_j | x_i \in X \text{ and } y_j \in Y\}$ for $X=(x_1, x_2, \cdots, x_n)$ and $Y=(y_1, y_2, \cdots, y_n)$, n-tuples of real numbers. The results are extended to $\sum_{i=1}^{m} X_i$ for $m>2$. There is strong evidence that this more general problem is difficult if m and K may be selected arbitrarily. However, algorithms can be shown which are fast for small K and arbitrary m.

Key words. selection, order statistics, weighted median, pair selection, optimal algorithm, $\mathcal{NP}$-hard problems

Introduction. Until the appearance of an $O(n)$ algorithm for selecting the Kth element in an n-element set with a total order [2], it was widely believed that any uniform algorithm for this problem would require time proportional to the time to sort the set. We consider the more general problem of finding the Kth element in the multiset $\sum_{i=1}^{m} X_i = \{\sum_{i=1}^{m} x_i | x_i \in X_i\}$ where for $i=1, 2, \cdots, m$, $X_i=(x_{i1}, x_{i2}, \cdots, x_{in})$, an n-tuple of real numbers, and the rank of an element in $\sum_{i=1}^{m} X_i$ is its rank in some nonincreasing order of the values $\sum_{i=1}^{m} x_i$. In the given n-tuples X_i, the elements appear in arbitrary order. When $m=2$ we denote the multiset $\sum_{i=1}^{m} X_i$ as $X+Y$.

A straightforward application of the linear algorithm for selecting singletons [2] to problems where $m \geqq 2$ involves forming all the m-tuples with values in $\sum_{i=1}^{m} X_i$, giving an algorithm with a running time which realizes $O(n^m)$. We show how to do better. For finding the Kth largest element in $X+Y$, we give an algorithm which runs in $O(n \log n)$ time,[1] less time asymptotically than is required to construct the pairs. This bound has been shown to be optimal to within a constant factor [5]. When our algorithm is extended to cases where $m>2$, the running time for all K is $O(mn^{\lceil m/2 \rceil} \log n)$. It appears unlikely that an algorithm can be found with running time polynomial in n if both m and K are unrestricted subject to $m \leqq n$. The existence of such an algorithm implies, as we show, that $\mathcal{P}$-TIME $=\mathcal{NP}$-TIME.[2] Enumerative algorithms can be exhibited which run in polynomial time when K is bounded by a polynomial in n.

The related problem of sorting $X+Y$ has been considered in [4] where it is shown that $n^2 \log n$ comparisons suffice and, under a certain class of comparison algorithms, are necessary. Under a less restrictive model of computation, $X+Y$ can be sorted in $O(n^2)$ comparisons [3].

Our results apply to finding the Kth largest element in $\sum_{i=1}^{m} X_i$ when the X_i are sets, and also to the problem on $X+X$ when the elements paired are either required to have distinct indices or not. In $X+X$ we adopt the convention that for each i and j, exactly one of x_i+x_j and x_j+x_i appears.

Weighted medians. To obtain our result we use a technique to find a certain partition in linear time. Let $A=(a_1, a_2, \cdots, a_n)$ be an n-tuple of elements drawn

* Received by the editors June 28, 1976, and in revised form April 4, 1977.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802.
‡ Mitsubishi Electric Corporation, Kamakura Works, 325 Kamimachiya, Kamakura City, Japan 247.

[1] All logarithms are base 2.

[2] $\mathcal{P}$-TIME is the class of decision problems solvable in time polynomial in the size of the problem input on a deterministic Turing machine. $\mathcal{NP}$-TIME is the analogous class defined with respect to nondeterministic Turing machines.

(with repetition allowed) from a universe with a total order, and let $w: A \to \mathbb{R}^+$ be a weight function defined on A. It is desired to find a partitioning element a_{i_m} satisfying $\sum_{j \leqq m} w(a_{i_j}) \geqq \sum_{j > m} w(a_{i_j})$ and $\sum_{j < m} w(a_{i_j}) \leqq \sum_{j \geqq m} w(a_{i_j})$ where $a_{i_1}, a_{i_2}, \cdots, a_{i_n}$ is an ordering of A consistent with the total order on the universe. The element a_{i_m} is called the *weighted median of A with respect to w*. It may be seen that a_{i_m} is unique for any fixed ordering of A.

Given an n-tuple A and the weights $w(A)$, the weighted median a_{i_m} can be found in $O(n)$ time by means of a binary search of A. Since A is not sorted, the binary search is implemented using a linear median-finding algorithm [2]. At the first step, the median element $a_{i_{\lceil n/2 \rceil}}$ is found and used to partition A. The weights are summed over each of the elements of the partition to test whether $a_{i_{\lceil n/2 \rceil}}$ is the weighted median. If not, the process is repeated on the partition element known to contain the weighted median, and so on. At each step the availability of the sums of weights previously computed allows the test for the weighted median to be performed in time proportional to the size of the collection of elements known to contain the weighted median. Since at each repetition of the process at least half of the elements in this collection is discarded, the running time of the entire procedure to find the weighted median and the partition it induces is $O(n)$.

Finding the Kth pair. The collection $X + Y$ may be represented in a canonical form as the matrix $B = (b_{ij})$ where $b_{ij} = x_i + y_j$ for x_i the ith largest element in X and y_j the jth largest element in Y. If elements repeat in either X or Y, rank i or rank j is defined with respect to some fixed total order consistent with the ordering of the domain. It is convenient to visualize the algorithm operating on B although, of course, B is not constructed.

The algorithm first sorts X and Y separately and then proceeds as follows. For each row i of B the median element $a_i = b_{il_i}$ is found. Each median element is assigned a weight $w(a_i)$, the number of elements in row i of M. Then the weighted median $a_m = b_{pl_p}$ of $(a_1, a_2, \cdots, a_n)$ is found. The first iteration is then completed by discarding from B at least $1/4$ of the elements known not to be the Kth largest. If K is sufficiently large, the elements discarded are $(b_{ij_i}|$ for $i = 1, 2, \cdots, n$, $b_{ij_i} \geqq a_m)$. If K is sufficiently small, the elements discarded are $(b_{ij_i}|$ for $i = 1, 2, \cdots, n$, $b_{ij_i} \leqq a_m)$. For certain intermediate values of K it is possible that a_m is a Kth pair. If a_m is a Kth pair, it is reported and the algorithm terminates. Otherwise, discarding the elements indicated yields a submatrix (not necessarily rectangular in shape) B' of B on which the above process is repeated. As will be brought out in the analysis, after $O(\log n)$ repetitions of this process either the Kth pair will have been reported or the submatrix of B will contain $O(n)$ elements from which the desired element may be extracted by known methods.

ALGORITHM KTHPAIR.

Input: $X = (x_1, x_2, \cdots, x_n)$, $Y = (y_1, y_2, \cdots, y_n)$, n-tuples of real numbers, and K, an integer satisfying $1 \leqq K \leqq n^2$.

Output: (i, j) satisfying $x_i + y_i$ is Kth in a nonincreasing ordering of all elements in $X + Y$.

Method:

1) Sort X and Y separately into nonincreasing order. Without loss of generality let $X = (x_1, x_2, \cdots, x_n)$ and $Y = (y_1, y_2, \cdots, y_n)$ where, for $i = 1, 2, \cdots, n - 1$, $x_i \geqq x_{i+1}$ and $y_i \geqq y_{i+1}$;
2) Let $L = 0$, $R = n^2$;
   For $i = 1, 2, \cdots, n$, let $Lb(i) = 1$, $Rb(i) = n$;

3) **while** $R-L>n$ **do**
begin

3a) Let $A=(a_i|a_i=x_i+y_{j_i}$, where $j_i=\lfloor(Lb(i)+Rb(i))/2\rfloor$, for all $i=1,2,\cdots,n$ satisfying $Lb(i)\leqq Rb(i))$. For $a_i\in A$ let $w(a_i)=Rb(i)-Lb(i)+1$;

3b) Let $a_m\in A$ be the weighted median of A with respect to w; Partition the pairs as follows:

$$\text{For } i=1,2,\cdots,n \text{ let } P(i)=\begin{cases}0 & \text{if } x_i+y_1\leqq a_m,\\ \max\{j|x_i+y_j>a_m\} & \text{otherwise,}\end{cases}$$

$$Q(i)=\begin{cases}n+1 & \text{if } x_i+y_n\geqq a_m,\\ \min\{j|x_i+y_j<a_m\} & \text{otherwise;}\end{cases}$$

3c) **if** $K\leqq\sum_{i=1}^{n}P(i)$ **then** for $i=1,2,\cdots,n$ let $Rb(i)=P(i)$
else if $K>\sum_{i=1}^{n}(Q(i)-1)$ **then** for $i=1,2,\cdots,n$ let $Lb(i)=Q(i)$
else return (i,j) satisfying $x_i+y_j=a_m$;
Let $L=\sum_{i=1}^{n}(Lb(i)-1)$, $R=\sum_{i=1}^{n}Rb(i)$
end;

4) Sort $((x_i+y_{j_i})|$ for $i=1,2,\cdots,n$, $Lb(i)\leqq j_i\leqq Rb(i))$. Let (x_i+y_j) be the element of rank $K-L$ in this sorted order.
Return (i,j).

Throughout the execution of algorithm KTHPAIR the arrays Lb and Rb define a partition of the canonical pair matrix B into three collections,

$$B_L=(b_{ij_i}| \quad \text{for } i=1,2,\cdots,n,\ 1\leqq j_i<Lb(i)),$$
$$B_C=(b_{ij_i}| \quad \text{for } i=1,2,\cdots,n,\ Lb(i)\leqq j_i\leqq Rb(i)),$$
$$B_R=(b_{ij_i}| \quad \text{for } i=1,2,\cdots,n,\ Rb(i)<j_i\leqq n).$$

Before the execution of step 3), $B_C=B$ and B_L and B_R are empty. To show the correctness of the algorithm it suffices to show that $R-L$ decreases with each iteration in step 3) and that

$$\mathrm{T}\equiv\begin{cases}b_1>b_2>b_3 \text{ for all } b_1\in B_L,\ b_2\in B_C,\ b_3\in B_R \text{ and}\\ \text{there exists a pair } x+y \text{ in } B_C \text{ which is of rank } K\\ \text{in } X+Y \text{ and of rank } K-L \text{ in } B_C\end{cases}$$

holds following execution of step 3).

Condition T holds trivially before the execution of step 3), and it may be seen that if T holds before some execution of step 3a) it is preserved over the execution of steps 3a), 3b), and 3c). Since a_m belongs to B_C before step 3c) and does not belong to B_C after normal termination of step 3c) it follows that $R-L$ decreases over each iteration of steps 3a), 3b), and 3c) except possibly the last. By induction it may be concluded that a Kth pair is always found either in step 3c) or in step 4).

To bound the running time of the algorithm it may be seen that steps 1), 2), and 4) together run in $O(n\log n)$ time, and the computations in steps 3a) and 3c) run in $O(n)$ time. With respect to step 3b), it has already been observed that the weighted median

can be found in $O(n)$ time. The values in P may be computed in $O(n)$ time as follows.

Let $j = 0$;
for $i = n, n-1, \cdots, 1$ **do**
  begin
    while $j < n$ and $x_i + y_{j+1} > a_m$ **do** let $j = j+1$;
    Let $P(i) = j$
  end.

A similar procedure computes the values in Q. To complete the proof of an $O(n \log n)$ bound on the running time for the entire algorithm it remains to show that steps 3a), 3b), and 3c) are repeated $O(\log n)$ times.

From the definition of the weighted median presented in the previous section it may be seen that step 3c) either returns the desired indices (i, j) or moves at least $1/4$ of the elements in B_C to either B_L or B_R. Let w be the number of times step 3c) is repeated. From T it is clear that B_C always contains at least one element. Thus

$$\left(\frac{3}{4}\right)^w n^2 \geqq 1,$$

$$w(\log 3 - 2) + 2 \log n \geqq 0,$$

$$w \leqq \frac{2 \log n}{2 - \log 3} = O(\log n).$$

The proof of the following theorem is completed by inspection of the data structures employed by the algorithm, verifying that $O(n)$ space is consumed.

THEOREM 1. *Algorithm KTHPAIR finds a Kth largest pair in $X + Y$, where X and Y are n-tuples of real numbers and K is an integer satisfying $1 \leqq K \leqq n^2$, in $O(n \log n)$ time and $O(n)$ space.*

COROLLARY. *The Kth pair in $X + X$ and the Kth pair, composed of members with distinct indices, in $X + X$ can be found in $O(n \log n)$ time.*

Proof. Algorithm KTHPAIR is easily modified to solve these problems. In the case of $X + X$ where indices within a pair are not required to be distinct, $Lb(i)$ is initialized to i for $i = 1, 2, \cdots, n$, and K is restricted to the range $1 \leqq K \leqq n(n+1)/2$. In the second case, pairs composed of elements with equal indices are excluded by initializing $Lb(i)$ to $i+1$ for $i = 1, 2, \cdots, n$ and restricting K to the range $1 \leqq K \leqq n(n-1)/2$. □

The selection problem on $X_1 + X_2 + \cdots + X_m$. The ideas in the previous section extend immediately to the selection problem on $\sum_{i=1}^{m} X_i$ for $m > 2$.

THEOREM 2. *The Kth m-tuple in $\sum_{i=1}^{m} X_i$ for $1 \leqq K \leqq n^m$ may be found in*

$$O(mn^{\lceil m/2 \rceil} \log n) \text{ time} \quad \text{and} \quad O(n^{\lceil m/2 \rceil}) \text{ space.}$$

Proof. For m even, let $X = \sum_{i=1}^{m/2} X_i$ and $Y = \sum_{i=m/2+1}^{m} X_i$. Algorithm KTHPAIR may be applied to find the Kth pair in $X + Y$. If care is taken to construct X and Y in lexicographic order so that the original indices can be recovered from the solution to the problem on $X + Y$ then the space required is bounded by $O(n^{m/2})$, the size of X and Y. It follows from the analysis for Theorem 1 that the running time is bounded by $O(mn^{m/2} \log n)$. The bounds for m odd are obtained in a similar manner. □

A drawback of this construction is the space consumed when m is large. As the following theorem indicates, however, it is not surprising that the time bound is exponential in m.

THEOREM 3. *The existence of an algorithm for selecting the Kth m-tuple in* $\sum_{i=1}^{m} X_i$ *which runs in time polynomial in n for all* $m \leqq n$ *and* $K \leqq \frac{1}{2}n^m$ *implies* $\mathcal{P}$*-TIME* $= \mathcal{NP}$*-TIME.*

Proof. It is known that determining the existence of nonnegative integers $x_1, x_2, \cdots, x_n$ satisfying $\sum_{i=1}^{n} a_i x_i = b$ and $\sum_{i=1}^{n} x_i \leqq n$, given nonnegative integers b and $a_1, a_2, \cdots, a_n$, is $\mathcal{NP}$-complete [7] (the reader unfamiliar with $\mathcal{NP}$-completeness and the significance of the question whether $\mathcal{P}$-TIME $= \mathcal{NP}$-TIME is referred to [1].) A selection algorithm which finds the Kth m-tuple in $\sum_{i=1}^{m} X_i$ can be used to solve this problem by performing a binary search of the set of m-tuples in $\sum_{i=1}^{m} (a_1, a_2, \cdots, a_n)$ for each $m = 1, 2, \cdots, n$. For each m, the search requires $O(m \log n)$ applications of the selection algorithm. Thus the existence of a polynomial-time selection algorithm implies a polynomial-time algorithm for an $\mathcal{NP}$-complete problem. □

In the preceding analysis the dependence on m of the complexity of the selection problem on $\sum_{i=1}^{m} X_i$ is analyzed for the worst case with respect to K. While the algorithms of Theorems 1 and 2 run in time independent of K, the result in Theorem 3 depends on allowing K to be large. Lawler [6] has given a general procedure for enumerating the K best combinatorial objects, generated from a given set of size n, in time $O(KnT(n))$ where the best object can be found in $T(n)$ time. A straightforward application of this result gives an algorithm for the selection problem on $\sum_{i=1}^{m} X_i$ which runs in $O(Kn^2m)$. Rather than recount the details here we present an improved algorithm for the related problem where only m-tuples composed of elements with distinct indices are drawn from $mX = X + X + \cdots + X$. This algorithm, which selects the Kth subset of size m from X, runs in $O(n \log n + K \log K)$ time.

The algorithm operates by generating a portion of a tree consistent with a total ordering of the set of subsets. If without loss of generality the given $X = (x_1, x_2, \cdots, x_n)$ satisfies $x_1 \geqq x_2 \geqq \cdots \geqq x_n$, then it is clear that the largest subset is $\{x_1, x_2, \cdots, x_m\}$. The second largest is $\{x_1, x_2, \cdots, x_{m-1}, x_{m+1}\}$, and the third largest is either $\{x_1, x_2, \cdots, x_{m-2}, x_m x_{m+1}\}$ or $\{x_1, x_2, \cdots, x_{m-1}, x_{m+2}\}$. The tree which the algorithm constructs has $\{x_1, x_2, \cdots, x_m\}$ at the root and, in general, a vertex in this tree has as sons those subsets which can be generated according to certain transformations on the father which produce candidates for the next subset in the total order. The transformations guarantee a tree with exactly one vertex for each m-tuple. At any point in time only a frontier of the tree is stored by the algorithm. The frontier never grows larger than K. The algorithm stores the frontier in the set Q. In Q, subsets are represented by configurations of the form (w, f, u, d). A configuration alone is not necessarily a unique description of a subset. However, any subset in the tree constructed by the algorithm is completely described by its configuration together with the sequence of configurations labeling the path to it from the root. Thus, in the algorithm, a configuration stands for precisely one subset. In a configuration, w is the sum of the elements in the corresponding subset. The indices f, u, and d are relative to a sorted representation of X. The index f is the largest index of the elements which will be *fixed* in all descendants of (w, f, u, d) in the tree. That is, if for $i < f$ the element x_i is present in the subset described by (w, f, u, d) it will be present in all descendants. If it is absent it will appear in no descendant. It is easily seen that x_f is absent in the subset described by (w, f, u, d). The index u is the greatest index of elements of X *used* in the subset corresponding to (w, f, u, d) or any of its predecessors in the tree. It can be seen that x_u is always present in the subset described by (w, f, u, d). For $f < i \leqq u$, x_i is present in the subset described by (w, f, u, d) unless $i = d$. Element x_d is the *deleted* element in the subset.

ALGORITHM KTHSUBSET.

Input: $X=(x_1, x_2, \cdots, x_n)$, an n-tuple of real numbers, integers $m \leqq n$ and K satisfying $1 \leqq K \leqq \binom{n}{m}$.

Output: A set $\{i_1, i_2, \cdots, i_m | i_j \in \{1, 2, \cdots, n\}$ for $j=1, 2, \cdots, m\}$ satisfying $\sum_{j=1}^{m} x_{i_j}$ is Kth largest among all subsets of size m drawn from X, where subsets are ranked on the sums of their elements.

Method:

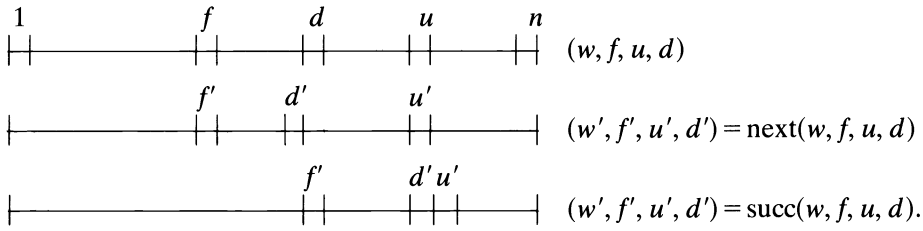
1) Sort X in nonincreasing order; without loss of generality let $X=(x_1, x_2, \cdots, x_n)$ where $x_i \geqq x_{i+1}$ for $i=1, 2, \cdots, n-1$;
2) **if** $K=1$ **then** output $\{1, 2, \cdots, m\}$
   else
   begin
   Put $(x_{m+1}+\sum_{i=1}^{m-1} x_i, 0, m+1, m)$ into Q;
   for $i := 2$ **until** $K-1$ **do**
   begin
   Let (w, f, u, d) maximize w among the 4-tuples in Q;
   Delete (w, f, u, d) from Q;
   Put next(w, f, u, d) and succ(w, f, u, d) into Q;
   while $|Q| > K-i$ **do**
   delete (w, f, u, d) minimizing w among the 4-tuples in Q
   end
   Let (w, f, u, d) maximize w among the 4-tuples in Q;
   Output $\{i_1, i_2, \cdots, i_m\}$ selected by (w, f, u, d)
   end;

where

$$\text{next}(w, f, u, d) = \begin{cases} (w + x_d - x_{d-1}, f, u, d-1) & \text{if } d-1 > f, \\ \Lambda & \text{otherwise,} \end{cases}$$

$$\text{succ}(w, f, u, d) = \begin{cases} (w + x_{u+1} - x_u, d, u+1, u) & \text{if } u+1 \leqq n \text{ and } u > d, \\ \Lambda & \text{otherwise.} \end{cases}$$

Putting Λ into Q is intended to be a vacuous operation. The functions next and succ may be represented pictorially as



The usual techniques employed in dynamic programming can record the changes leading to any (w, f, u, d) in Q without an asymptotic increase in storage used. From this information the output itself can be reconstructed in time $O(K+m)$.

THEOREM 4. *Algorithm KTHSUBSET constructs the Kth subset of size m from a set of n real numbers in $O(n \log n + K \log K)$ time and $O(n+K)$ space.*

Proof. Consider any sequence of configurations generated by applications of next and succ from $(x_{m+1}+\sum_{i=1}^{m-1} x_i, 0, m+1, m)$. If (w, f, u, d) occurs before (w', f', u', d') in this sequence then $w' \leqq w$ and either $u' > u$ or $u' = u$ and $d' < d$. Thus no configuration repeats and it follows that the algorithm does indeed generate a binary tree in

which subset weights are nonincreasing on any path from the root. It also may be seen from examination of the functions next and succ that if the next and succ configurations generated from some configuration (w, f, u, d) are both nonvacuous then any subset represented in the subtree rooted in one son will differ from every subset represented in the subtree rooted in the other son at some index i, $f < i < d$. Thus no subset is represented twice. Furthermore, given any subset, a sequence of next and succ moves can be shown which will generate a path representing this subset. It follows from these arguments that the entire tree is potentially generated by the algorithm though in general the algorithm will terminate before the construction is completed. Furthermore, the algorithm guarantees that a frontier of the tree will always be contained in Q. The frontier has the property that all predecessors, in the tree, of vertices in the frontier have been accounted for, that is, deleted from Q. Correctness follows from these arguments.

If Q is maintained as a double heap (see, for example, [1]) in which both a largest and a smallest element can be found and deleted in time $O(\log |Q|)$, the time bound $O(n \log n + K \log K)$ is immediate. The bound on space required follows from the bound of $O(K)$ on $|Q|$ and the observation that the number of vertices in the tree (and hence the amount of information necessary to determine the indices in the Kth subset) is also $O(K)$. □

Acknowledgment. One of the authors is grateful to Professor H. Enomoto at Tokyo Institute of Technology for fruitful discussions.

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. BLUM, R. W. FLOYD, V. R. PRATT, R. L. RIVEST AND R. E. TARJAN, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), no. 4, pp. 448–461.

[3] M. FREDMAN, *Two applications of a probabilistic search technique: sorting $X + Y$ and building balanced search trees*, Proc. 7th Annual ACM Symposium on Theory of Computing (May 1975), Association for Computing Machinery, 1975, pp. 240–244.

[4] L. H. HARPER, T. H. PAYNE, J. E. SAVAGE AND E. STRAUS, *Sorting $X + Y$*, Comm. ACM, 18 (1975), no. 6, pp. 347–349.

[5] D. B. JOHNSON AND S. D. KASHDAN, *A lower bound on the complexity of finding the K-th largest pair*, Tech. Rep. No. 183, Computer Science Dept., Pennsylvania State Univ., University Park, PA, (Feb. 1976).

[6] E. L. LAWLER, *A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem*, Management Sci., 18 (1972), no. 7, pp. 401–405.

[7] G. S. LUEKER, *Two polynomial complete problems in nonnegative integer programming*, Computer Science TR-178, Princeton Univ., Princeton, NJ, March 1975.

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

AN $O(N \cdot \log N)$ ALGORITHM FOR A CLASS OF MATCHING PROBLEMS*

NIMROD MEGIDDO† AND ARIE TAMIR‡

Abstract. The following class of matching problems is considered. The vertices of a complete undirected graph are indexed $1, \cdots, n$, where $n = 2m$. Every vertex i is assigned two numbers a_i, b_i. The length of every edge (i, j), where $i < j$, is $d(i, j) = a_i + b_j$. This class of weighted graphs is applicable to scheduling and optimal assignment problems. A maximum weighted (perfect) matching is found in $O(n \cdot \log n)$ operations.

Key words. matching, assignment, scheduling, polynomial-time algorithm, 2-3 trees

1. Introduction. The maximum matching problem has many applications in operations research. The first polynomial-time bounded algorithm for the maximum weighted matching problem is Edmonds' [2]. The most efficient algorithm for the maximum (cardinality) matching, known to the authors, is Even and Kariv's [3]. Gabow [4] has the most efficient algorithm for the weighted matching. In this paper we focus on a subclass of maximum weighted matching problems (see § 2 for a precise definition). Our study is motivated by the following two problems which are easily shown to belong to our class.

In the first problem, a group of individuals, ordered by seniority, is to be partitioned into teams, having the same mission. Each team consists of two positions—a senior position and a junior one. The senior position must be manned by the more senior individual between the members of the team. Assuming that we know the effectiveness of each individual in both the senior and the junior positions, we wish to maximize the total effectiveness of the teams.

The second problem is to schedule $2m$ jobs to m identical processors, two jobs to each processor, preserving the arrival ordering. The objective is to minimize the total flow time, or equivalently, the average waiting time of a job.

Using a dynamic programming approach, these two models can be solved in $O(m^2)$ time. In this paper we present an algorithm which solves the above problems in $O(m \cdot \log m)$ operations.

2. Preliminaries. Our goal is to develop an efficient algorithm for the following problem.

Problem 1. Given numbers a_i, b_i, $i = 1, \cdots, n$ $(n = 2m)$, find a perfect matching $(i_1, j_1), \cdots, (i_m, j_m)$, where $i_k < j_k$, $k = 1, \cdots, m$, which maximizes $\sum_{k=1}^{m} (a_{i_k} + b_{j_k})$.

We may assume without loss of generality that a maximum matching $(i_1, j_1), \cdots, (i_m, j_m)$ satisfies $i_k < i_{k+1}$, $j_k < j_{k+1}$, $k = 1, \cdots, m-1$. In view of this we shall restrict our attention to matchings $(i_1, j_1), \cdots, (i_m, j_m)$ which satisfy $i_k < j_k$, $k = 1, \cdots, m$, and $i_k < i_{k+1}$, $j_k < j_{k+1}$, $k = 1, \cdots, m-1$. These can be handled by introducing the following notation.

Let $x = (x_1, \cdots, x_n)$ be a vector whose components are either 1 or -1. Denote $H_i(x) = \sum_{k=1}^{i} x_k$, $i = 1, \cdots, n$. Let X be the set of all vectors x $(x = (x_1, \cdots, x_n)$, $x_i \in \{1, -1\})$ such that $H_i(x) \geqq 0$, $i = 1, \cdots, n-1$ and $H_n(x) = 0$. Consider the following problem.

* Received by the editors April 7, 1977.

† Department of Statistics, Tel-Aviv University, Tel-Aviv, Israel. Now at Department of Business Administration, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

‡ Department of Statistics, Tel-Aviv University, Tel-Aviv, Israel.

Problem 2. Maximize $c(x) \equiv \sum_{i=1}^{n} (a_i - b_i) \cdot x_i$ over X.

We claim that Problems 1 and 2 are equivalent. Specifically, if $(i_1, j_1), \cdots, (i_m, j_m)$ solves Problem 1 then the vector x, where $x_k = 1$ if $k = i_q$ and $x_k = -1$ if $k = j_q$, solves Problem 2. Conversely, if x solves Problem 2 then a solution to Problem 1 is defined recursively as follows. Let $i_1 = 1$. Suppose that $i_1, \cdots, i_q$ and $j_1, \cdots, j_r$ $(0 \leqq r \leqq q)$ have been defined, and $\{i_1, \cdots, i_q, j_1, \cdots, j_r\} = \{1, \cdots, q+r\}$. Then, if $x_{q+r+1} = 1$ let $i_{q+1} = q+r+1$ and if $x_{q+r+1} = -1$ let $j_{r+1} = q+r+1$. Thus, we shall henceforth be dealing with Problem 2. We note that Problem 2 can be transformed to a linear program with a totally unimodular matrix whose basic solutions yield solutions to our problem. Thus, the theory of linear programming suffices for solving Problem 2. However, we shall present an algorithm which is more suitable. Our algorithm is based on the following theorem.

THEOREM. *A vector x solves Problem* 2 *if and only if the following condition holds. For every pair i, j, $1 \leqq i < j \leqq n$,* (i) *if $x_i = -1$, $x_j = 1$ then $a_i - b_i \leqq a_j - b_j$ and* (ii) *if $x_i = 1$, $x_j = -1$ and $H_k(x) \geqq 2$, for $i \leqq k < j$, then $a_i - b_i \geqq a_j - b_j$.*

Necessity is obvious, since if the condition does not hold, then by defining $y_i = -x_i$, $y_j = -x_j$, and $y_k = x_k$ for $k \neq i, j$ we have $y \in X$ and $c(y) > c(x)$. We shall now prove the sufficiency of the condition. For $x, y \in X$ define a metric $D(x, y) = \#\{i: x_i \neq y_i\}$. Suppose that $x \in X$ does not solve Problem 2 and let $y \in X$ be a solution to Problem 2, which is nearest (with respect to (w.r.t.) D) to x. Let i be the smallest index such that $x_i = 1$ and $y_i = -1$. Let j be the smallest index such that $x_j = -1$ and $y_j = 1$. If $i > j$ then for every k, $j \leqq k < i$, $H_k(y) \geqq 2$. It follows that $a_j - b_j > a_i - b_i$ (equality cannot occur since it implies $D(x, z) < D(x, y)$, $c(z) = c(y)$, $z \in X$, where $z_i = 1$, $z_j = -1$, $z_k = y_k$ for $k \neq i, j$). Thus, part (i) of the condition does not hold. If $i < j$ then for every k, $i \leqq k < j$, $H_k(x) \geqq 2$. Similar arguments imply $a_j - b_j > a_i - b_i$ and in this case part (ii) does not hold.

3. The algorithm. We shall first describe our algorithm in general terms and then elaborate on its details. In this section we concentrate on the validity of the algorithm; an estimate of the number of operations is given in § 4.

Let $M_1 = \{1, \cdots, m\}$, $M_2 = \{m+1, \cdots, n\}$. For every $x \in X$ let

$$I(x) = \min \{i \in M_1 : H_k(x) \geqq 2 \text{ for all } k,\ i \leqq k \leqq m\},$$

$$J(x) = \max \{j \in M_2 : H_k(x) \geqq 2 \text{ for all } k,\ m+1 \leqq k \leqq j-1\}.$$

Our algorithm generates a sequence $x^0, \cdots, x^r$ of vectors in X such that $D(x^{k-1}, x^k) = 2$. This sequence develops according to the following scheme.

Scheme.

0. Initiate with $x = (1, \cdots, 1, -1, \cdots, -1) \in X$.
1. Find an $i \in M_1$ such that $x_i = 1$, $i \geqq I(x)$ and $a_i - b_i = \min \{a_k - b_k : I(x) \leqq k \leqq m, x_k = 1\}$; find a $j \in M_2$ such that $x_j = -1$, $j \leqq J(x)$ and $a_j - b_j = \max \{a_k - b_k : m+1 \leqq k \leqq J(x), x_k = -1\}$.
2. If $a_i - b_i \geqq a_j - b_j$ then terminate; otherwise, set $x_i = -1$, $x_j = 1$ and go to 1.

Let x^i $(i = 0, 1, \cdots)$ denote the vector x stored after i executions of step 2, and suppose that the scheme terminates after r iterations. It can be easily verified that $c(x^{k-1}) < c(x^k)$, $k = 1, \cdots, r$. Moreover, since $H_m(x^k) = m - 2k$ and $H_m(x^k) \geqq 0$ for $k = 0, 1, \cdots, r$, it follows that $r \leqq m/2$.

We shall now prove that upon termination the vector $x = x^r$ is a solution to Problem 2. This is done by verifying that the condition stated in the theorem is satisfied. Let $i < j$ be any pair $(1 \leqq i, j \leqq n)$. Distinguish cases: (i) $x_i = -1$, $x_j = 1$. If $i, j \in M_1$ then there is $q < r$ such that $x_i^q = 1$, $i \geqq I(x^q)$ and $a_i - b_i =$

$\min\{a_k - b_k : I(x^q) \leqq k \leqq m, \quad x_k^q = 1\}$. This implies $a_i - b_i \leqq a_j - b_j$. Analogous arguments hold in case $i, j \in M_2$. The case $i \in M_1$, $j \in M_2$ can be handled by applying this type of arguments twice. (ii) $x_i = 1$, $x_j = -1$, and $H_k(x) \geqq 2$ for $i \leqq k < j$. If $i, j \in M_1$ then there is $q < r$ such that $x_j^q = 1$, $j \geqq I(x^q)$ and $a_j - b_j = \min\{a_k - b_k : I(x^q) \leqq k \leqq m, x_k^q = 1\}$. However, since $H_k(x^q) \geqq H_k(x)$ $(k = 1, \cdots, m)$, it follows that $i \geqq I(x^q)$ and hence $a_i - b_i \geqq a_j - b_j$. A similar argument holds in case $i, j \in M_2$. If $i \in M_1$ and $j \in M_2$ then termination implies $a_i - b_i \geqq a_j - b_j$.

In fact, the sequence $x^0, \cdots, x^r$ can be generated without calculating the values $I(x)$, $J(x)$, $H_k(x)$ explicitly. This can be performed as follows. First, the elements i of M_1 are sorted according to increasing magnitude of $a_i - b_i$ and the elements j of M_2 are sorted according to decreasing magnitude of $a_j - b_j$. Let x^q be a vector in the sequence generated by the scheme. Let A_1 denote the ordered (by the natural order on M_1) q-tuple of the indices $i \in M_1$ such that $x_i^q = -1$. An index $i \in A_1$ is called a right minimum if $H_i(x^q) < H_k(x^q)$ for every $k \in M_1$ such that $k > i$. Let B_1 denote the ordered set of right minima. Linearly ordered sets A_2, B_2 are defined in analogous manner with respect to the elements in M_2; A_2 is the ordered tuple of the indices $j \in M_2$ such that $x_{j+1}^q = 1$ and B_2 consists of those $j \in A_2$ such that $H_j(x^q) < H_k(x^q)$ for every $k < j$ $(k \in M_2)$. Once the lists A_1, B_1, A_2, B_2 (w.r.t. a vector x) are known, it is easy to execute step 1 of the scheme. The following algorithm generates the same sequence as that generated by the scheme, and at the same time maintains the lists A_1, B_1, A_2, B_2. Our algorithm operates symmetrically on the sets M_1, M_2. Hence we shall describe in detail only the part concerning M_1.

ALGORITHM.

Phase I: Sort the elements i of M_1 to form a list L_1 arranged in order of increasing magnitude of $a_i - b_i$; sort M_2 to form a list L_2 arranged in decreasing order.

Phase II:

0. Initiate with $x = (1, \cdots, 1, -1, \cdots, -1) \in X$ and $A_1 = B_1 = A_2 = B_2 = \varnothing$.
1. Let i be the first element in L_1 and let $s = \#\{k : k \in A_1, k < i\}$.
2. If $i - 2s < 2$ then delete i from L_1 and go to 1; otherwise go to 3.
3. If there is no $k \in B_1$ such that $k > i$ then set $i^* = \infty$, $s^* = 0$ and go to 5; otherwise let i^* be the smallest element of B_1 such that $i^* > i$ and let $s^* = \#\{k : k \in A_1, k < i^*\}$.
4. If $i^* - 2(s^* + 1) < 2$ then delete i from L_1 and go to 1; otherwise go to 5.
5. Pick an element $j \in M_2$ in a manner similar to that by which i is picked from M_1 (see steps 1–4; j is the first in L_2 such that $2m - (j-1) - 2t \geqq 2$, where $t = \#\{k : k \in A_2, \ k \geqq j\}$, and either there is no $k \in B_2$ such that $k < j$, or $2m - j^* - 2t^* \geqq 2$, where j^* is the largest element of B_2 such that $j^* < j - 1$ and $t^* = \#\{k : k \in A_2, k \geqq j^*\}$).
6. If $a_i - b_i \geqq a_j - b_j$ then terminate; otherwise, set $x_i = -1$, $x_j = 1$ and go to 7.
7. Delete i from L_1 and insert i into A_1.
8. If $i - 2(s+1) \geqq i^* - 2(s^* + 2)$ then set $i = i^*$, $s = s^* + 1$ and go to 9; otherwise insert i into B_1.
9. If there is no $k \in B_1$ such that $k < i$ then go to 11; otherwise let i' be the largest element of B_1 such that $i' < i$ and let $s' = \#\{k : k \in A_1, k < i'\}$.
10. If $i' - 2(s' + 1) < i - 2(s + 1)$ then go to 11; otherwise delete i' from B_1 and go to 9.
11. Perform on j, A_2, B_2 operations similar to those performed on i, A_1, B_1 in steps 7–10 (delete j from L_2; insert $j - 1$ into A_2, if $j > m + 1$; insert $j - 1$ into B_2 if it has become a "left minimum" and delete from B_2 those elements that have ceased from being left minima).
12. Go to 1.

4. The efficiency of the algorithm. We may employ the device of a 2-3 tree (see [1, p. 146] for a precise definition) for handling the linearly ordered sets A_1, B_1, A_2, B_2 in our algorithm. Again, the symmetry enables us to restrict our attention to A_1 and B_1. Let T be a 2-3 tree which represents A_1. For every vertex v of T which is not a leaf, $L[v]$ is the largest element of A_1 assigned to the subtree whose root is the leftmost son of v; $M[v]$ is the largest element of A_1, assigned to the subtree whose root is the second son of v. For every vertex v of T let $a(v)$ denote the number of leaves of the subtree rooted in v, and let $b(v)$ denote the number of leaves of this subtree storing an element of B_1.

It can be easily verified (see [1]) that each one of the following operations can be executed in at most $O(\log n)$ steps: (a) Find the smallest element of A_1 which is greater than a given $i \in M_1$. (b) Find the smallest [largest] element of B_1 which is greater [smaller] than a given $i \in M_1$. (c) Insert an element into A_1. (d) Insert an element of A_1 into B_1. (e) Calculate s, s^*, s'.

Since each one of the operations listed above can be executed no more than $O(n)$ times in Phase II of our algorithm, and since these are essentially all the operations executed during Phase II, it follows that Phase II requires no more than $O(n \cdot \log n)$ steps. It is well-known that Phase I can also be executed in $O(n \cdot \log n)$ steps (see [1]).

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. EDMONDS, *Paths, trees and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[3] S. EVEN AND O. KARIV, *An $O(n^{5/2})$ algorithm for maximum matching in general graphs*, presented at the 16th Annual Symposium on Foundations of Computer Science, Univ. of California at Berkeley, October 1975.

[4] H. N. GABOW, *An efficient implementation of Edmonds' algorithm for maximum matching on graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 221–234.

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

ON GOOD EOL FORMS*

H. A. MAURER†, A. SALOMAA‡ AND D. WOOD¶

Abstract. This paper continues the study of EOL forms. The notion of a good EOL form is introduced as an important generalization of the notion of complete and very complete EOL forms. Transformations preserving the property good are obtained and the existence of a variety of good and bad (i.e. not good) forms is demonstrated. It is further shown that good and complete (i.e. vomplete) EOL forms do exist; that propagating EOL forms are bad except under very special circumstances; and that synchronized EOL forms are always bad.

Key words. grammar forms, L forms, goodness

Introduction. The notion of an EOL form and its interpretations is introduced in [2] to define families of structurally similar EOL systems. Basically, an EOL form F is just an EOL system, and the interpretation operator $\mathcal{G}$ defines for each such F an (infinite) family $\mathcal{G}(F)$ of related EOL systems. In a natural manner this also associates a family $\mathcal{L}(F)$ of EOL languages with each EOL form F by defining $\mathcal{L}(F)=\{L(F')|F'\in\mathcal{G}(F)\}$. An EOL form F is called *complete*, if $\mathcal{L}(F)=\mathcal{L}_{\text{EOL}}$ (the family of all EOL languages), and F is called *vomplete* (short for very complete) if for each EOL form $\bar{F}$ an $F'\in\mathcal{G}(F)$ exists such that $\mathcal{L}(\bar{F})=\mathcal{L}(F')$.

In this paper an important generalization of the notion vomplete is introduced. We call an EOL form F *good*, if for each EOL form $\bar{F}$ with $\mathcal{L}(\bar{F})\subseteq\mathcal{L}(F)$ some $F'\in\mathcal{G}(F)$ with $\mathcal{L}(\bar{F})=\mathcal{L}(F')$ exists. That is to say, an EOL form F is good if each subfamily of $\mathcal{L}(F)$ which can at all be generated by an EOL form can also be generated by an interpretation of F, i.e. by an EOL form structurally similar to the original one.

After a section containing a brief review of the basic notions of L form theory and some additional concepts needed in the sequel, we establish the existence of both good and bad (i.e. not good) EOL forms, both incomplete and complete (the construction of a good and complete, i.e. vomplete, EOL form solving an open question of [2]), we present transformations of EOL forms which preserve the property good, we establish that no propagating EOL form generating at least certain finite languages is good, and prove that even "weakly" synchronized EOL forms cannot be good. All together the results presented in this paper are a first attempt to provide some machinery for examining whether a given EOL form is good or not, a task which we believe is both important and difficult. In contrast to the results obtained it should e.g. be noted that the existence of bad *grammar* forms has not yet been established [1].

1. Preliminaries. In this section we review notions from L form theory required in this paper, and we introduce the central notion of the paper, the concept of a good EOL form.

An EOL *system* G is a quadruple $G=(V,\Sigma,P,S)$ where V is a finite set of symbols, $\Sigma\subseteq V$ is the set of *terminals*, $V-\Sigma$ the set of nonterminals, $S\in V-\Sigma$ is the *starting symbol* and P is a finite set of pairs (α,x) with $\alpha\in V$ and $x\in V^*$ such that for

* Received by the editors October 20, 1976. Part of this work was carried out at the University of Calgary and supported by the National Research Council of Canada Grant A-7700.

† Institut für Angewandte Informatik und Formale Beschreibungsverfahren Universität Karlsruhe, 7500 Karlsruhe, W-Germany.

‡ Department of Mathematics, University of Turku, Turku, Finland.

¶ Department of Applied Mathematics, McMaster University, Hamilton, Ontario, Canada.

each $\alpha \in V$ at least one such pair is in P. The elements $p = (\alpha, x)$ of P are called *productions* and are usually written as $\alpha \to x$. G is called *deterministic*, if for each $\alpha \in V$ exactly one production $\alpha \to x$ is in P, and G is called *propagating* if in each production $\alpha \to x$ the righthand side x differs from the empty word ε.

For words $x = \alpha_1 \alpha_2 \cdots \alpha_n$ with $\alpha_i \in V$ and $y = y_1 y_2 \cdots y_n$ with $y_i \in V^*$ we write $x \underset{G}{\Longrightarrow} y$ if $\alpha_i \to y_i$ is a production of P for every i. We write $x \overset{0}{\underset{G}{\Longrightarrow}} x$ for every x in V^* and write $x \overset{n}{\underset{G}{\Longrightarrow}} y$ if for some z in V^* $x \underset{G}{\Longrightarrow} z \overset{n-1}{\underset{G}{\Longrightarrow}} y$ holds. By $x \overset{*}{\underset{G}{\Longrightarrow}} y$ we mean $x \overset{n}{\underset{G}{\Longrightarrow}} y$ for some $n \geqq 0$, and by $x \overset{+}{\underset{G}{\Longrightarrow}} y$ we mean $x \overset{n}{\underset{G}{\Longrightarrow}} y$ for some $n \geqq 1$.

For convenience, the EOL system will often not be indicated below the arrow $\Longrightarrow$ if it is understood by the context.

A sequence of words $x_0, x_1, x_2, \cdots, x_n$ with $x_0 \Longrightarrow x_1 \Longrightarrow x_2 \Longrightarrow \cdots \Longrightarrow x_n$ is called a *derivation* (of *length n leading from* x_0 *to* x_n). The *language generated by* an EOL system $G = (V, \Sigma, P, S)$ is denoted by $L(G)$ and defined as $L(G) = \{x \in \Sigma^* | S \overset{*}{\underset{G}{\Longrightarrow}} x\}$. For convenience, languages which differ by at most ε will be considered equal. Classes of languages will be considered equal if for any nonempty language in one class a language in the other class, and conversely, exists which differ by at most ε.

For a set M of symbols and a set N of words, $M \to N$ denotes the set of productions $\{\alpha \to x | \alpha \in M, x \in N\}$. An EOL system $G = (V, \Sigma, P, S)$ is called *short* if for each production $\alpha \to x$ of P $|x| \leqq 2$ holds, and is called *binary* if each production is of one of the forms $A \to \varepsilon$, $A \to a$, $A \to B$, $A \to BC$ or $a \to A$, where $a \in \Sigma$ and $A, B, C \in V - \Sigma$.

Let $G = (V, \Sigma, P, S)$ be an EOL system and x_0, x_l be words in V^*. We say $x_0 \overset{l}{\underset{G}{\Longrightarrow}} x_l$ is *nonterminal* and write $x_0 \overset{l}{\underset{\text{nt } G}{\Longrightarrow}} x_l$, if for some sequence of words $x_1, x_2, \cdots, x_{l-1}$ with $x_i \underset{G}{\Longrightarrow} x_{i+1}$ for $i = 0, 1, \cdots, l-1$,

$$S \overset{*}{\underset{G}{\Longrightarrow}} y_0 x_0 z_0 \underset{G}{\Longrightarrow} y_1 x_1 z_1 \underset{G}{\Longrightarrow} y_{l-1} x_{l-1} z_{l-1} \underset{G}{\Longrightarrow} y_l x_l z_l$$

implies $y_i x_i z_i$ contains at least one nonterminal for $1 \leqq i \leqq l-1$. We say $x_0 \overset{l}{\underset{G}{\Longrightarrow}} x_l$ is *total nonterminal* and write $x_0 \overset{l}{\underset{\text{tnt } G}{\Longrightarrow}} x_l$ if the existence of a derivation $S \overset{*}{\Longrightarrow} w_0 \Longrightarrow w_1 \Longrightarrow \cdots \Longrightarrow w_l$ with x_0, x_l being substrings of w_0, w_l respectively, implies that w_i contains at least one nonterminal for $1 \leqq i \leqq l-1$.

We will now review basic definitions concerning EOL forms as introduced in [2]. We have defined above how an EOL system F generates a language $L(F)$. We will now define how an EOL system F can be used to generate a family of languages $\mathscr{L}(F)$. If we are interested in the generation of a language family (rather than a single language) then we will call F an EOL *form*. Thus, the terms "EOL form" and "EOL system" can be used interchangeably. The first emphasizes the aspect of language family generation, the second emphasizes the aspect of language generation.

Let $F = (V, \Sigma, P, S)$ be an EOL form. An EOL system $F' = (V', \Sigma', P', S')$ is called an interpretation of F (*modulo* μ), symbolically $F' \lhd F(\mu)$ if μ is a substitution defined on V and (i)–(v̈) hold:

(i) $\mu(A) \subseteq V' - \Sigma'$ for each $A \in V - \Sigma$.

(ii) $\mu(a) \subseteq \Sigma'$ for each $a \in \Sigma$.

(iii) $\mu(\alpha)\cap\mu(\beta)=\varnothing$ for all $\alpha\neq\beta$ in V.
(iv) $P'\subseteq(P)$ where $\mu(P)=\cup_{\alpha\to x\in P}\mu(\alpha)\to\mu(x)$.
(v) $S'\in\mu(S)$.

$\mathscr{G}(F)=\{F'|F'\lhd F\}$ is the *family of* EOL *forms generated by* F, and $\mathscr{L}(F)=\{L(F')|F'\lhd F\}$ is the *family of languages generated by* F.

We call two EOL forms F_1 and F_2 *form equivalent* if $\mathscr{L}(F_1)=\mathscr{L}(F_2)$, and *strongly form equivalent* if $\mathscr{G}(F_1)=\mathscr{G}(F_2)$. Note that the notion of form equivalence of two EOL forms F_1 and F_2 differs significantly from the well known notion of equivalence of F_1 and F_2, the latter meaning $L(F_1)=L(F_2)$, i.e. equality of languages rather than language families.

An EOL form $\bar{F}=(\bar{V},\bar{\Sigma},\bar{P},\bar{S})$ nt *simulates* an EOL form $F=(V,\Sigma,P,S)$ if for some integer $l\geqq 1$ $\alpha\overset{l}{\underset{\text{nt }\bar{F}}{\Longrightarrow}}x$ holds for each $\alpha\to x\in P$. Let $F=(V,\Sigma,P,S)$ be an EOL form and let $l\geqq 1$ be an integer. A symbol $\alpha\in V$ is called an *l-symbol* of F if for some $k\in\{0,l,2l,3l,\cdots\}$ words x,y exist such that $S\overset{k}{\underset{F}{\Longrightarrow}}x\alpha y$ holds. An EOL form $F=(V,\Sigma,P,S)$ is called *synchronized* if $x\in L(F)$ and $x\overset{+}{\Longrightarrow}y$ implies $y\notin\Sigma^*$. An EOL form F is called *complete*, if $\mathscr{L}(F)=\mathscr{L}_{\text{EOL}}$ (the family of all EOL languages, i.e. $\mathscr{L}_{\text{EOL}}=\{L(F)|F$ is an EOL system$\}$). If F is not complete it is called *incomplete*. The central notion of this paper is the concept of a *good* EOL form.

DEFINITION. An EOL form F is called *good*, if for each EOL form $\bar{F}$ with $\mathscr{L}(\bar{F})\subseteq\mathscr{L}(F)$ an EOL form F' exists such that $F'\lhd F$ and $\mathscr{L}(F')=\mathscr{L}(\bar{F})$. If F is not good it is called *bad*.

Intuitively, an EOL form F is good, if every language family $\mathscr{L}$ contained in $\mathscr{L}(F)$ which can at all be generated by an EOL form, can also be generated by an interpretation of F. Thus a good EOL form F describes, in a sense, all language families contained in $\mathscr{L}(F)$ which can be generated by EOL forms. An EOL form F is called *vomplete* (introduced in [2] as abbreviation of *very complete*) if F is complete and good.

We conclude this section by mentioning a convention used throughout. When specifying examples of EOL forms, small letters are used to denote terminals, capital letters to denote nonterminals, and S (or $\bar{S},\hat{S},S',\cdots$) to indicate the starting symbol.

2. Results. By a result in [2] there are bad complete EOL forms; we will show later that good complete EOL forms also exist. But we first establish the existence of both good and bad incomplete EOL forms.

THEOREM 2.1. *The EOL form F_1 with productions $S\to a$, $a\to N$, $N\to N$ is bad, the EOL form F_2 with productions $S\to a$, $a\to a$ is good, and both forms are incomplete.*

Proof. That both F_1 and F_2 are incomplete is clear. We first prove that F_1 is bad. Note that $\mathscr{L}(F_1)=\mathscr{L}_{\text{Symb}}$ (the family of languages in which each language is a finite set of single letter words). Consider the EOL form H with productions $S\to a$, $a\to b$, $b\to N$, $N\to N$. Clearly, $\mathscr{L}(H)\subseteq\mathscr{L}(F_1)$ and $\mathscr{L}(H)\neq\mathscr{L}(F_1)$, since $\{a\}\in\mathscr{L}(F_1)$ but $\{a\}\notin\mathscr{L}(H)$. However, for every interpretation F_1' of F_1 evidently $\mathscr{L}(F_1')=\mathscr{L}_{\text{Symb}}\neq\mathscr{L}(H)$.

We now show that F_2 is good. To see this, we let $\mathscr{L}_k$ be the subfamily of $\mathscr{L}_{\text{Symb}}$ consisting of languages with at least k elements and we establish the following assertion.

ASSERTION. *If K is an* EOL *form such that $\mathscr{L}(K)\subseteq\mathscr{L}_{\text{Symb}}$, then $\mathscr{L}(K)=\mathscr{L}_k$ for some $k>0$.*

Proof of assertion. Clearly, only the cardinality of the sets in $\mathscr{L}(K)$ is essential not the names of symbols. If $\{a_1, \cdots, a_m\} \in \mathscr{L}(K)$, then also $\{a_1, \cdots, a_{m+1}\} \in \mathscr{L}(K)$ since we can "split a_m into a_m and a_{m+1}" in the interpretation K' generating $\{a_1, \cdots, a_m\}$, i.e. we modify K' to K'' so that for every production containing a_m another copy is made throughout of which a_m is replaced by a_{m+1}. Clearly, K'' is again an interpretation of K and $L(K'') = \{a_1, \cdots, a_{m+1}\}$.

That F_2 is good follows from above assertion readily. For suppose K is an EOL form with $\mathscr{L}(K) \subseteq \mathscr{L}(F_2)$, i.e. $\mathscr{L}(K) \subseteq \mathscr{L}_{\text{Symb}}$. Then $\mathscr{L}(K) = \mathscr{L}_k$ for some $k > 0$. Thus, $\mathscr{L}(K) = \mathscr{L}(H_k)$ where H_k is given by the productions $S \to a_i$, $a_i \to a_i$ for $1 \leqq i \leqq k$. Noting that H_k is an interpretation of F_2 completes the proof.

It should be clear that even for rather simple EOL forms it is, in general, not easy to determine directly whether they are good or bad. In what follows we will develop methods which will ease this kind of task. In particular, the next two theorems permit us to show that a form is good [bad] by transforming it in a certain way into a form for which this property is already known.

THEOREM 2.2. *Let F and $\bar{F}$ be arbitrary EOL forms with $\mathscr{L}(F) = \mathscr{L}(\bar{F})$ and $\bar{F} \lhd F$. Then $\bar{F}$ good implies that F is good.*

Proof. Consider an arbitrary H with $\mathscr{L}(H) \subseteq \mathscr{L}(F)$. Then $\mathscr{L}(H) \subseteq \mathscr{L}(F) = \mathscr{L}(\bar{F})$, and since $\bar{F}$ is known to be good, for some $\bar{F}' \lhd \bar{F}$ we have $\mathscr{L}(\bar{F}') = \mathscr{L}(H)$. But $\bar{F}' \lhd \bar{F} \lhd F$. Thus $\mathscr{L}(H) = \mathscr{L}(\bar{F}')$ for some interpretation $\bar{F}'$ of F, i.e. F is good.

Above theorem says that—provided the language families are the same—"inverse interpretation" preserves the property good, and interpretation preserves the property bad. We will see later that the converse of Theorem 2.2 does not hold and that the condition $\mathscr{L}(F) = \mathscr{L}(\bar{F})$ is essential.

THEOREM 2.3. *Let F and $\bar{F}$ be arbitrary EOL forms with $\mathscr{L}(F) = \mathscr{L}(\bar{F})$ such that F nt-simulates $\bar{F}$. Then $\bar{F}$ good implies that F is good.*

Proof. Consider an arbitrary EOL form H with $\mathscr{L}(H) \subseteq \mathscr{L}(F) = \mathscr{L}(\bar{F})$. Since $\bar{F}$ is good, for some $\bar{F}'$ with $\bar{F}' \lhd \bar{F}(\bar{\mu})$ we have $\mathscr{L}(\bar{F}') = \mathscr{L}(H)$. We have to show that for some $F' \lhd F(\mu)$ indeed $\mathscr{L}(F') = \mathscr{L}(H)$. We proceed by constructing an EOL form $F' \lhd F$ and then show that $\mathscr{L}(F') = \mathscr{L}(\bar{F}') = \mathscr{L}(H)$ holds.

Let $F = (V, \Sigma, P, S)$, $\bar{F} = (\bar{V}, \bar{\Sigma}, \bar{P}, \bar{S})$, $\bar{F}' = (\bar{V}', \bar{\Sigma}', \bar{P}', \bar{S}')$ and suppose that for some $l \geqq 1$ we have $\alpha \underset{\text{nt } F}{\overset{l}{\Longrightarrow}} x$ for each $\alpha \to x$ of $\bar{P}$. We construct $F' = (V', \Sigma', P', S')$ by putting into P' for each production q of $\bar{F}'$ a set of production as follows.

Suppose q is the production $\beta \to y$. Then there exists a production $\alpha \to x$ of $\bar{F}$ such that $\beta \in \bar{\mu}(\alpha)$, $y \in \bar{\mu}(x)$. Thus $\alpha \underset{\text{nt } F}{\overset{l}{\Longrightarrow}} x$ holds and thus there is a sequence of words $\alpha = x_0, x_1, x_2, \cdots, x_{l-1}, x_l = x$ such that $x_i \underset{F}{\Longrightarrow} x_{i+1}$ and such that

$$S \underset{F}{\overset{*}{\Longrightarrow}} y_0 x_0 z_0 \underset{F}{\Longrightarrow} y_1 x_1 z_1 \underset{F}{\Longrightarrow} \cdots \underset{F}{\Longrightarrow} y_{l-1} x_{l-1} z_{l-1} \underset{F}{\Longrightarrow} y_l x_l z_l$$

implies that $y_i x_i z_i$ $(1 \leqq i \leqq l-1)$ contains at least one nonterminal. Write

$$\begin{aligned} x_1 &= \alpha_{1,1}\alpha_{1,2} \cdots \alpha_{1,t_1}, \\ x_2 &= \alpha_{2,1}\alpha_{2,2} \cdots \alpha_{2,t_2}, \qquad (\alpha_{i,j} \in V) \\ &\vdots \\ x_{l-1} &= \alpha_{l-1,1}\alpha_{l-1,2} \cdots \alpha_{l-1,t_{l-1}}, \end{aligned}$$

introduce $t_1 + t_2 + \cdots + t_{l-1}$ new symbols $\alpha_{i,j}^{(q)}$, where $\alpha_{i,j}^{(q)}$ is a terminal symbol iff $\alpha_{i,j}$ is

a terminal, and let $x_i^{(q)}$ be the word x_i with $\alpha_{i,j}$ replaced by $\alpha_{i,j}^{(q)}$. Suppose the set of productions of F involved in $\alpha \Longrightarrow x_1 \Longrightarrow x_2 \Longrightarrow \cdots \Longrightarrow x_{l-1} \Longrightarrow x$ is Q. Put into P' of F' exactly those productions (each of them is the interpretation of a production of Q) required for the derivation $\beta \underset{F'}{\Longrightarrow} x_1^{(q)} \underset{F'}{\Longrightarrow} x_2^{(q)} \underset{F'}{\Longrightarrow} \cdots \underset{F'}{\Longrightarrow} x_{l-1}^{(q)} \Longrightarrow y$. By construction it is clear that $F' \lhd F$.

Since F' nt-simulates $\bar{F}'$, Lemma 3 of [2] yields $\mathscr{L}(\bar{F}') \subseteq \mathscr{L}(F')$. Since for an l-symbol α of F', $\alpha \overset{l}{\underset{F'}{\Longrightarrow}} x$ implies $\alpha \overset{l}{\underset{\text{nt } F'}{\Longrightarrow}} x$, and (since there is one possible derivation leading from α to x in l steps) indeed $\alpha \overset{l}{\underset{\text{tnt } F'}{\Longrightarrow}} x$, and $\alpha \to x$ is a production of $\bar{F}$, Lemma 3.4 of [2] gives $\mathscr{L}(F') \subseteq \mathscr{L}(\bar{F}')$ and the proof is complete.

We will see later that the converse of Theorem 2.3 does not hold and that the condition $\mathscr{L}(F) = \mathscr{L}(\bar{F})$ is essential.

Our next aim is to establish the existence of good and complete (i.e. vomplete) EOL forms, answering an open problem of [2]. To this end we first prove two auxiliary results and a theorem of interest in itself since it provides a further transformation which preserves the property good.

Throughout the rest of this paper let maxr (F) for an arbitrary EOL form F be the length of the longest righthand side of any production of F.

LEMMA 2.1. *For every EOL form F with* maxr $(F) = m \geqq 3$ *a form equivalent EOL form $\bar{F}$ with* maxr $(\bar{F}) = m - 1$ *can be constructed; and if F is good, so is $\bar{F}$.*

Proof. Let $F = (V, \Sigma, P, S)$ and construct $\bar{F} = (\bar{V}, \Sigma, \bar{P}, S)$ as follows:

(a) for every production $p: \alpha \to x$ of P with $|x| \leqq 2$ put into $\bar{P}$ the two productions $\alpha \to [p]$ and $[p] \to x$, where $[p]$ is a new nonterminal,

(b) for every production $p: \alpha \to x$ of P with $|x| \geqq 3$ and $x = \beta y$, $\beta \in V$, $y \in V^*$ take into $\bar{P}$ the three productions $\alpha \to [\bar{p}][\bar{\beta}]$, $[\bar{p}] \to \beta$, $[\bar{\beta}] \to y$, where $[\bar{p}]$, $[\bar{\beta}]$ are new nonterminals.

Clearly, maxr $(\bar{F}) = m - 1$ and $\bar{F}$ nt-simulates F, i.e. $\mathscr{L}(F) \subseteq \mathscr{L}(\bar{F})$. If α is a 2-symbol of $\bar{F}$ then $\alpha \overset{2}{\underset{\bar{F}}{\Longrightarrow}} x$ implies $\alpha \overset{2}{\underset{\text{tnt } \bar{F}}{\Longrightarrow}} x$ and $\alpha \to x \in P$, i.e. $\mathscr{L}(\bar{F}) \subseteq \mathscr{L}(F)$. Since $\mathscr{L}(F) = \mathscr{L}(\bar{F})$ Theorem 2.3 applies. Thus $\bar{F}$ is good if F is good, as desired.

Applying Lemma 2.1 repeatedly we obtain

LEMMA 2.2. *For every EOL form F a short, form equivalent EOL form $\bar{F}$ can be constructed, such that $\bar{F}$ is good if F is good.*

Proof. The proof is clear.

THEOREM 2.4. *For every EOL form F a binary form equivalent EOL form $\bar{F}$ can be constructed such that $\bar{F}$ is good if F is good.*

Proof. By Lemma 2.2 we may assume that $F = (V, \Sigma, P, S)$ is short. Construct $\bar{F} = (\bar{V}, \Sigma, \bar{P}, S)$ as follows:

(a) for every production $p: \alpha \to \beta\gamma$ with α, β, γ in V put into $\bar{P}$ the four productions: $\alpha \to [p]$, $[p] \to [\bar{p}][\bar{\beta}]$, $[\bar{p}] \to \beta$, $[\bar{\beta}] \to \gamma$, where $[p]$, $[\bar{p}]$, $[\bar{\beta}]$ are new nonterminals,

(b) for every production $p: \alpha \to x$ with $|x| \leqq 1$ put into $\bar{P}$ the three productions: $\alpha \to [p]$, $[p] \to [\bar{p}]$, $[\bar{p}] \to x$, where $[p]$ and $[\bar{p}]$ are new nonterminals.

Note that $\bar{F}$ is binary and nt-simulates F, i.e. $\mathscr{L}(F) \subseteq \mathscr{L}(\bar{F})$. If α is a 3-symbol of $\bar{F}$ then $\alpha \overset{3}{\underset{\bar{F}}{\Longrightarrow}} x$ implies $\alpha \overset{3}{\underset{\text{tnt } \bar{F}}{\Longrightarrow}} x$ and $\alpha \to x \in P$, i.e. $\mathscr{L}(\bar{F}) \subseteq \mathscr{L}(F)$. By Theorem 2.3, $\bar{F}$ is good, if F is good. It is now easy to see that there are good and complete (i.e. vomplete) EOL forms.

COROLLARY 2.1. *The EOL form* $F=(\{S, a\}, \{a\}, P, S)$ *with productions* $S \to \{\varepsilon, a, S, SS\}$, $a \to S$ *is vomplete.*

Proof. Let H be an arbitrary EOL form. By Theorem 2.4 a binary EOL form $\bar{F}$ exists such that $\mathscr{L}(H)=\mathscr{L}(\bar{F})$. $\bar{F}$ is clearly an interpretation of F.

Corollary 2.1 raises the question whether the production $S \to \varepsilon$ is actually required for vompleteness. The following lemma will allow us to establish that productions with ε on the righthand side must occur in any good form except under very special circumstances.

LEMMA 2.3. *No propagating EOL form is form equivalent to the EOL form* $F=(\{S, a, b, c, d\}, \{a, b, c, d\}, P, S)$ *with productions* $S \to aba$, $a \to cd$, $b \to \varepsilon$, $c \to c$, $d \to d$.

Proof. $L=L(F)=\{aba, cdcd\}$. Note that each language in $\mathscr{L}(F)$ contains at least two words, one of length 3 and one of length 4. Assume some propagating $\bar{F}$ is form equivalent to F. Hence there exists an $\bar{F}' \lhd \bar{F}$ such that $L(\bar{F}')=L$. Let $\bar{F}'=(\bar{V}', \bar{\Sigma}', \bar{P}', \bar{S}')$ and consider the derivation

$$D: \bar{S}'=\bar{x}_0' \underset{\bar{F}'}{\Longrightarrow} \bar{x}_1' \underset{\bar{F}'}{\Longrightarrow} \bar{x}_2' \Longrightarrow \cdots \underset{\bar{F}'}{\Longrightarrow} \bar{x}_n'=cdcd.$$

Since F' is propagating, $cdcd \overset{*}{\underset{\bar{F}'}{\Longrightarrow}} aba$ is impossible. No terminal word other than aba can occur in the derivation D since no such terminal word is in L. If aba does not occur in D, and thus no terminal word of length 3 occurs in D, we can easily construct an interpretation $\bar{\bar{F}}'$ of $\bar{F}'$ (and thus of $\bar{F}$) such that $L(\bar{\bar{F}}')$ does not contain any word of length 3 as follows: rename the symbols in $\bar{x}_0', \bar{x}_1', \cdots, \bar{x}_n'$ such that words $\bar{\bar{S}}'=\bar{\bar{x}}_0', \bar{\bar{x}}_1', \cdots, \bar{\bar{x}}_n'$ are obtained with the property that the word $\bar{\bar{x}}_0'\bar{\bar{x}}_1' \cdots \bar{\bar{x}}_{n-1}'$ contains no symbol more than once. Define the EOL form $\bar{\bar{F}}'=(\bar{\bar{V}}', \bar{\bar{\Sigma}}', \bar{\bar{P}}', \bar{\bar{S}}')$ by taking $\bar{\bar{P}}'=\mathring{P} \cup \hat{P}$, where $\hat{P}$ contains exactly those productions obtained from productions of $\bar{P}'$ by the renaming required for the derivation $\bar{\bar{S}}'=\bar{\bar{x}}_0' \Longrightarrow \bar{\bar{x}}_1' \Longrightarrow \cdots \Longrightarrow \bar{\bar{x}}_{n-1}' \Longrightarrow cdcd$, and where $\mathring{P}$ contains exactly those productions of $\bar{P}'$ required to continue the derivation past $cdcd$. Evidently, $\bar{\bar{F}}' \lhd \bar{F}' \lhd \bar{F}$ and $L(\bar{\bar{F}}')$ does indeed contain no terminal word of length 3, a contradiction. Thus abc must occur in the derivation D, i.e.

$$\bar{S}' \overset{*}{\underset{\bar{F}'}{\Longrightarrow}} aba \overset{m}{\underset{\bar{F}'}{\Longrightarrow}} cdcd.$$

Thus we have either

(i) $a \overset{m}{\Longrightarrow} c$, $b \overset{m}{\Longrightarrow} d$, $a \overset{m}{\Longrightarrow} cd$ or

(ii) $a \overset{m}{\Longrightarrow} c$, $b \overset{m}{\Longrightarrow} dc$, $a \overset{m}{\Longrightarrow} d$ or

(iii) $a \overset{m}{\Longrightarrow} cd$, $b \overset{m}{\Longrightarrow} c$, $a \overset{m}{\Longrightarrow} d$.

In case (i),

$$aba \overset{m}{\underset{\bar{F}'}{\Longrightarrow}} cddcd;$$

in case (ii),

$$aba \overset{m}{\underset{\bar{F}'}{\Longrightarrow}} ddcc$$

and in case (iii),

$$aba \overset{m}{\underset{\bar{F}'}{\Longrightarrow}} cdccd.$$

Since neither of above three words is in L, $L(\bar{F}') \neq L$ in each case, we have a contradiction.

THEOREM 2.5. *Let F be the EOL form of Lemma 2.3. No propagating EOL form H with $\mathscr{L}(F) \subseteq \mathscr{L}(H)$ is good.*

Proof (by contradiction). If indeed H is good, for some $H' \lhd H$, H' propagating, $\mathscr{L}(F) = \mathscr{L}(H')$, in contradiction to Lemma 2.3.

We obtain the immediate corollary:

COROLLARY 2.2. *No propagating EOL form is vomplete.*

Corollary 2.2 has interesting ramifications: while the EOL form F with productions $S \to \{\varepsilon, a, S, SS\}$, $a \to S$ is vomplete by Corollary 2.1 the EOL form $\bar{F}$ with the same productions except that $S \to \varepsilon$ is missing is not vomplete by Corollary 2.2. (It is of course well known that both F and $\bar{F}$ are complete.) Note that $\bar{F} \lhd F$ (and F nt-simulates $\bar{F}$), $\mathscr{L}(F) = \mathscr{L}(\bar{F})$, F is good, but $\bar{F}$ is bad, i.e. neither the converse of Theorem 2.2 nor of Theorem 2.3 is true. Also note at this point that the condition $\mathscr{L}(F) = \mathscr{L}(\bar{F})$ is critical in both Theorem 2.2 and Theorem 2.3. Just consider F with productions: $S \to \{a, S, SS, B\}$, $a \to S$, $B \to b$, $b \to C$, $C \to b$ and $\bar{F}$ with productions $S \to B$, $B \to b$, $b \to C$, $B \to b$ (where a, b are terminals, S, B, C are nonterminals). $\bar{F}$ is easily seen to be good (based on the fact that F_2 of Theorem 2.1 is good, and on the construction under (a) in the proof of Lemma 2.1), F is clearly complete (and thus bad since it is propagating) but both $\bar{F} \lhd F$ and F nt-simulates $\bar{F}$.

In [2] it is shown that no synchronized EOL form is vomplete. We now generalize this result considerably.

DEFINITION. An EOL form $F = (V, \Sigma, P, S)$ is called *k-synchronized* ($k \geqq 1$) if $S \overset{+}{\Longrightarrow} x_1 \overset{+}{\Longrightarrow} x_2 \overset{+}{\Longrightarrow} \cdots \overset{+}{\Longrightarrow} x_k \overset{+}{\Longrightarrow} y$, $x_i \in \Sigma^*$, implies $y \notin \Sigma^*$. Note that 1-synchronized is the usual concept synchronized.

THEOREM 2.6. *No k-synchronized EOL form F for which $L(F)$ contains at least one nonempty word is good.*

Proof (by contradiction). Suppose $F = (V, \Sigma, P, S)$ is k-synchronized and good and for some $x \in \Sigma^+$, $x \in L(F)$. We will construct an $\bar{F}$ with $\mathscr{L}(\bar{F}) \subseteq \mathscr{L}(F)$ such that for every interpretation F'' of F, $\mathscr{L}(F'') \neq \mathscr{L}(\bar{F})$, a contradiction.

For each $x_1 \in L(F)$ define

$$M(x_1) = \max \{t \mid x_1 \overset{+}{\underset{F}{\Longrightarrow}} x_2 \overset{+}{\underset{F}{\Longrightarrow}} \cdots \overset{+}{\underset{F}{\Longrightarrow}} x_t, x_i \in \Sigma^*\}$$

and let $t = \max \{M(x_1) \mid x_1 \in L(F)\}$. Then there exists a derivation $S = x_0 \underset{F}{\Longrightarrow} x_1 \underset{F}{\Longrightarrow} x_2 \underset{F}{\Longrightarrow} \cdots \underset{F}{\Longrightarrow} x_n$ containing exactly t terminal words $x_{i_1}, x_{i_2}, \cdots, x_{i_t}$ and $x_n \overset{+}{\Longrightarrow} y$ implies $y \notin \Sigma^*$. By renaming all symbols in $x_0, x_1, \cdots, x_n$ words $w_0, w_1, \cdots, w_n$ can be obtained such that in $w_0 w_1 \cdots w_n$ no symbol occurs more than once. Clearly, an interpretation $\hat{F} = (\hat{V}, \hat{\Sigma}, \hat{P}, \hat{S})$ of F can be constructed such that $\hat{F}$ is deterministic,

$$\hat{S} = w_0 \underset{\hat{F}}{\Longrightarrow} w_1 \underset{\hat{F}}{\Longrightarrow} \cdots \underset{\hat{F}}{\Longrightarrow} w_n \text{ holds}, \qquad w_n \overset{+}{\Longrightarrow} y \text{ implies } y \notin \hat{\Sigma}^*$$

and $w_{i_1}, w_{i_2}, \cdots, w_{i_t}$ are the only terminal words in $L(\hat{F})$.

Define an EOL form $\bar{F} = (\bar{V}, \Sigma, \bar{P}, \bar{S})$ as follows: Put exactly those productions of $\hat{P}$ into $\bar{P}$ which are required in the derivation $\hat{S} \overset{*}{\underset{\hat{F}}{\Longrightarrow}} w_{i_1} \overset{*}{\underset{\hat{F}}{\Longrightarrow}} \cdots \overset{*}{\underset{\hat{F}}{\Longrightarrow}} w_{i_{t-1}} \overset{*}{\underset{\hat{F}}{\Longrightarrow}} u \underset{\hat{F}}{\Longrightarrow} w_{i_t}$. Call the set of all productions used in this derivation up to the last but one step $\tilde{P}$, and let $\tilde{\tilde{P}}$ be the set of productions used in the last step $u \underset{\hat{F}}{\Longrightarrow} w_{i_t}$. For each terminal a occurring in w_{i_t} put the productions $a \to \bar{a}$, $\bar{a} \to N$, $N \to N$ ($\bar{a}$ a new terminal, N a nonterminal) into $\bar{P}$.

It is of crucial importance to observe that $\mathscr{L}(\bar{F}) \subseteq \mathscr{L}(\hat{F})$. This is seen as follows. Let $\bar{F}' = (\bar{V}', \bar{\Sigma}', \bar{P}', \bar{S}')$ be an arbitrary interpretation of $\bar{F}$, $\bar{F}' \lhd \bar{F}(\bar{\mu})$. We construct an interpretation $\hat{F}' = (\hat{V}', \hat{\Sigma}', \hat{P}', \bar{S}')$ of $\hat{F}$, $\hat{F}' \lhd \hat{F}(\hat{\mu})$ and demonstrate $L(F') = L(\hat{F}')$. Define $\hat{\mu}$ as follows:

$$\hat{\mu}(\alpha) = \begin{cases} \bar{\mu}(\alpha) \cup \bar{\mu}(\alpha) & \text{for each } \bar{\alpha} \text{ in } w_{i_t}, \\ \bar{\mu}(\alpha) & \text{for each } \alpha \text{ occurring in a production of } \tilde{P}, \\ \{\alpha\} & \text{for all other symbols.} \end{cases}$$

Put into P' all productions of $\bar{P}'$ except those steming from productions $a \to \bar{a}$ (a in w_{i_t}), $\bar{a} \to N$, and $N \to N$. Further, for each production $p\colon \alpha \to a_1 a_2 \cdots a_r$ of $\bar{P}'$ which is an interpretation of a production of $\tilde{\tilde{P}}$, let $N(a_i) = \{b \mid a_i \to b \in \bar{P}'\}$ and put into $\hat{P}'$ all productions $\alpha \to N(a_1)N(a_2) \cdots N(a_r)$. Put also into $\hat{P}'$ all such interpretations of productions of $\hat{P}$ needed to insure that for every $\alpha \in \hat{V}'$ a production with left side α exists. It is readily seen that $L(\bar{F}') = L(\hat{F}')$.

For a derivation

$$\bar{S}' \overset{*}{\Longrightarrow} z_1 \overset{*}{\Longrightarrow} \cdots \overset{*}{\Longrightarrow} z_t \overset{*}{\Longrightarrow} z_{t+1}$$

in $\bar{F}'$ involving the terminal words $z_1, z_2, \cdots, z_{t+1}$ only (and no further terminal word derivable from z_{t+1}) we have the two derivations

$$\bar{S}' \overset{*}{\Longrightarrow} z_1 \overset{*}{\Longrightarrow} \cdots \overset{*}{\Longrightarrow} z_{t-1} \overset{*}{\Longrightarrow} z_t$$

and

$$\bar{S}' \overset{*}{\Longrightarrow} z_1 \overset{*}{\Longrightarrow} \cdots \overset{*}{\Longrightarrow} z_{t-1} \overset{*}{\Longrightarrow} z_{t+1}$$

in $\hat{F}'$ (and no further terminal word is derivable from z_t or z_{t+1} or occurs in between the z_i's).

Conversely, for any derivation

$$\bar{S}' \overset{*}{\Longrightarrow} z_1 \overset{*}{\Longrightarrow} \cdots \overset{*}{\Longrightarrow} z_{t-1} \overset{*}{\Longrightarrow} x$$

in $\hat{F}'$ involving only the terminal words $z_1, z_2, \cdots, z_{t-1}, x$ and no further terminal word derivable from x there exists the derivation

$$\bar{S} \overset{*}{\Longrightarrow} z_1 \overset{*}{\Longrightarrow} \cdots \overset{*}{\Longrightarrow} z_{t-1} \overset{*}{\Longrightarrow} z_t \overset{*}{\Longrightarrow} z_{t+1}$$

in $\bar{F}'$, where either $x = z_t$ or $x = z_{t+1}$.

We have thus demonstrated $\mathscr{L}(\bar{F}) \subseteq \mathscr{L}(\hat{F})$ and thus $\mathscr{L}(\bar{F}) \subseteq \mathscr{L}(F)$. We note that every language in $\mathscr{L}(\bar{F})$ contains at least $t+1$ words.

Take an arbitrary interpretation $F'' = (V'', \Sigma'', P'', S'')$ of F with $L(F'')$ containing at least one nonempty word. We show $\mathscr{L}(\bar{F}) \neq \mathscr{L}(F'')$. In F'' a derivation $S_0'' = x'' \Longrightarrow x_1'' \Longrightarrow x_2'' \Longrightarrow \cdots \Longrightarrow x_n''$ exists such that at most $m \leqq t$ of the words $x_0'', \cdots, x_n''$ are terminal and $x_n'' \overset{+}{\underset{F''}{\Longrightarrow}} y$ implies $y \notin \Sigma''^*$. By renaming all occurrences of symbols in all x_i an interpretation F''' of F'' can be obtained such that $L(F''')$ consists of $m \leqq t$ words. Thus $L(F''') \in \mathscr{L}(F'')$ but $L(F''') \notin \mathscr{L}(\bar{F})$, completing the proof.

Note that part 1 of Theorem 2.1 is a simple corollary to Theorem 2.6.

It is interesting that as far as language families generated by EOL forms are concerned, synchronization seems to be a drawback rather than an advantage as usual in customary L systems theory.

While the current paper does provide some insight into the question of when an EOL form is good or bad, many open questions remain. We are, for example, unable to establish whether the (complete) EOL form $\bar{\bar{F}}$ with productions $S \to a$, $S \to S$, $S \to SS$, $a \to S$, $a \to \varepsilon$ is good or bad.

Acknowledgment. The authors are indebted to the referees for their comments and suggestions, especially for the shortening of the proof of Theorem 2.1.

REFERENCES

[1] S. GINSBURG, *A survey of context-free grammar forms*, Formal Languages and Programming, North-Holland, Amsterdam, 1976.

[2] H. A. MAURER, A. SALOMAA AND D. WOOD, *EOL forms*, Acta Informat., 8 (1977), pp. 75–96.

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

LINEAR LANGUAGES AND THE INTERSECTION CLOSURES OF CLASSES OF LANGUAGES*

RONALD V. BOOK† AND MAURICE NIVAT‡

Abstract. The intersection closure and the closure under homomorphic replication and intersection of certain classes of languages are studied and related to a specific class $\mathscr{L}_{\text{BNP}}$ of languages defined in [5]. The proof techniques rely on the "set-theoretic algebra" of language theory instead of arguments involving abstract families of acceptors.

Key words. linear context-free languages, intersection closure, homomorphic replication, linear erasing

Introduction. The class of linear context free languages has been a fruitful source of examples and counterexamples in formal language theory throughout the development of the subject. Recently this class has been used in establishing new results concerning the structure of some important classes of languages [1], [3], [5], [9], [10], [12], [13]. In this paper the class of linear context-free languages is used to obtain new results concerning the closure of certain classes of languages under the operations of intersection, linear-erasing homomorphism, and linear-erasing homomorphic replication.

There has been a great deal of research activity in formal language theory aimed at studying classes of languages specified in terms of certain closure properties and the relationship between these classes and those specified by abstract automata behaving in some specified manner. This research has lead to the study of abstract families of languages and abstract families of acceptors [6], [7], [8], [11]. In particular certain classes of languages recognized by multitape acceptors have been characterized in terms of classes of languages closed under intersection [11]. However, the results obtained in the abstract setting have not been particularly useful when specific classes have been studied. One reason for this is that proofs involving abstract families of acceptors (as in [6], [7], [8], [11]) are quite technical.

Here we present results similar to those in [11] but we do not make reference to abstract families of acceptors. Instead we depend heavily on the properties of one specific class of languages, the class $\mathscr{L}_{\text{BNP}}$ of languages accepted in linear time by nondeterministic multitape Turing machines whose read-write heads are reversal-bounded [5]. It is known [5] that a language is in $\mathscr{L}_{\text{BNP}}$ if and only if it is accepted in real time by a nondeterministic machine with just three pushdown stores as storage which operates in such a way that in every computation each push-down store makes at most one reversal, if and only if it is the nonerasing homomorphic image of the intersection of just three linear context-free languages. Thus, in considering machine specifications of languages in $\mathscr{L}_{\text{BNP}}$ one can reduce "multitape" to just "three push-down stores" and in characterizing $\mathscr{L}_{\text{BNP}}$ in terms of closure operations it is enough to consider the intersection of just three linear context-free languages instead of the intersection closure of the class of linear context-free languages. Recall that similar characterizations of the class of quasi-real time languages were established in [4].

* Received by the editors May 1 1977, and in revised form September 29, 1977. This research was supported in part by the National Science Foundation under Grants MCS76-05744 and MCS77-11360. Some of these results were announced at the John Hopkins Symposium on Information Science and Systems in March, 1977.

† Department of Mathematics and Computer Science Program, University of California, Santa Barbara, California 93106.

‡ U.E.R. de Mathématiques, Université Paris VII, 75005 Paris, France.

The proof techniques in this paper rely on the "set-theoretic algebra" of formal language theory as represented by results in [2], [3], [6], [7], [14]. In § 2 we study classes of languages with some simple properties that are not necessarily abstract families of languages and consider the intersection closures of these classes by relating them to classes obtained by combining $\mathscr{L}_{\text{BNP}}$ and the given classes. In § 3 we carry this study forward by characterizing the closure of a class under intersection and linear-erasing homomorphic replication. The results in § 3 also represent an attempt to capture the technique of "tape-folding" used in the programming of Turing machines in [4], [5] in terms of algebraic operations.

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: if e is the empty string, then $|e|=0$; if a is a symbol and y is a string, then $|ay|=1+|y|$.

The *reversal* w^{R} of a string w is the string obtained by writing w in reverse order: $e^{\text{R}}=e$; if a is a symbol and y is a string, then $(ay)^{\text{R}}=y^{\text{R}}a$.

Recall that a *homomorphism* (between free monoids) is a function $h\colon \Sigma^*\to\Delta^*$ such that for all $x, y\in\Sigma^*$, $h(xy)=h(x)h(y)$. A homomorphism $h\colon \Sigma^*\to\Delta^*$ is *nonerasing* if $|w|>0$ implies $|h(w)|>0$ and is *length-preserving* if for all $w\in\Sigma^*$, $|h(w)|=|w|$. A homomorphism $h\colon \Sigma^*\to\Delta^*$ is *linear-erasing on language* $L\subseteq\Sigma^*$ if there is a constant $k>0$ such that for all $w\in L$ with $|w|\geqq k$, $|w|\leqq k|h(w)|$.

A class $\mathscr{L}$ of languages is *closed under* (*nonerasing, linear-erasing*) *homomorphism* if for every language $L\in\mathscr{L}$ and any homomorphism h (that is nonerasing, linear-erasing on L) $h(L)=\{h(w)|w\in L\}$ is in $\mathscr{L}$.

The research leading to the results presented here was motivated by the authors' investigation [5] of the class of languages accepted in real time by nondeterministic multitape Turing acceptors whose storage tape heads are "reversal-bounded." Recall that a "reversal" is a change in direction of a read-write head's motion; for a pushdown store this means a change from "popping" to "pushing" or vice versa. A machine is *reversal-bounded* if there is a fixed constant k such that in every computation each read-write head makes at most k reversals.

The following result is Theorem 3.1 of [5].

PROPOSITION 1.1. *Let L be a language. The following are equivalent*:

(i) *L is accepted in linear time by a nondeterministic reversal-bounded multipushdown acceptor*;

(ii) *L is accepted in real time by a nondeterministic multipushdown acceptor which operates in such a way that in every accepting computation each pushdown store makes at most one reversal*;

(iii) *L is the length-preserving homomorphic image of the intersection of some finite number of linear context-free languages*;

(iv) *L is accepted in real time by a nondeterministic acceptor with three pushdown stores which operates in such a way that in every computation each pushdown store makes at most one reversal*;

(v) *L is the length-preserving homomorphic image of the intersection of three linear context-free languages.*

The class of languages described in Proposition 1.1 will be referred to here as $\mathscr{L}_{\text{BNP}}$.

From the characterizations of the class $\mathscr{L}_{\text{BNP}}$ given in Proposition 1.1, it is easy to show the following result.

PROPOSITION 1.2.

(i) *The class $\mathscr{L}_{\mathrm{BNP}}$ contains the linear context-free languages and is closed under union, intersection, inverse homomorphism, concatenation, linear-erasing homomorphism, and reversal.*

(ii) *The class $\mathscr{L}_{\mathrm{BNP}}$ is the smallest class containing the linear context-free languages and closed under intersection and nonerasing homomorphism.*

It is well known that the class of linear context-free languages is the smallest class containing the language $\{wcw^{\mathrm{R}} | w \in \{a, b\}^*\} \cup \{e\}$ and closed under inverse homomorphism, nonerasing homomorphism, and intersection with regular sets. From this fact and Proposition 1.2, one obtains the following characterization of $\mathscr{L}_{\mathrm{BNP}}$.

PROPOSITION 1.3. *The class $\mathscr{L}_{\mathrm{BNP}}$ is the smallest class containing the language $\{wcw^{\mathrm{R}} | w \in \{a, b\}^*\} \cup \{e\}$ and closed under intersection with regular sets, inverse homomorphism, nonerasing homomorphism, and intersection.*

The properties of the class $\mathscr{L}_{\mathrm{BNP}}$ described in Proposition 1.2 will be quite useful in establishing the results in §§ 2 and 3. In addition certain set-theoretic notation will be of value.

Notation. Let $\mathscr{C}$ be a class of languages. Let $H(\mathscr{C}) = \{h(L) | h$ is a nonerasing homomorphism and $L \in \mathscr{C}\}$, let $H^{-1}(\mathscr{C}) = \{h^{-1}(L) | h$ is a homomorphism and $L \in \mathscr{C}\}$, and let $H_{\mathrm{lin}}(\mathscr{C}) = \{h(L) | h$ is a homomorphism that is linear-erasing on L and $L \in \mathscr{C}\}$. For classes $\mathscr{C}_1$ and $\mathscr{C}_2$ of languages, let $\mathscr{C}_1 \bigwedge \mathscr{C}_2 = \{L_1 \cap L_2 | L_i \in \mathscr{C}_i, i = 1, 2\}$. For a class $\mathscr{C}$ of languages, the closure of $\mathscr{C}$ under intersection is $\bigwedge \mathscr{C} = \{L_1 \cap \cdots \cap L_k | k \geqq 1$, each $L_i \in \mathscr{C}, 1 \leqq i \leqq k\}$.

To illustrate the use of this notation, let us restate some of the abovementioned properties of $\mathscr{L}_{\mathrm{BNP}}$. For this purpose let LIN denote the class of all linear context-free languages.

(i) $\mathscr{L}_{\mathrm{BNP}}$ is closed under intersection: $\mathscr{L}_{\mathrm{BNP}} \bigwedge \mathscr{L}_{\mathrm{BNP}} \subseteq \mathscr{L}_{\mathrm{BNP}}$ and $\bigwedge \mathscr{L}_{\mathrm{BNP}} \subseteq \mathscr{L}_{\mathrm{BNP}}$.

(ii) $\mathscr{L}_{\mathrm{BNP}}$ is closed under inverse homomorphism: $H^{-1}(\mathscr{L}_{\mathrm{BNP}}) \subseteq \mathscr{L}_{\mathrm{BNP}}$.

(iii) $\mathscr{L}_{\mathrm{BNP}}$ is closed under linear-erasing homomorphism: $H_{\mathrm{lin}}(\mathscr{L}_{\mathrm{BNP}}) \subseteq \mathscr{L}_{\mathrm{BNP}}$.

(iv) A language is in $\mathscr{L}_{\mathrm{BNP}}$ if and only if it is the nonerasing homomorphic image of the intersection of some finite number of linear context-free languages: $H(\bigwedge \mathrm{LIN}) = \mathscr{L}_{\mathrm{BNP}}$.

We will use this notation freely throughout this paper.

2. Intersection closure and closure under linear-erasing homomorphisms. We turn to the study of the intersection closure and the closure under linear-erasing homomorphisms of an arbitrary class of languages. We do not attempt to characterize these classes but rather to relate them to classes obtained by combining $\mathscr{L}_{\mathrm{BNP}}$ and the given class. The first results are concerned with the closure of a class under linear-erasing homomorphisms.

THEOREM 2.1. *Let $\mathscr{C}$ be a class of languages with the property that if $L \in \mathscr{C}$, then $L \cup \{e\} \in \mathscr{C}$. If $L \in \mathscr{C}$ and h is a homomorphism that is linear-erasing on L, then there exist homomorphisms f and g with f length-preserving, a language $L_1 \in \mathscr{C}$, and a language $L_2 \in \mathscr{L}_{\mathrm{BNP}}$ such that $h(L) = f(g^{-1}(L_1) \cap L_2)$. Thus, $H_{\mathrm{lin}}(\mathscr{C}) \subseteq H(H^{-1}(\mathscr{C}) \bigwedge \mathscr{L}_{\mathrm{BNP}})$.*

Proof. Let $L \in \mathscr{C}$ and let Σ be a finite alphabet such that $L \subseteq \Sigma^*$. Let $h: \Sigma^* \to \Delta^*$ be a homomorphism such that for some k and all $w \in L$, $|w| \leqq k \max(|h(w)|, 1)$, so that h is linear-erasing on L. If $e \in h(L)$, let $L_1 = L \cup \{e\}$; otherwise, let $L_1 = L$. Notice that $h(L_1) = h(L)$ and $L_1 \in \mathscr{C}$.

Let $\Gamma = \{[w, b] | w \in \Sigma^*, 1 \leqq |w| \leqq k, b \in \Delta\}$ be a set of new symbols. Let $g: \Gamma^* \to \Sigma^*$ be the homomorphism determined by defining $g([w, b]) = w$ for each $[w, b] \in \Gamma$, and let $f: \Gamma^* \to \Delta^*$ be the homomorphism determined by defining $f([w, b]) = b$ for each

$[w, b] \in \Gamma$. Notice that f is length-preserving. Let $L' = \{[w_1, b_1] \cdots [w_n, b_n] | n \geqq 1$, each $[w_i, b_i] \in \Gamma$, $h(w_1 \cdots w_n) = b_1 \cdots b_n\}$. If $e \in h(L)$, let $L_2 = L' \cup \{e\}$; otherwise, let $L_2 = L'$.

Notice that $(g^{-1}(L_1) \cap L_2) - \{e\} = \{[w_1, b_1] \cdots [w_n, b_n] | n \geqq 1$, each $[w_i, b_i] \in \Gamma$, $h(w_1 \cdots w_n) = b_1 \cdots b_n$, $w_1 \cdots w_n \in L_1 - \{e\}\}$ so that $f(g^{-1}(L_1) \cap L_2) = h(L_1) = h(L)$.

It suffices to show that L_2 is in $\mathscr{L}_{\text{BNP}}$. Let M be a deterministic Turing acceptor with a one-way read-only input tape and with two pushdown stores to be used as auxiliary storage tapes. Upon reading input $[w_1, b_1] \cdots [w_n, b_n]$ in Γ^*, M writes $h(w_1)h(w_2) \cdots h(w_n)$ on the first pushdown store and simultaneously writes $b_1 \cdots b_n$ on the second pushdown store. Having read the entire input, M empties the two pushdown stores simultaneously while attempting to match their contents. The input string $[w_1, b_1] \cdots [w_n, b_n]$ is accepted by M if and only if the two pushdown stores have the same contents, if and only if $h(w_1 \cdots w_n) = h(w_1) \cdots h(w_n) = b_1 \cdots b_n$. Thus, M accepts precisely the strings in L_2. If $m = \max\{|h_1(a)| \, | a \in \Sigma\}$, then M operates in time $2mn$ so that M operates in linear time. Clearly, each pushdown store makes only one reversal. Thus $L(M) = L_2$ is in $\mathscr{L}_{\text{BNP}}$. □

THEOREM 2.2. *Let $\mathscr{C}$ be a class of languages that is closed under inverse homomorphism and that has the property that if $L \in \mathscr{C}$, then $L \cup \{e\} \in \mathscr{C}$. Then the class $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under linear-erasing homomorphism, that is, $H_{\text{lin}}(H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})) \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$.*

Proof. If f and g are homomorphisms such that g is nonerasing and f is linear-erasing on $g(L)$ for some language L, then the result of composing f and g is a homomorphism that is linear-erasing on L. Thus, $H_{\text{lin}}(H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})) = H_{\text{lin}}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$.

The class $\mathscr{L}_{\text{BNP}}$ has the property that if $L \in \mathscr{L}_{\text{BNP}}$, then $L \cup \{e\} \in \mathscr{L}_{\text{BNP}}$, and by hypothesis, the class $\mathscr{C}$ has this property. Thus, the class $\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}$ also has this property. Hence, by Theorem 2.1, $H_{\text{lin}}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq H(H^{-1}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge \mathscr{L}_{\text{BNP}})$. For any sets X and Y and any function f, $f^{-1}(X \cap Y) = f^{-1}(X) \cap f^{-1}(Y)$, so that $H^{-1}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq H^{-1}(\mathscr{C}) \wedge H^{-1}(\mathscr{L}_{\text{BNP}})$. By hypothesis $\mathscr{C}$ is closed under inverse homomorphism, and (as noted in Proposition 1.2) $\mathscr{L}_{\text{BNP}}$ is closed under inverse homomorphism. Thus, $H^{-1}(\mathscr{C}) \wedge H^{-1}(\mathscr{L}_{\text{BNP}}) \subseteq \mathscr{C} \wedge \mathscr{L}_{\text{BNP}}$. Since $\mathscr{L}_{\text{BNP}}$ is closed under intersection, $\mathscr{L}_{\text{BNP}} \wedge \mathscr{L}_{\text{BNP}} \subseteq \mathscr{L}_{\text{BNP}}$ and so $(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge \mathscr{L}_{\text{BNP}} \subseteq \mathscr{C} \wedge \mathscr{L}_{\text{BNP}}$. Thus, $H_{\text{lin}}(H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})) = H_{\text{lin}}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ so that $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under linear-erasing homomorphism. □

COROLLARY. *Let $\mathscr{C}$ be a class of languages that is closed under inverse homomorphism and that has the property that if $L \in \mathscr{C}$, then $L \cup \{e\} \in \mathscr{C}$. Then the closure of $\mathscr{C}$ under linear-erasing homomorphism is included in the class $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$, that is, $H_{\text{lin}}(\mathscr{C}) \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$.*

Now we consider the intersection closure of a class of languages. Again we show a relationship between this closure and a class obtained by combining $\mathscr{L}_{\text{BNP}}$ and the original class.

THEOREM 2.3. *Let $\mathscr{C}$ be any class of languages that contains the language $\{1\}^*$ and all of the singleton sets and that is closed under concatenation and inverse homomorphism. For any $k \geqq 1$ and any choice of $L_1, \cdots, L_k \in \mathscr{C}$, there exist a language C in $\mathscr{C}$, linear context-free languages M_1 and M_2, and a homomorphism h such that $h(C \cap M_1 \cap M_2) = L_1 \cap \cdots \cap L_k$ and h is linear-erasing on $C \cap M_1 \cap M_2$. Thus, $\wedge\mathscr{C} \subseteq H_{\text{lin}}(\mathscr{C} \wedge \text{LIN} \wedge \text{LIN}) \subseteq H_{\text{lin}}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$.*

Proof. Let $L = L_1 \cap \cdots \cap L_k$ and let Σ be a finite alphabet such that $L \subseteq \Sigma^*$. Let Δ be a set of new symbols in one-to-one correspondence with Σ, say $\Delta = \{\bar{a} | a \in \Sigma\}$, $\Delta \cap \Sigma = \varnothing$, and let $\#$ be a new symbol, $\# \notin \Delta \cup \Sigma$. Let $\alpha: \Delta^* \to \Sigma^*$ be the homomorphism determined by defining $\alpha(\bar{a}) = a$ for each $\bar{a} \in \Delta$. Let $\beta: \Delta^* \to \{1\}^*$ be the homomorphism determined by defining $\beta(\bar{a}) = 1$ for each $\bar{a} \in \Delta$.

Let $C = L_1 \# \alpha^{-1}(L_2) \# \cdots \# \alpha^{-1}(L_k) \# (\beta^{-1}(\{1\}^*)\#)^k$. By hypothesis $\{\#\}$ and $\{1\}^*$ are in $\mathscr{C}$ and by choice $L_1, \cdots, L_k$ are in $\mathscr{C}$. Since $\mathscr{C}$ is closed under concatenation and inverse homomorphism, the language C is in $\mathscr{C}$. Notice that $C = \{w_1 \# y_2 \# \cdots y_k \# z_1 \# \cdots \# z_k \# | w_1 \in L_1, \alpha(y_i) \in L_i \text{ for } i \geqq 2, \text{ and } z_j \in \Delta^* \text{ for } j \geqq 1\}$.

Let $M_1 = \{w_1 \# y_2 \# \cdots \# y_k \# y_k^R \# \cdots \# y_2^R \# y_1^R | w_1 \in \Sigma^*, y_i \in \Delta^* \text{ for } i \geqq 1, \text{ and } \alpha(y_1) = w_1\}$ and let $M_2 = \{w_1 \# y_2 \# \cdots \# y_k \# z \# y_k^R \# \cdots \# y_2^R \# | w_1 \in \Sigma^*, y_i \in \Delta^* \text{ for } i \geqq 2, z \in \Delta^*\}$. Clearly both M_1 and M_2 are linear context-free languages.

It is easy to see that $C \cap M_1 \cap M_2 = \{w \# (\alpha^{-1}(w)\#)^{k-1} (\alpha^{-1}(w)^R \#)^k | w \in L_1 \cap \cdots \cap L_k = L\}$. Let $h: (\Sigma \cup \Delta \cup \{\#\})^* \to \Sigma^*$ be the homomorphism determined by defining

$$h(a) = \begin{cases} a & \text{for } a \in \Sigma, \\ e & \text{for } a \in \Delta \cup \{\#\}. \end{cases}$$

Then $h(C \cap M_1 \cap M_2) = L$. For any $x \in C \cap M_1 \cap M_2$, $|x| = 2k|h(x)| + 2k$ so that h is linear erasing on $C \cap M_1 \cap M_2$. □

By combining Theorems 2.2 and 2.3 we see that if $\mathscr{C}$ is a class of languages with suitable properties, then $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under both intersection and linear-erasing homomorphism. Thus we can relate the closure under intersection and linear-erasing homomorphism of an arbitrary class $\mathscr{C}$ of languages to a combination of $\mathscr{C}$ and $\mathscr{L}_{\text{BNP}}$.

THEOREM 2.4. *Let $\mathscr{C}$ be any class of languages such that* (i) *$\mathscr{C}$ contains the language $\{1\}^*$ and all of the singleton sets,* (ii) *$\mathscr{C}$ is closed under concatenation and inverse homomorphism, and* (iii) *$\mathscr{C}$ has the property that if $L \in \mathscr{C}$, then $L \cup \{e\} \in \mathscr{C}$. Then $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under intersection and linear-erasing homomorphism.*

Proof. From Theorem 2.2, it is immediate that $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under linear-erasing homomorphism. To show that $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under intersection, one fact is particularly useful. Recall that for any sets X and Y and any function f, $f(X) \cap Y = f(X \cap f^{-1}(Y))$.

This leads to the following result.

CLAIM. *For any classes $\mathscr{L}_1$ and $\mathscr{L}_2$ of languages, $H(\mathscr{L}_1) \wedge \mathscr{L}_2 \subseteq H(\mathscr{L}_1 \wedge H^{-1}(\mathscr{L}_2))$.*

From the claim we see that

$$H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq H((\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge H^{-1}(H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}))). \tag{1}$$

Using the techniques of [7], especially those displayed in part (3) of the proof of Theorem 1.1, one can show that

$$H^{-1}(H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})) \subseteq H(H^{-1}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})). \tag{2}$$

Both $\mathscr{C}$ and $\mathscr{L}_{\text{BNP}}$ are closed under inverse homomorphism and so $H^{-1}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq H^{-1}(\mathscr{C}) \wedge H^{-1}(\mathscr{L}_{\text{BNP}}) \subseteq \mathscr{C} \wedge \mathscr{L}_{\text{BNP}}$. Thus from (1) and (2) we obtain

$$H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq H((\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})). \tag{3}$$

From the claim we see that

$$(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq H(H^{-1}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge (\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})). \tag{4}$$

Since $\wedge$ is associative and commutative and $\mathscr{L}_{\text{BNP}}$ is closed under intersection, and since (as above) $H^{-1}(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq \mathscr{C} \wedge \mathscr{L}_{\text{BNP}}$, (4) yields

$$(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \wedge H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}}) \subseteq H(\mathscr{C} \wedge \mathscr{C} \wedge \mathscr{L}_{\text{BNP}}). \tag{5}$$

From Theorem 2.3 we see that $\mathscr{C} \wedge \mathscr{C} \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$. Applying this fact and the claim to (5) while recalling that $\mathscr{L}_{\text{BNP}}$ is closed under intersection and inverse

homomorphism and that the composition of nonerasing homomorphisms is again a nonerasing homomorphism, we obtain

$$
\begin{aligned}
(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}) \wedge H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}) &\subseteq H(\mathscr{C} \wedge \mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}) \\
&\subseteq H(H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}) \wedge \mathscr{L}_{\mathrm{BNP}}) \\
&\subseteq H(H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}} \wedge H^{-1}(\mathscr{L}_{\mathrm{BNP}}))) \\
&\subseteq H(H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}} \wedge \mathscr{L}_{\mathrm{BNP}})) \\
&\subseteq H(H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})) \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}).
\end{aligned} \tag{6}
$$

Combining (3) and (6), we have

$$
\begin{aligned}
H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}) \wedge H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}) &\subseteq H((\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}) \wedge H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})) \\
&\subseteq H(H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})) \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}).
\end{aligned} \tag{7}
$$

Thus, $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ is closed under intersection. □

Recall that a semi-AFL is a class of languages containing the e-free regular sets and closed under inverse homomorphism, nonerasing homomorphism, union, and intersection with regular sets. An intersection closed semi-AFL is closed under concatenation. If a semi-AFL contains the language $\{e\}$, then it contains the language $\{1\}^*$ and has the property that for any language L, if it contains L, then it contains $L \cup \{e\}$.

From Theorem 2.4 and results of [6]–[8], we have the following result.

THEOREM 2.5. *If $\mathscr{C}$ is a semi-AFL containing $\{e\}$ and closed under concatenation, then the intersection closure of $\mathscr{C}$ is included in $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$.*

Using techniques from [6], it can be shown that certain alterations of the hypotheses of Theorems 2.3 and 2.4 can be made without affecting the conclusions of those theorems. For example, conditions (i) and (ii) of Theorem 2.4 can be replaced by (i′) and (ii′) below:

(i′) $\mathscr{C}$ is a nonempty class containing at least one nonempty language;

(ii′) $\mathscr{C}$ is closed under marked concatenation, inverse homomorphism, and intersection with regular sets.

3. Use of intersection and linear-erasing homomorphic replication. In [5] the proof that any language in $\mathscr{L}_{\mathrm{BNP}}$ can be accepted in real time by a nondeterministic machine with just three reversal-bounded pushdown stores as auxiliary storage uses a technique commonly known as "tape-folding": symbols are written on tape squares as if the tape squares have several "channels" and at some later time the tape squares are read in such a way that the contents of different channels are compared. A similar use of this technique occurs in [4] where it is shown that a language is accepted in real time by a nondeterministic multitape machine if and only if it is accepted in real time by a nondeterministic machine with just two work tapes, one a pushdown store and the other a nonerasing stack. This technique also is used in studying properties of deterministic machines. The results in § 2 suggest that the use of this technique in reducing the number of working tapes needed to accept a language is in some way related to the property of a class of languages being closed under intersection with languages in $\mathscr{L}_{\mathrm{BNP}}$ and thus a characterization of $\mathscr{L}_{\mathrm{BNP}}$ in terms of "algebraic" notions may clarify the power of this technique. In [3] it is shown that $\mathscr{L}_{\mathrm{BNP}}$ is the smallest class containing the regular sets and closed under the operations of intersection and "linear-erasing homomorphic replication." Here we show that if a class $\mathscr{C}$ has certain properties, then the class $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ can be obtained from $\mathscr{C}$ and the class of regular sets by use of intersection and linear-erasing homomorphic replication.

Let n be a positive integer and let ρ be a function from $\{1, \cdots, n\}$ to $\{1, R\}$. Let L be a language and let $h_1, \cdots, h_n$ be homomorphisms. The language $\langle p; h_1, \cdots, h_n\rangle(L) = \{h_1(w)^{\rho(1)} \cdots h_n(w)^{\rho(n)} | w \in L\}$ is a *homomorphic replication of type ρ on L*. A class $\mathscr{L}$ of languages is *closed under homomorphic replication* if for every $n > 0$, every function $\rho: \{1, \cdots, n\} \to \{1, R\}$, every language $L \in \mathscr{L}$, and every n homomorphisms $h_1, \cdots, h_n$, the language $\langle p; h_1, \cdots, h_n\rangle(L)$ is in $\mathscr{L}$; and if for every $n > 0$, every function $\rho: \{1, \cdots, n\} \to \{1, R\}$, every language $L \in \mathscr{L}$, and every n homomorphisms $h_1, \cdots, h_n$ each of which is linear-erasing on L, the language $\langle p; h_1, \cdots, h_n\rangle(L)$ is in $\mathscr{L}$, then $\mathscr{L}$ is *closed under linear-erasing homomorphic replication.*

Clearly a class of languages closed under linear-erasing homomorphic replication is closed under linear-erasing homomorphism.

Let $\rho: \{1, 2\} \to \{1, R\}$ be defined by $\rho(1) = 1$ and $\rho(2) = R$. It is known that a language L is linear context-free if and only if there is a regular set S and two homomorphisms h_1, h_2 such that $\langle p; h_1, h_2\rangle(S) = L$. Further, the homomorphisms h_1, h_2 can be taken to be linear erasing on S.

The operation of homomorphic replication has been used in a variety of situations to characterize certain classes of languages [3], [9], [10], [12], [13]. Here we consider classes of the form $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$.

THEOREM 3.1. *Let $\mathscr{C}$ be any class of languages such that* (i) *$\mathscr{C}$ contains the language $\{1\}^*$ and all of the singleton sets,* (ii) *$\mathscr{C}$ is closed under concatenation and inverse homomorphism, and* (iii) *$\mathscr{C}$ has the property that if $L \in \mathscr{C}$, then $L \cup \{e\} \in \mathscr{C}$. Then $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is the smallest class containing the languages in $\mathscr{C}$ and all of the regular sets and closed under intersection and linear-erasing homomorphic replication.*

Before proving Theorem 3.1, note that if $\mathscr{C}$ is a semi-AFL containing $\{e\}$ and closed under concatenation, then $\mathscr{C}$ satisfies (i)–(iii). Thus we have the following corollary.

COROLLARY. *If $\mathscr{C}$ is a semi-AFL containing $\{e\}$ and closed under concatenation, then $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is the smallest semi-AFL containing the languages in $\mathscr{C}$ and closed under intersection and linear-erasing homomorphic replication.*

The remainder of this section is devoted to the proof of Theorem 3.1. We assume that the class $\mathscr{C}$ has the properties (i)–(iii) referred to in the statement of Theorem 3.1.

First notice that every language in $\mathscr{C}$ and every language in $\mathscr{L}_{\text{BNP}}$ (hence, every regular set and linear context-free language) is in $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$. To see this, consider $L \in \mathscr{C} \cup \mathscr{L}_{\text{BNP}}$ and a finite alphabet Σ such that $L \subseteq \Sigma^*$. Since $\{1\}^* \in \mathscr{C}$ and $\mathscr{C}$ is closed under inverse homomorphism, $\Sigma^* \in \mathscr{C}$. Since Σ^* is regular, Σ^* is in $\mathscr{L}_{\text{BNP}}$. Thus, $L = L \cap \Sigma^* \in \mathscr{C} \wedge \mathscr{L}_{\text{BNP}} \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$.

Second, notice that if L_1 and L_2 are in $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$, then so is L_1L_2, that is, $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under concatenation. To see this, notice that L_1 is in $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$, then for some $A_1 \in \mathscr{C}$, $B_1 \in \mathscr{L}_{\text{BNP}}$, and some nonerasing homomorphism $h_1: \Sigma_1^* \to \Delta_1^*$, $L_1 = h_1(A_1 \cap B_1)$. Let Σ_2 be a "copy" of Σ_1, that is, let Σ_2 be a set of new symbols, $\Sigma_1 \cap \Sigma_2 = \varnothing$, and let $c: \Sigma_1 \to \Sigma_2$ be one-to-one and onto. Let $h_2: (\Sigma_1 \cup \Sigma_2)^* \to (\Delta_1 \cup \Sigma_2)^*$ be the homomorphism determined by defining $h_2(a) = h_1(a)$ for $a \in \Sigma_1$ and $h_2(a) = a$ for $a \in \Sigma_2$. Since $\{1\}^* \in \mathscr{C}$ and $\mathscr{C}$ is closed under inverse homomorphism, Σ_2^* is in $\mathscr{C}$. Since $\mathscr{C}$ is closed under concatenation and $A_1, \Sigma_2^* \in \mathscr{C}$, the language $A_1\Sigma_2^*$ is in $\mathscr{C}$. Similarly, $B_1\Sigma_2^*$ is in $\mathscr{L}_{\text{BNP}}$. Thus, $h_2(A_1\Sigma_2^* \cap B_1\Sigma_2^*) = h_1(A_1 \cap B_1)\Sigma_2^* = L_1\Sigma_2^*$ is in $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$. Similarly, a copy $\hat{L}_2$ of L_2 can be constructed in $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ so that $\hat{L}_2 \subseteq \Sigma_2^*$ and $\Delta_1^*\hat{L}_2 \in H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$. Since $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under intersection, $L_1\Sigma_2^* \cap \Delta_1^*\hat{L}_2 = L_1\hat{L}_2$ is in $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$. Since L_1L_2 is the image of $L_1\hat{L}_2$ under a nonerasing homomorphism and $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$ is closed under nonerasing homomorphism, we have L_1L_2 in $H(\mathscr{C} \wedge \mathscr{L}_{\text{BNP}})$.

LEMMA 3.2. *The class $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ is closed under linear-erasing homomorphic replication.*

Proof. Let L be any language in $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ and let Σ be a finite alphabet such that $L \subseteq \Sigma^*$. Let ρ be any function, $\rho: \{1, \cdots, n\} \to \{1, R\}$, and for each $i = 1, \cdots, n$, let $h_i: \Sigma^* \to \Delta^*$ be a homomorphism that is linear-erasing on L. We must show that $\langle \rho; h_1, \cdots, h_n \rangle(L) = \{h_1(w)^{\rho(1)} \cdots h_n(w)^{\rho(n)} | w \in L\}$ is in $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$.

First, for each $i = 1, \cdots, n$, let $f_i: \Sigma^* \to \Delta^*$ be the homomorphism determined by defining $f_i(a) = h_i(a)$ for each $a \in \Sigma$ if $\rho(i) = 1$ and $f_i(a) = h_i(a)^{\mathrm{R}}$ for each $a \in \Sigma$ if $\rho(i) = R$. Notice that for any $w \in \Sigma^*$, $f_i(w) = h_i(w)^{\rho(i)}$ if $\rho(i) = 1$ and $f_i(w^{\mathrm{R}}) = h_i(w)^{\rho(i)}$ if $\rho(i) = R$.

Second, for each $i = 1, \cdots, n$, if $\rho(i) = 1$, then let $A_i = \{x \# y \# z_1 \# \cdots \# z_n | x, y \in \Sigma^*$, each $z_j \in \Delta^*$, and $z_i = f_i(y^R)\}$, and if $\rho(i) = R$, then let $A_i = \{x \# y \# z_1 \# \cdots \# z_n | x, y \in \Sigma^*$, each $z_j \in \Delta^*$, and $z_i = f_i(x^{\mathrm{R}})\}$, where $\#$ is a symbol not in $\Sigma \cup \Delta$. Let $A_0 = \{x \# x^{\mathrm{R}} \# z_1 \# \cdots \# z_n | x \in \Sigma^*$, each $z_j \in \Delta^*\}$. Now $A_0 \cap A_1 \cap \cdots \cap A_n = \{x \# x^{\mathrm{R}} \# h_1(x)^{\rho(1)} \# \cdots \# h_n(x)^{\rho(n)} | x \in \Sigma^*\}$. Clearly each A_i is a linear context-free language so that $A_0 \cap A_1 \cap \cdots \cap A_n$ is in $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ since $\mathscr{L}_{\mathrm{BNP}}$ contains the linear context-free languages and is closed under intersection (Proposition 1.2) and since $\mathscr{L}_{\mathrm{BNP}} \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$.

Third, let $B = L \# \Sigma^* (\# \Delta^*)^n$ so that B is in $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ by choice of L and the fact that $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ is closed under concatenation with regular sets. Since $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ is closed under intersection, $B \cap (A_0 \cap \cdots \cap A_n)$ is in $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$. But

$$B \cap (A_0 \cap \cdots \cap A_n) = \{x \# x^{\mathrm{R}} \# h_1(x)^{\rho(1)} \# \cdots \# h_n(x)^{\rho(n)} | x \in L\}$$

so that $\langle \rho; h_1, \cdots, h_n \rangle(L) = \{h_1(w)^{\rho(1)} \cdots h_n(w)^{\rho(n)} | w \in L\}$ is the image of $B \cap A_0 \cap \cdots \cap A_n$ under a homomorphism that is linear erasing on $B \cap A_0 \cap \cdots \cap A_n$. Since $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ is closed under linear erasing homomorphism, $\langle \rho; h_1, \cdots, h_n \rangle(L)$ is in $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$. □

Now we can prove Theorem 3.1. As noted above both $\mathscr{C}$ and the class of regular sets are included in $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$. By Theorem 2.4, $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ is closed under intersection, and by Lemma 3.2, $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ is closed under linear-erasing homomorphic replication. Letting $\mathscr{L}_0$ be the smallest class containing every language in $\mathscr{C}$ and every regular set and closed under intersection and linear-erasing homomorphic replication, we see that $\mathscr{L}_0 \subseteq H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$. On the other hand, it has been shown [3] that $\mathscr{L}_{\mathrm{BNP}}$ is the smallest class containing all regular sets and closed under intersection and linear-erasing homomorphic replication. Thus $\mathscr{L}_{\mathrm{BNP}} \subseteq \mathscr{L}_0$, and by choice of $\mathscr{L}_0$, $\mathscr{C} \subseteq \mathscr{L}_0$, so that $\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}} \subseteq \mathscr{L}_0$ since $\mathscr{L}_0$ is closed under intersection. Since $\mathscr{L}_0$ is closed under linear-erasing homomorphic replication, $\mathscr{L}_0$ is closed under nonerasing homomorphism so that $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}}) \subseteq \mathscr{L}_0$. Thus $H(\mathscr{C} \wedge \mathscr{L}_{\mathrm{BNP}})$ is the smallest class containing the languages in $\mathscr{C}$ and all of the regular sets and closed under intersection and linear-erasing homomorphic replication.

4. Properties of $\mathscr{L}_{\mathrm{BNP}}$. Now we consider some properties of $\mathscr{L}_{\mathrm{BNP}}$. Recall that for a language L, the Kleene + of L, L^+, is defined to be $L^+ = \bigcup_{i \geq 1} L^i$, where $L^1 = L$ and L^{n+1} is the concatenation of L^n and L, $L^{n+1} = L^n L$. Recall that an abstract family of languages (AFL) is a semi-AFL that is closed under concatenation and Kleene +. For an arbitrary class $\mathscr{C}$ of languages, let $\mathscr{F}(\mathscr{C})$ be the smallest AFL containing $\mathscr{C}$ and let $\mathscr{F}_\cap(\mathscr{C})$ be the smallest AFL containing $\mathscr{C}$ and closed under intersection.

It is not known whether $\mathscr{L}_{\mathrm{BNP}}$ is closed under Kleene +. Clearly $\mathscr{L}_{\mathrm{BNP}}$ is closed under Kleene + if and only if $\mathscr{L}_{\mathrm{BNP}}$ is an AFL. We will develop other necessary and sufficient conditions for $\mathscr{L}_{\mathrm{BNP}}$ to be an AFL.

Consider $\mathscr{F}_\cap(\text{LIN})$. Since $\mathscr{F}_\cap(\text{LIN})$ contains LIN and is closed under intersection and nonerasing homomorphism and since $\mathscr{L}_{\text{BNP}}$ is the smallest class containing LIN and closed under intersection and nonerasing homomorphism, we see that $\mathscr{L}_{\text{BNP}} = H(\wedge\text{LIN}) \subseteq \mathscr{F}_\cap(\text{LIN})$ and hence that $\mathscr{F}(\mathscr{L}_{\text{BNP}}) \subseteq \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}}) \subseteq \mathscr{F}_\cap(\text{LIN})$. Since $\text{LIN} \subseteq \mathscr{L}_{\text{BNP}}$, $\mathscr{F}_\cap(\text{LIN}) \subseteq \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$ so that $\mathscr{F}_\cap(\text{LIN}) = \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$. Also, from the results in [6], [8], we see that $\mathscr{F}_\cap(\text{LIN}) = H(\wedge\mathscr{F}(\text{LIN}))$. From the results in § 2, we see that $H(\wedge\mathscr{F}(\text{LIN})) = H(\mathscr{F}(\text{LIN}) \wedge \mathscr{L}_{\text{BNP}})$ since $\mathscr{L}_{\text{BNP}} = H(\wedge\text{LIN})$. Thus we have the following result.

PROPOSITION 4.1. *The smallest intersection-closed AFL containing the class of linear context-free languages is the smallest intersection-closed AFL containing the class* $\mathscr{L}_{\text{BNP}}$. *A language is in this AFL if and only if it is the nonerasing homomorphic image of the intersection of a language in* $\mathscr{L}_{\text{BNP}}$ *and a language in the smallest AFL containing the class of linear context-free languages, that is,* $\mathscr{F}_\cap(\text{LIN}) = \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}}) = H(\mathscr{F}(\text{LIN}) \wedge \mathscr{L}_{\text{BNP}})$.

A class $\mathscr{C}$ of languages is *translatable* if for every L in $\mathscr{C}$ and every choice of two symbols a, b that do not occur in any string in L, the language $\{a^n w b^n \mid n \geqq 0, w \in L\}$ is in $\mathscr{C}$ [2], [15].

The following result is useful.

LEMMA 4.2 [2, 15]. *If* $\mathscr{C}$ *is a semi-AFL with the property that* $\mathscr{F}(\mathscr{C})$ *is translatable, then* $\mathscr{C}$ *is an AFL, that is,* $\mathscr{C} = \mathscr{F}(\mathscr{C})$.

Since $\mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$ is an AFL closed under intersection, $\mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$ is closed under e-free substitution. Since $\{a^n c b^n \mid n \geqq 0\}$ is in LIN and hence in $\mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$, it is clear that $\mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$ is translatable. Also, $\mathscr{L}_{\text{BNP}}$ is translatable.

We are interested in comparing the classes $\mathscr{L}_{\text{BNP}}$, $\mathscr{F}(\mathscr{L}_{\text{BNP}})$, and $\mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$. While it is clear that $\mathscr{L}_{\text{BNP}} \subseteq \mathscr{F}(\mathscr{L}_{\text{BNP}}) \subseteq \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$, it is not known whether either of these inclusions is strict. The next result speaks to this question.

THEOREM 4.3. *Either* $\mathscr{F}(\mathscr{L}_{\text{BNP}})$ *is translatable and* $\mathscr{L}_{\text{BNP}} = \mathscr{F}(\mathscr{L}_{\text{BNP}}) = \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$, *or* $\mathscr{F}(\mathscr{L}_{\text{BNP}})$ *is not translatable and* $\mathscr{L}_{\text{BNP}} \subsetneq \mathscr{F}(\mathscr{L}_{\text{BNP}}) \subsetneq \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$.

Proof. Suppose that $\mathscr{F}(\mathscr{L}_{\text{BNP}})$ is translatable. Since $\mathscr{L}_{\text{BNP}}$ is a semi-AFL and $\mathscr{F}(\mathscr{L}_{\text{BNP}})$ is translatable, Lemma 4.2 shows that $\mathscr{L}_{\text{BNP}}$ is an AFL. Thus, $\mathscr{L}_{\text{BNP}} = \mathscr{F}(\mathscr{L}_{\text{BNP}})$. But $\mathscr{L}_{\text{BNP}}$ is closed under intersection (Proposition 1.2) and $\mathscr{F}(\mathscr{L}_{\text{BNP}})$ is an AFL so that $\mathscr{L}_{\text{BNP}}$ is itself the smallest intersection closed AFL containing $\mathscr{L}_{\text{BNP}}$, that is, $\mathscr{L}_{\text{BNP}} = \mathscr{F}(\mathscr{L}_{\text{BNP}}) = \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$.

Suppose that $\mathscr{F}(\mathscr{L}_{\text{BNP}})$ is not translatable. Then $\mathscr{F}(\mathscr{L}_{\text{BNP}}) \neq \mathscr{L}_{\text{BNP}}$ since $\mathscr{L}_{\text{BNP}}$ is translatable and $\mathscr{F}(\mathscr{L}_{\text{BNP}}) \neq \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$ since $\mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$ is translatable. Thus, $\mathscr{L}_{\text{BNP}} \subsetneq \mathscr{F}(\mathscr{L}_{\text{BNP}}) \subsetneq \mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$. □

Let us consider characterizations of the classes $\mathscr{L}_{\text{BNP}}$, $\mathscr{F}(\mathscr{L}_{\text{BNP}})$, and $\mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$ in terms of machines. Recall from Proposition 1.1 that a language is in $\mathscr{L}_{\text{BNP}}$ if and only if it is accepted in real time by a nondeterministic Turing machine with three pushdown stores as auxiliary storage which operates in such a way that in every computation each pushdown store makes at most one reversal. From [6]–[8] we see that a language is in $\mathscr{F}(\mathscr{L}_{\text{BNP}})$ if and only if it is accepted in real time by a nondeterministic Turing machine with three pushdown stores as auxiliary storage which operates in such a way that in every computation (i) each pushdown store makes at most one reversal between the times it is empty, and (ii) when any one pushdown store becomes empty, it cannot begin to write again until the other two pushdown stores become empty (that is, all three pushdown stores must be re-initialized before any one can be restarted). Similarly, a language is in $\mathscr{F}_\cap(\mathscr{L}_{\text{BNP}})$ if and only if it can be accepted in real time by a nondeterministic Turing machine with three pushdown stores as auxiliary storage which operates in such a way that in every computation each pushdown store makes at most one reversal between the times it is empty.

In the above characterizations, if one removes the restriction that the machines operate in real time and considers operation without time bounds, then the three types of machines are equivalent—they accept precisely the recursively enumerable sets [1].

Let $\mathrm{PAL}_e = \{wcw^R | w \in \{a, b\}^*\} \cup \{e\}$ and recall that the class of linear context-free languages is the smallest semi-AFL containing PAL_e. The smallest AFL containing PAL_e is the smallest AFL containing the linear context-free languages and this class is the smallest semi-AFL containing PAL_e^+. Thus we see that $\mathscr{L}_{\mathrm{BNP}}$ is an AFL if and only if PAL_e^+ is in $\mathscr{L}_{\mathrm{BNP}}$.

CONJECTURE 1. *Each of the following statements is true*:

(i) $\mathscr{L}_{\mathrm{BNP}}$ *is not closed under Kleene* $+$;

(ii) *The language* $\{wcw^R c | w \in \{a, b\}^*\}^+$ *is not in* $\mathscr{L}_{\mathrm{BNP}}$;

(iii) $\mathscr{L}_{\mathrm{BNP}}$ *is not an AFL* ($\mathscr{L}_{\mathrm{BNP}} \subsetneq \mathscr{F}(\mathscr{L}_{\mathrm{BNP}})$);

(iv) *The smallest AFL containing* $\mathscr{L}_{\mathrm{BNP}}$ *is not closed under intersection* ($\mathscr{F}(\mathscr{L}_{\mathrm{BNP}}) \subsetneq \mathscr{F}_\cap(\mathscr{L}_{\mathrm{BNP}})$);

(v) *The smallest AFL containing* $\mathscr{L}_{\mathrm{BNP}}$ *is not translatable*;

(vi) $\mathscr{L}_{\mathrm{BNP}} \subsetneq \mathscr{F}(\mathscr{L}_{\mathrm{BNP}}) \subsetneq \mathscr{F}_\cap(\mathscr{L}_{\mathrm{BNP}})$.

5. Applications. In this section we consider some applications of the results presented in §§ 2 and 3 and an open question.

Let us consider classes of languages accepted by certain restricted Turing machines. In each case the Turing machines have a two-way read-only input tape, finite-state control, and some auxiliary storage. For any $k \geqq 1$, let DSPACE $((\log n)^k)$ (NSPACE $((\log n)^k)$) be the class of languages accepted by those machines that are deterministic (nondeterministic) and have a storage tape that is bounded in length by $(\log n)^k$ where n is the length of the input string. Let 2DPDA (2DSA, 2NEDSA) be the class of languages accepted by two-way deterministic pushdown store acceptors (respectively, stack acceptors, nonerasing stack acceptors), and let 2NPDA (2NSA, 2NENSA) be the class of languages accepted by the analogous nondeterministic devices. If $\mathscr{C}$ is any one of these classes, then consider $H(\mathscr{C})$. It is straightforward to show that $H(\mathscr{C})$ is closed under union, intersection, inverse homomorphism, nonerasing homomorphism, concatenation, and contains the linear context-free languages. Hence, $H(\mathscr{C})$ contains $\mathscr{L}_{\mathrm{BNP}}$ and by the results of §§ 2 and 3, $H(\mathscr{C})$ is closed under linear-erasing homorphic replication and under intersection. The same comments can be applied to the class of full rudimentary predicates when this class is viewed as a class of languages as in [14].

Now consider the class $\mathscr{L}_{\mathrm{BNP}}$. Let Q be the class of languages accepted in real time by nondeterministic multitape Turing machines. Clearly, $\mathscr{L}_{\mathrm{BNP}} \subseteq Q$. As noted in [5], $Q = \mathscr{L}_{\mathrm{BNP}}$ if and only if every context-free language is in $\mathscr{L}_{\mathrm{BNP}}$. Since all of the context-free languages can be obtained from the Dyck set on two letters by means of operations under which $\mathscr{L}_{\mathrm{BNP}}$ is closed, every context-free language is in $\mathscr{L}_{\mathrm{BNP}}$ if and only if the Dyck set on two letters is in $\mathscr{L}_{\mathrm{BNP}}$.

CONJECTURE 2. *The Dyck set on two letters is not in* $\mathscr{L}_{\mathrm{BNP}}$.

Note that Conjecture 2 follows from Conjecture 1.

REFERENCES

[1] B. BAKER AND R. BOOK, *Reversal-bounded multipushdown machines*, J. Comput. System Sci., 8 (1974), pp. 315–332.

[2] L. BOASSON AND M. NIVAT, *Sur diverses familles de langages fermées par transduction rationnelle*, Acta Informatica, 2 (1973), pp. 180–188.

[3] R. BOOK, *Simple representations of certain classes of languages*, J. Assoc. Comput. Mach., to appear.

[4] R. BOOK AND S. GREIBACH, *Quasi-realtime languages*, Math. Systems Theory, 4 (1970), pp. 97–111.
[5] R. BOOK, M. NIVAT AND M. PATERSON, *Reversal-bounded acceptors and intersections of linear languages*, this Journal, 3 (1974), pp. 283–295.
[6] S. GINSBURG AND S. GREIBACH, *Abstract families of languages*, Studies in Abstract Families of Languages, Memoir no. 87, American Mathematical Society, Providence, RI, 1969, pp. 1–32.
[7] S. GINSBURG, S. GREIBACH AND J. HOPCROFT, *Pre-AFL*, Studies in Abstract Families of Languages, Memoir no. 87, American Mathematical Society, Providence, RI, 1969, pp. 41–51.
[8] S. GINSBURG AND S. GREIBACH, *Principal AFL*, J. Comput. System Sci., 4 (1970), pp. 308–338.
[9] S. GREIBACH, *Control sets on context-free grammar forms*, Ibid., 15 (1977), pp. 35–98.
[10] ———, *One-way finite visit automata*, Theoret. Comput. Sci., to appear.
[11] S. GREIBACH AND S. GINSBURG, *Multi-tape AFA*, J. Assoc. Comput. Mach., 19 (1972), pp. 193–221.
[12] K. KLINGENSTEIN, *Structures of bounded languages in certain families of languages*, Information Control, to appear.
[13] ———, *ρ-matrix languages*, Theoret. Comput. Sci., to appear.
[14] C. WRATHALL, *Rudimentary predicates and relative computation*, this Journal, to appear.
[15] S. GREIBACH, *Erasing in context-free AFLs*, Information Control, 21 (1972), pp. 436–465.

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

APPROXIMATION ALGORITHMS FOR SOME ROUTING PROBLEMS*

GREG N. FREDERICKSON†, MATTHEW S. HECHT‡ AND CHUL E. KIM¶

Abstract. Several polynomial time approximation algorithms for some *NP*-complete routing problems are presented, and the worst-case ratios of the cost of the obtained route to that of an optimal are determined. A mixed-strategy heuristic with a bound of 9/5 is presented for the stacker-crane problem (a modified traveling salesman problem). A tour-splitting heuristic is given for k-person variants of the traveling salesman problem, the Chinese postman problem, and the stacker-crane problem, for which a minimax solution is sought. This heuristic has a bound of $e+1-1/k$, where e is the bound for the corresponding 1-person algorithm.

Key words. *NP*-complete problems, polynomial-time approximation algorithm, heuristic, worst-case performance bound, traveling salesman problem, Chinese postman problem, stacker-crane problem, k-person routing

1. Introduction. Routing problems that involve the periodic collection and delivery of goods and services are of great practical importance. Common examples of such problems include mail delivery, newspaper delivery, parcel pickup and delivery, trash collection, schoolbus routing, snow removal, and fuel oil delivery. The practical goals of scrutinizing such problems are cost minimization and service improvement. Abstractions of these problems can be modeled easily and naturally with graphs.

Unfortunately, many of the interesting routing problems, including for instance the well known traveling salesman problem, are *NP*-complete in the sense of Cook [3] and Karp [10]. The problems we consider in this paper, the stacker-crane problem and three k-person routing problems, are also *NP*-complete. It has been conjectured that there exist no efficient exact algorithms for any of the optimization versions of the *NP*-complete problems. Consequently, attention has been given to developing algorithms that solve various *NP*-complete problems efficiently but only approximately [7], [8], [9], [14]. We shall restrict our attention to approximation algorithms for routing problems for which worst-case analysis has been performed.

Previous worst-case analysis of approximation algorithms for routing problems has focused on the traveling salesman problem. Sahni and Gonzalez [16] have shown that if the triangle inequality is not satisfied, the problem of finding an approximate solution for the traveling salesman problem within any constant bound ratio of the optimum is as difficult as finding an exact solution. Papadimitriou and Steiglitz [12] have derived a similar result for local search algorithms.

If the triangle inequality is satisfied (or, equivalently, if the tour is allowed to visit vertices more than once), then approximation algorithms with a constant worst-case bound exist. Rosenkrantz, Lewis, and Stearns [14] have applied worst-case analysis to several incremental (insertion) heuristics. They show that none of the algorithms examined have a worst-case bound better than 2. This bound is the same as that yielded by doubling up the edges in a minimum spanning tree. Recently, Christofides [2] has developed an algorithm with a worst-case bound of 1.5, which involves forming a minimum spanning tree and performing a minimum cost matching on the vertices of

* Received by the editors July 23, 1976, and in final revised form June 27, 1977.

† Department of Computer Science, University of Maryland, College Park, Maryland 20742.

‡ Department of Computer Science, University of Maryland, College Park, Maryland 20742. This research was partially supported by the National Science Foundation under Grant DCR-08361.

¶ Department of Computer Science, University of Maryland, College Park, Maryland 20742. This research was partially supported by a University of Maryland General Research Board Award.

odd degree. We make use of Christofides' techniques in achieving a worst-case bound of 1.8 for an algorithm for the stacker-crane problem.

The stacker-crane problem, which was brought to our attention by Rosenkrantz [13], is a modified traveling salesman problem that requires that a set of arcs be traversed, rather than a set of vertices visited. The problem encompasses practical applications such as operating a crane or a forklift, or driving a pick-up and delivery truck. The crane must start from an initial position, perform a set of moves, and return to a terminal position. The goal is to schedule the moves so as to minimize total tour length. No order is imposed on the moves, and no moves may be combined and performed simultaneously.

We also consider several k-person routing problems: the k-traveling salesman (k-TSP), the k-Chinese postman (k-CPP), and the k-stacker-cranes (k-SCP), for $k \geqq 2$. These problems reflect more of the flavor of real world problems than 1-person problems. When one salesman cannot handle a large territory, k salesmen are dispatched to collectively visit each city in the territory, under the constraint that no one of the k salesmen has too large of a task.

We thus choose as our optimization criterion the minimizing of the maximum of the k salesmen's tour costs. This criterion differs from the minimizing of total tour costs subject to each salesman visiting at least one city, as suggested by Bellmore and Hong [1], and also differs from minimizing total tour costs subject to no salesman visiting more than p cities, as suggested by Miller, Tucker, and Zemlin [11].

We have found no worst-case analysis for any k-person routing problems in the literature. Our results indicate that generalizations of 1-person incremental algorithms may not do well. The best bounds we have been able to achieve for incremental algorithms for k-TSP have a multiplicative factor of k. In contrast, we have found that a simple tour-splitting heuristic, where a reasonable tour for 1 person is split into k tours, has better worst-case behavior. The bound increases only very modestly as a function of k, so that if e is the bound on a 1-person algorithm, then the bound for our k-TSP, k-CPP, and k-SCP algorithms is $e + 1 - 1/k$.

We now proceed to some basic definitions. An *undirected graph* $G = (V, E)$ consists of a set V of vertices and a set E of undirected edges. Each *edge* (u, v) connects two vertices u and v in V. A *mixed graph* $G = (V, E, A)$ consists of a set V of vertices, a set E of undirected edges, and a set A of arcs. An *arc* $\langle t, h \rangle$ is a directed edge from t to h, both vertices in V. We call t the *tail* of arc $\langle t, h \rangle$, and h the *head* of arc $\langle t, h \rangle$.

A multiset is a set with a function mapping the elements of the set into the positive integers, to indicate that an element may appear more than once. A *multigraph* is a graph $G = (V, E)$ in which E is a multiset.

The *degree* of a vertex is the number of edges and arcs incident on the vertex. The *indegree* is the number of arcs directed into the vertex. The *outdegree* is the number of arcs directed out of the vertex. A vertex is of *even degree* if the degree is an even number, and is of *odd degree* otherwise.

Given a mixed graph $G = (V, E, A)$, a *path* is a sequence of vertices such that for each adjacent pair of vertices v_i and v_{i+1}, there is either an edge (v_i, v_{i+1}) or an arc $\langle v_i, v_{i+1} \rangle$ in the graph. The first vertex in a path is called the *initial vertex*, the last is the *terminal vertex*. A *cycle* is a path with identical initial and terminal vertices. A *tour* is a cycle visiting all vertices in V, and a *subtour* is a tour visiting a subset of the vertices in V. A *k-tour* is a set of k subtours, each visiting the initial vertex, and collectively visiting all vertices in V.

Given a multigraph $G = (V, E)$, a *route* is a sequence of alternating vertices and edges, covering all edges, such that each edge connects the immediately preceding and

succeeding vertices. A *subroute* is a route which covers a subset of E. An arc $\langle t, h\rangle$ is *traversed* by covering it in a direction from its tail t to its head h.

A (maximum cardinality) *matching* $E' \subseteq E$ of a graph $G = (V, E)$ is a (maximal) subset of edges which share no vertices. A *minimum-cost matching* of G is a matching such that the total cost of the edges between the paired vertices is minimum. Given a partition $V = V_1 \cup V_2$, a *bipartite matching* is a matching of $G' = (V, (V_1 \times V_2) \cap E)$. A *spanning tree* of a connected graph $G = (V, E)$ is a subgraph $T = (V, E')$ of G which is a tree. A *minimum cost spanning tree* is a spanning tree such that the total cost of the edges E' is minimum.

2. The stacker-crane problem. We define the stacker-crane problem (SCP) as follows: Let $G = (V, E, A)$ be a mixed graph with a distinguished initial vertex v_s. Let c be a cost function from $E \cup A$ to the set of nonnegative integers, such that for every arc there is a parallel edge of no greater cost. The optimization version is to find a tour starting at v_s and traversing each arc in A, such that the cost of the tour is minimum. The recognition version is, given a positive integer C, decide if there is a tour of cost at most C.

As stated previously, the stacker-crane problem is *NP*-complete. An instance of TSP can be transformed into an instance of SCP as follows:

For each vertex v_i in the TSP graph, create two vertices v_{it} and v_{ih}, and an arc $\langle v_{it}, v_{ih}\rangle$ of cost zero. For each edge (v_i, v_j), create edges (v_{it}, v_{jt}), (v_{it}, v_{jh}), (v_{ih}, v_{jt}), and (v_{ih}, v_{jh}), and assign cost $c(v_i, v_j)$ to each of the edges. (See Fig. 1 for an example of the translation.) Choose any vertex v_{it} and designate it the initial vertex.

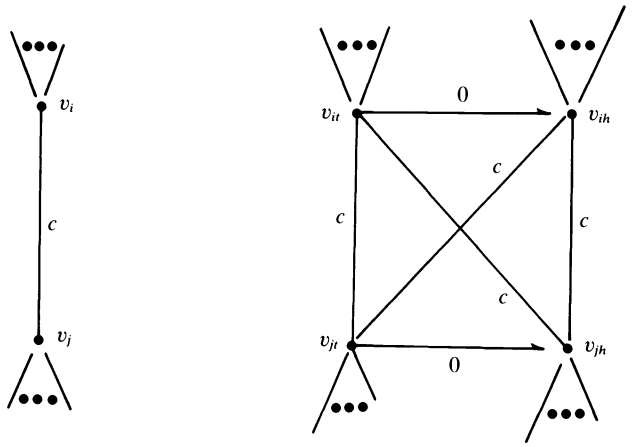


FIG. 1. *Translation from* TSP *to* SCP.

A tour of cost at most C for TSP can be translated into a tour of cost at most C for SCP. Just replace each v_i in the TSP tour by "v_{it}, v_{ih}" to get an SCP tour. Conversely, a tour of cost at most C for SCP can be translated into a tour of cost at most C for TSP by changing v_{ih} to v_i and deleting v_{it}, for all v_i in V.

We now consider approximation algorithms for SCP. We point out that there is a fairly simple algorithm which consists of forming the analogue of a minimum-cost spanning tree for the arcs, and then introducing an additional edge for each edge and arc in the spanning tree. This algorithm will have a worst-case bound of 2, which is approachable. In this section we shall present an algorithm with a better worst-case bound.

We require that a mixed graph satisfy the following properties:

1. Each vertex is either the head or the tail of at least one arc in A.
2. The cost function on edges satisfies the triangle inequality.

If a graph does not satisfy the preceding properties, Algorithm PREPROCESS will transform the graph into an equivalent graph which satisfies these properties.

ALGORITHM PREPROCESS.

Input: A mixed graph $G = (V, E, A)$ and a cost function c.

Output: A mixed graph $G' = (V', E', A)$ satisfying properties 1 and 2, and a cost function c'.

1. If v_s is not an end point of any arc, create a terminal vertex v_f and an arc $\langle v_f, v_s \rangle$ of cost zero from the terminal vertex to the initial vertex. For all v_i in V, create edges (v_i, v_f) of cost $c(v_i, v_s)$.
2. Insert all vertices which are endpoints of arcs into V'. Calculate the shortest path between every pair of vertices v_i and v_j in V', insert (v_i, v_j) into E', and set $c(v_i, v_j)$ to the cost obtained.

We shall use the following notation in the analysis of the worst-case behavior of our algorithm. Let C^* represent the cost of an optimal tour, and let C_A be the total cost of all arcs.

We consider two strategies which depend on the relative size of C_A as compared with C^*. If C_A is large relative to C^*, then the analogue of a minimum spanning tree for the arcs will be small. An appropriate strategy in this case is to perform a minimum cost matching on the heads and tails of arcs, and link the resulting cycles together with the spanning edges. If C_A is small relative to C^*, then the problem is essentially a traveling salesman problem. In this case shrink the arcs to nodes, perform Christofides' algorithm [2], and add certain edges corresponding to the arcs to make the tour traverse arcs correctly.

Thus our algorithm consists of two algorithms, each handling certain cases well. Each algorithm is run, and the better of the two results is chosen. This procedure will guarantee a better worst-case bound than either algorithm alone. Since we know of no efficient way to compute the exact ratio of C_A to C^*, it is difficult to know in advance which of the two algorithms will guarantee a better worst-case bound.

We now present an algorithm which does well if the cost of the arcs is large relative to the cost of the optimum tour. An example of Algorithm LARGEARCS applied to a graph is shown in Fig. 2.

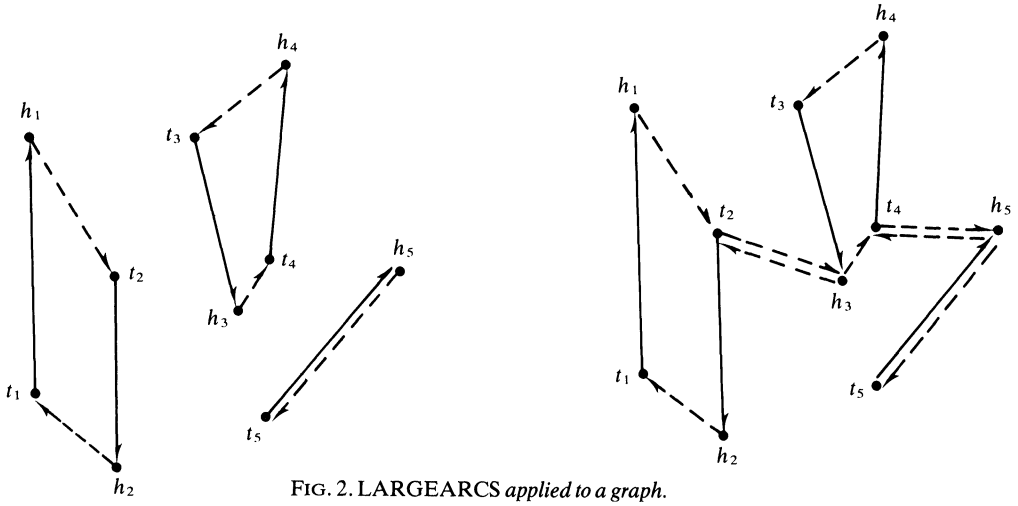


FIG. 2. LARGEARCS *applied to a graph.*

ALGORITHM LARGEARCS.

Input: A mixed graph G' satisfying properties 1 and 2.

Output: A sequence of vertices representing a tour.

1. Find a minimum cost bipartite matching between the multisets of heads and tails of arcs.
2. Initialize E'' to be empty. For each edge included in the matching, associate a direction with it, going from the vertex which is a head of an arc to the vertex which is a tail of an arc, and insert it into E''. (This results in $m \geqq 1$ disjoint cycles consisting of alternating edges and arcs.)
3. Let the m disjoint cycles be R_i, $1 \leqq i \leqq m$, with each R_i represented by a single node n_i. Form the inter-node distances from the original edge costs:

$$d(n_i, n_j) = \min \{c'(u, v) | u \in R_i, v \in R_j\}.$$

   Associate with (n_i, n_j) the particular edge (u, v) which yields the minimum cost.
4. Find a minimum cost spanning tree for the nodes $\{n_i \mid 1 \leqq i \leqq m\}$, using the distance function d.
5. Rename each spanning edge in terms of the original vertices. Make two copies of each edge, associating one direction with one edge, and the opposite direction with the other edge, and insert these edges into E''.
6. Call POSTPROCESS.

We next present an algorithm that performs well if the cost of the arcs is small relative to the cost of the optimum tour. An example of Algorithm SMALLARCS applied to a graph is shown in Fig. 3.

ALGORITHM SMALLARCS.

Input: Same as LARGEARCS.

Output: Same as LARGEARCS.

1. Represent each arc $\langle t_i, h_i \rangle$ by a node n_i. For each pair of nodes n_i and n_j, define $c'(n_i, n_j)$ as the minimum of $c(t_i, t_j)$, $c(t_i, h_j)$, $c(h_i, t_j)$ and $c(h_i, h_j)$. Perform an all shortest paths algorithm using c' to find the inter-node distance $d(n_i, n_j)$. Associate with each edge (n_i, n_j) the edges in the shortest path between n_i and n_j.
2. Find a minimum cost spanning tree for the nodes $\{n_i\}$, using the distance function d.
3. Identify nodes of odd degree in the spanning tree, and perform a minimum cost matching on these nodes using the distance function d.
4. Rename the spanning edges and matching edges in terms of the vertices in V'. Define $G'' = (V, E'', A)$, where E'' is the multiset of spanning edges and matching edges. Consider the degree of vertices in G''. For all arcs $\langle u, v \rangle$ whose endpoints have odd degree, add the edge (u, v) to E'' and associate with it the direction opposite that of the arc. These arcs requiring no such edge shall be called *even arcs*, with total cost C'_A.
5. Find a tour which exactly covers $E'' \cup A$, ignoring even arc directions. If the cost of the even arcs which are traversed backwards is more than $(1/2)C'_A$, then reverse the direction of the tour.
6. Associate with each undirected edge the direction of the tour. For even arc that is incorrectly traversed, add two edges to E'', both with associated direction opposite that of the arc.
7. Call POSTPROCESS.

We next present a postprocessing algorithm, and then finally the complete algorithm with subroutine calls. Step 1 in Algorithm POSTPROCESS can make use of

an algorithm for finding tours in directed graphs where all vertices have the same indegree as outdegree, such as the algorithm in Edmonds and Johnson [4]. We complete the example of Fig. 3 in Fig. 4, showing the effect of step 2 in POSTPROCESS.

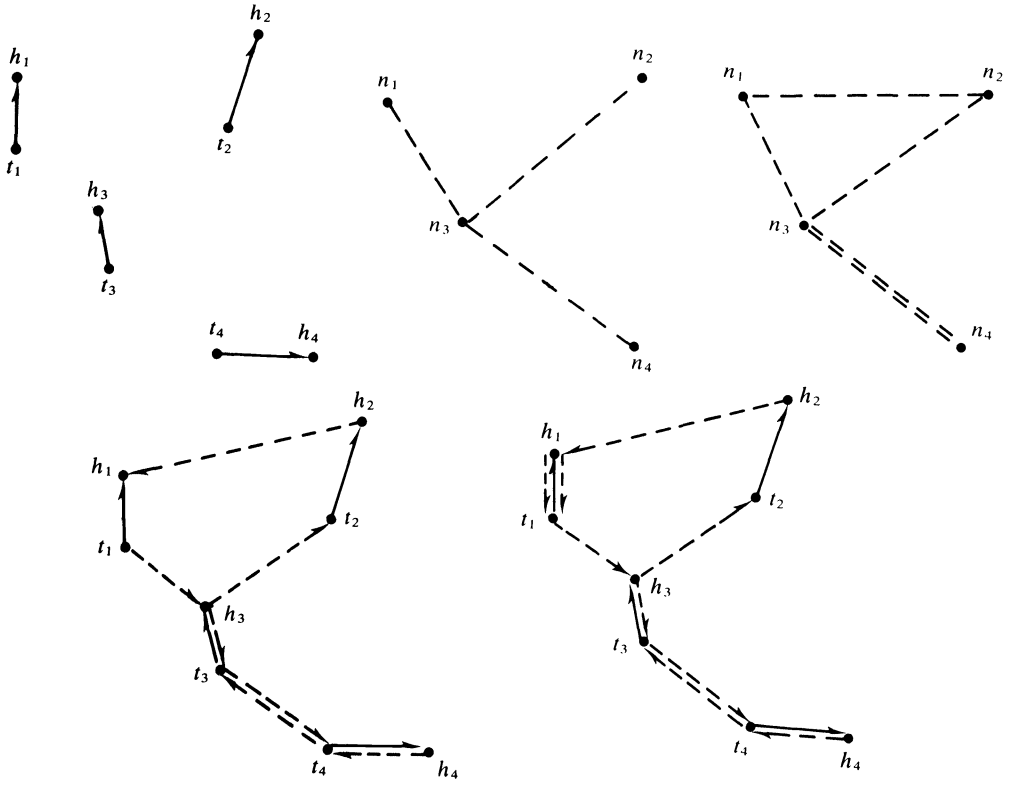


FIG. 3. SMALLARCS *applied to a graph.*

ALGORITHM POSTPROCESS.
Input: A set of edges and arcs from which a tour can be constructed.
Output: A sequence of vertices representing a tour.

1. Given a set of arcs and edges with an associated direction, from which a tour can be constructed, find the tour.
2. For any series of two or more consecutive edges in the tour, replace them with one edge, thus eliminating the intermediate vertices.
3. For any two vertices v_i and v_j anchoring an edge in the tour, insert any vertices that would create a shorter path between v_i and v_j. (Undo Step 2 of PREPROCESS).
4. List the tour beginning at v_s, the initial vertex, and finishing at v_f the terminal vertex.

ALGORITHM CRANE.
Input: A mixed graph G.
Output: A sequence of vertices representing a tour.

1. Call PREPROCESS.
2. Call LARGEARCS.
3. Call SMALLARCS.
4. Select the tour of smaller cost.

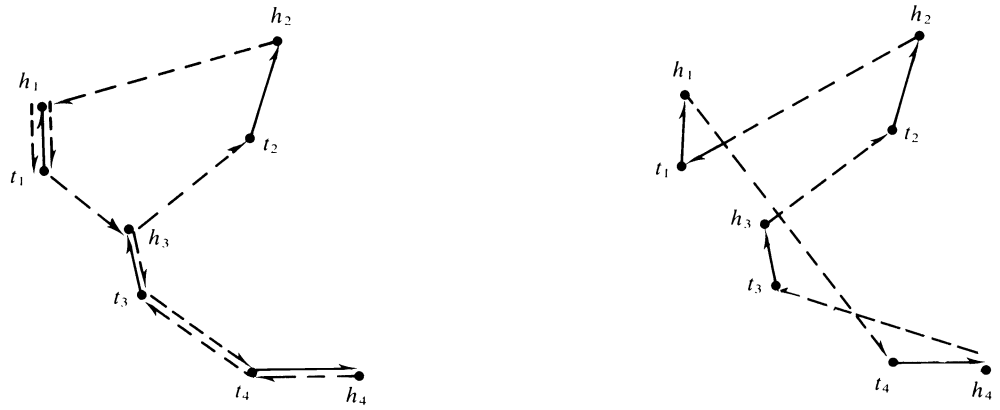


FIG. 4. *Postprocessing a tour.*

We now verify that the tours created by LARGEARCS and SMALLARCS do in fact traverse all of the arcs, and we analyze the cost of the tours in the worst case.

LEMMA 1. *Algorithm* LARGEARCS *produces a tour which traverses all of the arcs, and whose cost is at most* $3C^* - 2C_A$.

Proof. The bipartite matching of heads and tails of arcs in step 1 produces a set of edge disjoint cycles, each of which is consistent with arc directions. The indegree and outdegree of each nodes are thus equal. The spanning tree edges created in steps 3 and 4 connect the cycles. Since two of each spanning edge are added, with one edge having the opposite direction from the other, the indegree and outdegree of each vertex in G'' are still equal. Thus step 6 will produce a tour from $E'' \cup A$.

The cost of the matching, including the arcs, in steps 1 and 2 will be at most C^*. An optimum tour is in fact a bipartite matching of heads and tails of arcs, and can be no smaller than the minimum cost bipartite matching. The cost of the spanning tree edges in steps 3 and 4 must be smaller than $C^* - C_A$, since all arcs must be included in the optimum tour. Step 5 doubles the spanning edges, so that the tour produced will have cost at most $3C^* - 2C_A$. □

LEMMA 2. *Algorithm* SMALLARCS *produces a tour which traverses all of the arcs, and whose cost is at most* $(3/2)C^* + (1/2)C_A$.

Proof. Step 1 has the effect of collapsing each arc to a single node. Steps 2 and 3 create a connected graph in which all nodes have even degree. A node actually represents two vertices, the head and the tail of an arc. If the degree of a node is even, then both vertices must either be of even degree or odd degree. If odd, then step 4 adds an edge which makes both vertices even, so that after completion of step 4, all vertices are of even degree. Steps 5 and 6 show how to augment the graph to allow the tour to traverse arcs in the proper direction. Given a tour which ignores arc direction, step 6 adds two edges for each incorrectly traversed arc. One edge will create a cycle with the arc, and the second edge can be traversed in a direction consistent with the tour. Since two edges are added, the vertices will remain of even degree. Thus step 7 can produce a tour from $E'' \cup A$.

Step 1 has the effect of creating a traveling salesman problem. Since the minimum inter-arc distances were selected, the cost of an optimum traveling salesman tour will be at most $C^* - C_A$. Steps 2 and 3 are Christofides' approximation algorithm for the traveling salesman problem, which guarantees a solution no worse than 3/2 times optimal. Thus the cost of the spanning edges in step 2 and the matching edges in step 3 is at most 3/2 times optimum. The cost of the edges added in step 4 is $C_A - C'_A$, since no arc is shorter than its corresponding edge. The cost of the extra edges added in step 6 is

at most $2 * (1/2)C'_A$. The cost of all edges added in steps 4 and 6 is at most $C_A - C'_A + 2 * (1/2)C'_A = C_A$. The cost of the arcs, spanning edges, matching edges, and additional edges is $C_A + (3/2)(C^* - C_A) + C_A$. □

THEOREM 1. *If C^* is the cost of an optimal tour for the stacker-crane problem, and $\hat{C}$ is the cost of the tour generated by Algorithm* CRANE, *then*

$$\hat{C}/C^* \leqq 9/5.$$

The time complexity of Algorithm CRANE *is* $O(\max\{|V|^3, |A|^3\})$.

Proof. If $C_A \geqq (3/5)C^*$, consider the results of Algorithm LARGEARCS, from Lemma 1.

$$\hat{C}/C^* \leqq (3C^* - 2C_A)/C^*$$
$$\leqq (3C^* - (6/5)C^*)/C^* = 9/5.$$

If $C_A < (3/5)C^*$, consider the results of Algorithm SMALLARCS, from Lemma 2.

$$\hat{C}/C^* \leqq ((3/2)C^* + (1/2)C_A)/C^*$$
$$\leqq ((3/2)C^* + (3/10)C^*)/C^* = 9/5.$$

We now consider the time complexity of CRANE. Algorithm PREPROCESS is dominated by the all shortest paths algorithm [5], which requires $O(|V|^3)$ time. Algorithm LARGEARCS is dominated by the algorithm for weighted bipartite matching. The weighted general matching algorithm of Edmonds and Johnson [4] as implemented by Gabow and Lawler [6] can be used and will require $O(|A|^3)$ time. Algorithm SMALLARCS is dominated by the all shortest paths algorithm, which requires $O(|A|^3)$ time, and the weighted general matching algorithm, which is no worse than $O(|V|^3)$. □

3. k-person routing problems. In this section we consider three different k-person routing problems. We first show that the recognition versions of all three problems are *NP*-complete. We then present and analyze heuristic algorithms which yield approximate solutions.

k-TSP (k-traveling salesmen problem, $k > 1$): Let $G = (V, E)$ be an undirected complete graph with a distinguished initial vertex v_s. A nonnegative integer cost function is defined on E that satisfies the triangle inequality. A k-tour is a set of k cycles that start from v_s and collectively visit every vertex.

k-CPP (k-Chinese postman problem, $k > 1$): Let $G = (V, E)$ be an undirected multigraph with a cost function defined on E. A k-route is a set of k cycles that start from v_s, and collectively cover every edge in the graph.

k-SCP (k-stacker-cranes problem, $k > 1$): Let $G = (V, E, A)$ be a mixed graph with a cost function defined on $E \cup A$. A k-tour is a set of k-cycles that start from v_s and collectively traverse every arc in the graph.

We define the cost of a k-tour as the maximum of the costs of each cycle, and an optimal k-tour is a k-tour with minimum cost. Similar definitions apply to k-route. The optimization version is thus a minimax problem and requires that given an integer $k > 1$, we find an optimal k-tour. The recognition version is to decide, given a positive integer C, whether a k-tour of cost at most C exists.

For $k > 1$, k-TSP, k-CPP, and k-SCP are all *NP*-complete. The k-partition problem, which Sahni and Gonzalez [16] have shown to be *NP*-complete, can be reduced to the k-person problems:

k-PP (k-partition problem, $k > 1$): Consider a multiset $S = \{a_1, \cdots, a_m\}$, with

$A = \sum_{i=1}^{m} a_i$ divisible by k. Decide if there exists a partition $S_1, \cdots, S_k$ such that $\sum_{a \in S_j} a = A/k$ for all $1 \leqq j \leqq k$.

k-PP can be transformed into k-TSP as follows: Let $S = \{a_1, \cdots, a_m\}$ be an instance of k-PP. Form a complete graph $G = (V, E)$, where $V = \{v_s, v_1, v_2, \cdots, v_m\}$. For each $v_i \neq v_s$, set $c(v_s, v_i) = a_i$. For all pairs $v_i \neq v_s$, $v_j \neq v_s$, set $c(v_i, v_j) = a_i + a_j$. Let $C = (2/k) \sum_{i=1}^{m} a_i$.

k-PP can be reduced to k-CPP as follows: Let $S = \{a_1, \cdots, a_m\}$ be an instance of k-PP. Form a multigraph with a single vertex v_s, and an edge of cost a_i for every element in S. Let $C = (1/k) \sum_{i=1}^{m} a_i$.

k-SCP is shown to be *NP*-complete by reducing k-TSP to it, performing the same transformation as in our reduction from TSP to SCP.

3.1. Approximation algorithms for k-TSP. We shall consider two basically different methods for building a k-tour. The first method is to build k subtours simultaneously, modifying a heuristic of an incremental nature that generates an approximate solution for 1-person problems. The second method is to build a k-tour by splitting a good tour for 1 person into k subtours.

For the traveling salesman problem, there are several well known heuristic algorithms that find a tour by adding vertices one by one to a subtour. Among these are the nearest neighbor, nearest insertion, cheapest insertion, and farthest insertion algorithms, which have been analyzed by Rosenkrantz, et al., [14]. We shall consider generalizations of the nearest neighbor and nearest insertion algorithms to handle k salesmen. We shall find these generalized algorithms to perform rather poorly in worst case.

Let $R = (v_{i_1}, \cdots, v_{i_m})$ be a subtour, where $v_{i_1} = v_{i_m}$. For a vertex u not in R, we define the distance from u to R by $c(u, R) = \min \{c(u, v) | v \in R\}$. The cost of inserting a vertex u between v_{i_p} and $v_{i_{p+1}}$, $1 \leqq p < m$, is $c(v_{i_p}, u) + c(u, v_{i_{p+1}}) - c(v_{i_p}, v_{i_{p+1}})$. We say that a vertex u is inserted into a subtour R if u is inserted between two vertices in R with the minimum cost, and denote it by $R \leftarrow (u)$. The cost $c(R)$ of a subtour R is the sum of costs of all edges in R. Let $R_1, \cdots, R_k$ be k subtours. We define their cost $c(R_1, \cdots, R_k)$ as the maximum of the costs of each subtour, that is, $c(R_1, \cdots, R_k) = \max_{1 \leqq i \leqq k} \{c(R_i)\}$.

ALGORITHM k-NEARINSERT.

1. Start with k subtours $R_j = (v_s, v_s)$, $1 \leqq j \leqq k$, where v_s is the start vertex.
2. For each j, $1 \leqq j \leqq k$, find a vertex u_j not currently in any subtour such that $c(u_j, R_j)$ is minimal.
3. Let i be such that $c(R_1, \cdots, R_i \leftarrow (u_i), \cdots, R_k)$ is minimum. Insert u_i into the subtour R_i.
4. If $(R_1, \cdots, R_k)$ covers all vertices, then stop. Otherwise go to step 2.

LEMMA 3. *If $\hat{C}_k$ is the cost of the largest of the k subtours generated by algorithm k-*NEARINSERT*, and C_1^* is the cost of an optimal tour for one salesman, then*

$$\hat{C}_k / C_1^* < 2.$$

Proof. Let $(R_1, \cdots, R_k)$ be a k-tour with $R_i = (v_s, v_{i_1}, \cdots, v_{i_{m_i}}, v_s)$. Let $G_i = (V_i, E_i)$ be the complete subgraph of G, where $V_i = \{v_s, v_{i_1}, \cdots, v_{i_{m_i}}\}$. Suppose the subtour R_i has been built by adding vertices $v_{p(i_1)}, v_{p(i_2)}, \cdots, v_{p(i_{m_i})}$ in this order, where p is a permutation of $(i_1, i_2, \cdots, i_m)$. Then R_i is a tour of G_i that is obtained by the 1-person nearest insertion algorithm on $G_i = (V_i, E_i)$. Thus if F_i^* is the cost of an optimal tour of G_i, then $c(R_i)/F_i^* < 2$ [14, Thm. 3]. Since $\hat{C}_k = \max_{1 \leqq i \leqq k} \{c(R_i)\}$, and $F_i^* \leqq C_1^*$ for all i, $1 \leqq i \leqq k$, then $\hat{C}_k / C_1^* < 2$. □

THEOREM 2. *If $\hat{C}_k$ is the cost of the largest of the k subtours generated by algorithm k-*NEARINSERT, *and C_k^* is the cost of the largest subtour in an optimal solution of k-*TSP, *then*

$$\hat{C}_k/C_k^* < 2k,$$

and the bound is approachable.

Proof. By Lemma 3, $\hat{C}_k < 2C_1^*$. By the triangle inequality $C_1^* \leqq kC_k^*$. Thus $\hat{C}_k/C_k^* < 2k$. To show that the bound is approachable, we consider the following example. Let G_n be a graph of n vertices shown in Fig. 5, which is taken from [14, Thm. 4]. Let the cost of all edges not shown be 2. Let $R_n' = (v_1, v_3, \cdots, v_n, v_{n-1}, v_{n-3}, \cdots, v_1)$ be a tour for the single TSP obtained by 1-NEARINSERT. Then $c(R_n') = 2(n-1)$ and $C_1^* = n$.

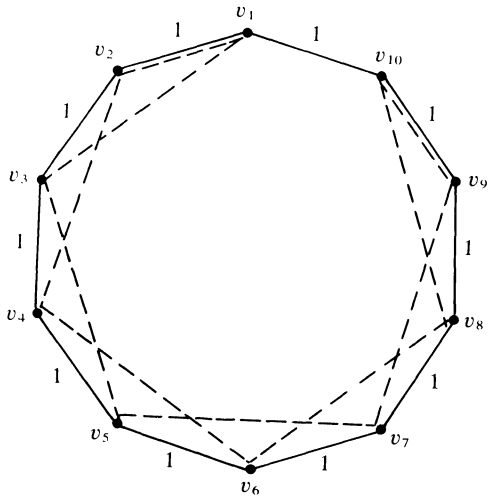


FIG. 5. G_{10} *and a* 1*-tour by* 1-NEARINSERT.

Consider the graph G_n^k which is k copies of G_{kn} with v_1 in common. We denote the vertices of the jth copy $G_{j,kn}$ as $v_1 = v_{j,1}, v_{j,2}, \cdots, v_{j,kn}$. The cost of edges in $G_{j,kn}$ is the same as in G_{kn}. The cost of edges between vertices in different copies, $v_{j,p}$ and $v_{i,m}$ is 0 if $p = m$; is 1 if $|p-m| = 1$, or if $p = 1$ and $m = kn$; and is 2 otherwise. A k-tour by k-NEARINSERT is $(R_1, \cdots, R_k)$, where for each $1 \leqq j \leqq k$, $R_j = (v_{j,1}, v_{j,3}, \cdots, v_{j,kn}, v_{j,kn-1}, v_{j,kn-3}, \cdots, v_{j,1})$ with $c(R_j) = 2(kn-1)$. Thus $\hat{C}_k = 2(kn-1)$. On the other hand, an optimal k-tour is $(R_1^*, \cdots, R_k^*)$, where $R_1^* = (v_1, v_{1,2}, v_{2,2}, \cdots, v_{k,2}, v_{1,3}, \cdots, v_{k,3}, \cdots, v_{i,n}, \cdots, v_{k,n}, v_1)$, $\cdots$, $R_k^* = (v_1, v_{1,(k-1)n+1}, \cdots, v_{k,(k-1)n+1}, \cdots, v_{1,kn}, \cdots, v_{k,kn}, v_1)$. Thus $c(R_j^*) = n+1$, and $C_k^* = n+1$. Hence

$$C_k/C_k^* = 2(kn-1)/(n+1) = 2k - 2(k+1)/(n+1)$$

which approaches $2k$ for large n. □

The following notation is used in describing the algorithm k-NEARNEIGHBOR. Let $R = (v_{i_1}, \cdots, v_{i_m})$ be a path. We call v_{i_1} the initial vertex and v_{i_m} the terminal vertex of R. A vertex u is added to R by connecting u to the terminal vertex, and is denoted by $R \leftarrow u$.

ALGORITHM k-NEARNEIGHBOR.

1. Set each of the k paths initially to the initial vertex, that is, $R_j = (v_s)$ for all j, $1 \leqq j \leqq k$.
2. For each j, $1 \leqq j \leqq k$, let u_j be the vertex nearest to the terminal vertex of R_j.
3. Let i be such that $c(R_1, \cdots, R_i \leftarrow u_i, \cdots, R_k)$ is minimum. Add u_i to R_i.
4. If there are vertices remaining to be added to a path then go to step 2. Otherwise build a k-tour from the R_j's by connecting their terminal vertices to their initial vertices.

LEMMA 4. *If $\hat{C}_k$ is the cost of the largest of the k subtours obtained by k-*NEARNEIGHBOR, *and C_1^* is the cost of an optimal tour for one salesman, then*

$$\hat{C}_k / C_1^* < (1/2) \log n + 1.$$

Proof. We use the result in [14, Thm. 1];

$$\hat{C}_1 / C_1^* \leqq (1/2)\lceil \log n \rceil + 1/2.$$

A proof similar to that for Lemma 3 then applies. □

THEOREM 3. *If $\hat{C}_k$ is as in Lemma* 4, *and C_k^* is the cost of the largest subtour in an optimal solution of k-*TSP, *then*

$$\hat{C}_k / C_k^* < (k/2) \log n + k.$$

Also, there is a graph for which $\hat{C}_k / C_k^* > (k/6) \log n$.

Proof. By the triangle inequality, $C_1^* \leqq kC_k^*$, and by Lemma 4, $\hat{C}_k/(kC_k^*) < (1/2) \log n + 1$. Thus $\hat{C}_k / C_k^* < (k/2) \log n + k$. The second assertion can be proved as follows: Consider the complete graph $\bar{G}_{m-1} = (V, E)$ in [14, Thm. 2], where $|V| = n = 2^m - 1$. Use an argument similar to that in Theorem 2 to show that $\hat{C}_k > (n/3) \log n$ and $C_k^* < 2n/k$. □

We now describe an algorithm which employs a very simple tour-splitting heuristic. Given a tour for one traveling salesman, the algorithm splits the tour into k subtours of more or less equal cost. Obviously, the worst-case behavior depends on the (generally) nonoptimal tour which is split into k subtours. When a tour obtained by Christofides' algorithm is used, this simple heuristic is far superior to the heuristics already analyzed, in terms of worst-case behavior. In the algorithm, $c_{\max}$ denotes $\max c(v_1, v_i)$, and for any path R, $t(R)$ denotes its terminal vertex. An example of splitting a 1-tour into a 3-tour is shown in Fig. 6.

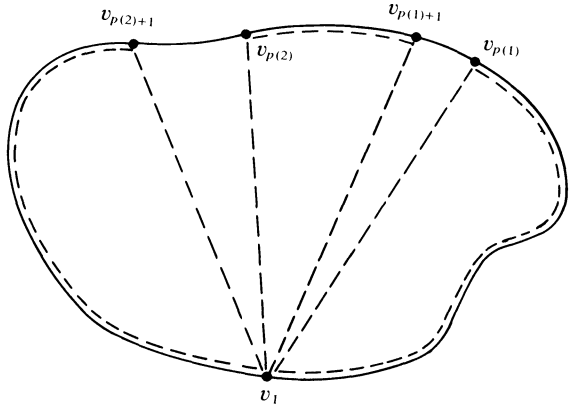


FIG. 6. *Splitting a* 1*-tour into a* 3*-tour.*

ALGORITHM k-SPLITOUR.

1. Find a 1-tour $R=(v_1, v_2, \cdots, v_n, v_1)$ with $c(R)=L$, where v_1 is the initial vertex.
2. For each j, $1 \leqq j < k$, find the last vertex $v_{p(j)}$ such that the cost of the path from v_1 to $v_{p(j)}$ along R is not greater than $(j/k)(L-2c_{\max})+c_{\max}$.
3. Obtain the k-tour by forming k subtours as $R_1=(v_1, \cdots, v_{p(1)}, v_1)$, $R_2=(v_1, v_{p(1)+1}, \cdots, v_{p(2)}, v_1), \cdots, R_k=(v_1, v_{p(k-1)+1}, \cdots, v_n, v_1)$.

THEOREM 4. *If $\hat{C}_k$ is the cost of the largest of the k subtours generated by Algorithm* k-SPLITOUR, *and C_k^* is the cost of the largest subtour in an optimal solution of* k-TSP, *then*

$$\hat{C}_k/C_k^* \leqq e+1-1/k$$

where e is the bound for the single traveling salesman algorithm.

Proof. The costs of the paths from v_1 to $v_{p(1)}$ and from $v_{p(k-1)+1}$ to v_1 along R are each no greater than $(1/k)(L-2c_{\max})+c_{\max}$. For each j, $1 \leqq j \leqq k-2$, the cost of the path from $v_{p(j)+1}$ to $v_{p(j+1)}$ is no greater than $(1/k)(L-2c_{\max})$. Thus for each j, $1 \leqq j \leqq k$, the cost of the tour R_j does not exceed $(1/k)(L-2c_{\max})+2c_{\max}$. Therefore,

$$\begin{aligned}\hat{C}_k = \max c(R_j) &\leqq (1/k)(L-2c_{\max})+2c_{\max} \\ &\leqq (L/k)+2(1-1/k)c_{\max}.\end{aligned}$$

Due to the triangle inequality, $C_k^* \geqq (1/k)C_1^*$ and $c_{\max} \leqq (1/2)C_k^*$. Using these with $L \leqq eC_1^*$, we obtain $\hat{C}_k \leqq (1/k)ekC_k^* + 2(1-1/k)(1/2)C_k^*$, which yields $\hat{C}_k/C_k^* \leqq e+1-1/k$. □

COROLLARY 1. *There is an $O(|V|^3)$ approximation algorithm for* k-TSP *with bound*

$$\hat{C}_k/C_k^* \leqq 5/2-1/k.$$

Proof. In step 1, we find a 1-tour R using Christofides' algorithm [2]. Since R can be found in $O(|V|^3)$ time, and $c(R)/C_1^* \leqq 3/2 = e$, the result is immediate. □

The bound in the corollary is tight if the algorithm in the proof is employed. An example for $k=2$ which realizes the worst case bound of $5/2-1/2=2$ is shown in Fig. 7. All edges shown in Fig. 7(a) have unit cost, while all other edge costs are determined by the shortest path along the edges shown. An optimum 2-tour is shown in dotted lines in Fig. 7(a). Figure 7(b) shows a 1-tour that could be produced by Christofides' algorithm in solid lines, and a 2-tour that could result from our algorithm in dotted lines.

3.2. Approximation algorithms for k-CPP and k-SCP. The tour splitting heuristic yields a good bound for the Chinese postman problem, since an optimal 1-route for the Chinese postman problem can be obtained in polynomial time by using the algorithm of Edmonds and Johnson [4]. Let $R=(v_1, e_{i1}, v_{i2}, e_{i2}, \cdots, v_{im}, e_{im}, v_1)$ be the 1-route so obtained. Let $L=c(R)$ and let $R_{v_{in}}$ denote the path $(v_1, e_{i1}, v_{i2}, \cdots, v_{in})$, with $n \leqq m$. We denote the cost of a shortest path from a vertex v to u by $s(v, u)$. Define $s_{\max}$ as $(1/2)\max\{s(v_1, v_{ij})+c(v_{ij}, v_{ij+1})+s(v_{ij+1}, v_1)\}$ and for each j, $1 \leqq j < k$, $L_j = (j/k)(L-2s_{\max})+s_{\max}$. Figure 8 shows the building of a k-route, and Fig. 9 shows a completed 4-route.

ALGORITHM k-POSTMEN.

1. Find an optimal 1-route $R=(v_1, e_{i1}, v_{i2}, \cdots, v_{im}, e_{im}, v_1)$ using the algorithm in [4], where v_1 is the start vertex.
2. For each j, $1 \leqq j \leqq k$, find the last vertex $v_{p'(j)}$ such that $c(R_{v_{p'(j)}}) \leqq L_j$.
3. Let $r_j = L_j - c(R_{v_{p'(j)}})$. For each j, $1 \leqq j < k$, if $r_j + s(v_{p'(j)}, v_1) \leqq c(v_{p'(j)}, v_{p'(j)+1}) - r_j + s(v_{p'(j)+1}, v_1)$, then $v_{p(j)} = v_{p'(j)}$. Otherwise $v_{p(j)} = v_{p'(j)+1}$.

4. Let $R_1 = (v_1, e_{i1}, v_{i2}, \cdots, v_{p(1)})$, $R_2 = (v_{p(1)}, \cdots, v_{p(2)}), \cdots, R_k = (v_{p(k-1)}, \cdots, v_1)$. Build the k-route by connecting v_1 to both the initial and terminal vertices of the R_j's with shortest paths to transform R_j into a subroute.

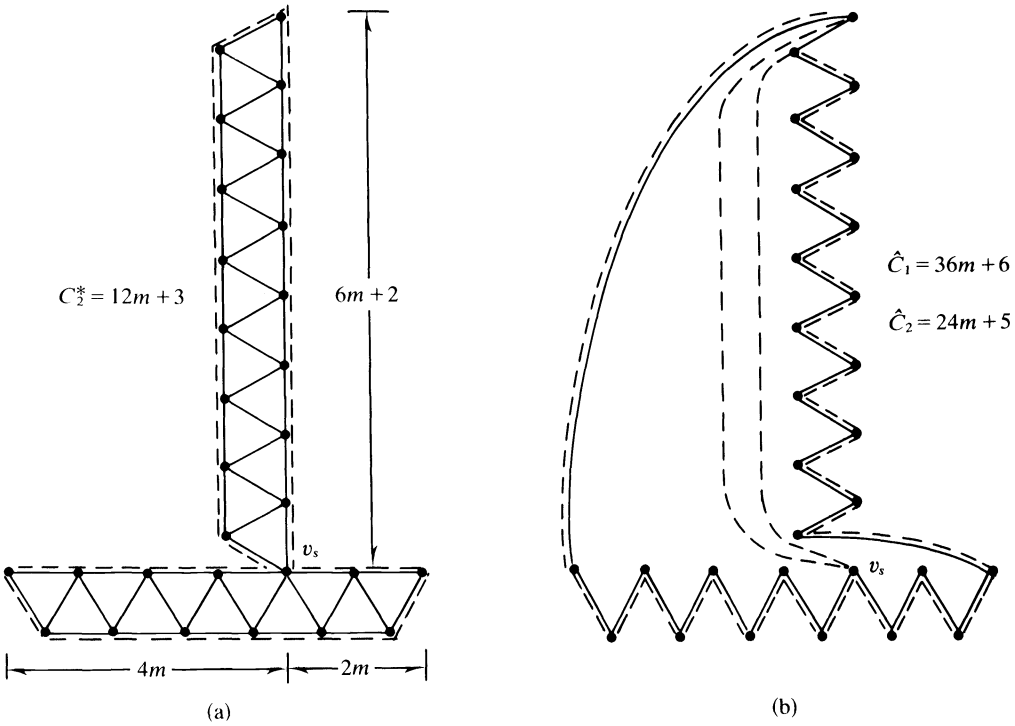


FIG. 7. *Worst case for* 2-TSP.

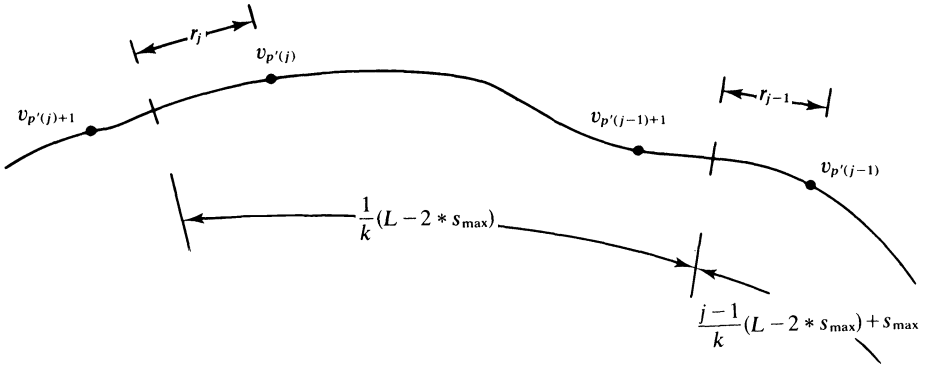


FIG. 8. *Building a k-route.*

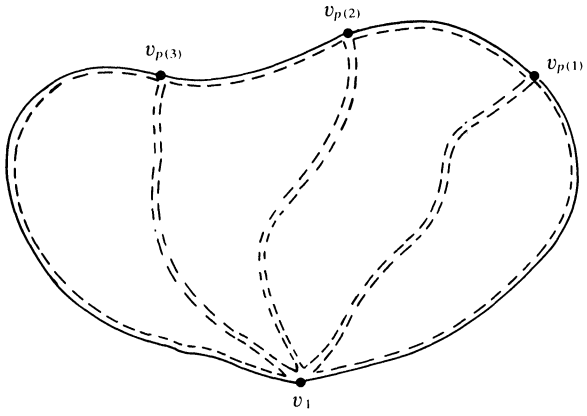


FIG. 9. *A 4-route.*

We note that k-POSTMEN is more complicated than k-SPLITOUR. If the split point falls somewhere in the middle of an edge, then the algorithm must decide in which subtour to include the edge.

THEOREM 5. *The algorithm* k-POSTMEN *produces* $\hat{C}_k$ *in* $O(|V|^3)$ *time such that*

$$\hat{C}_k/C_k^* \leqq 2-1/k.$$

Proof. We consider the jth subroute R_j, $1 \leqq j \leqq k$. Since each edge in the graph must be covered, $s_{\max} \leqq (1/2)C_k^*$. By the definition of $s_{\max}$,

$$s(v_1, v_{p'(j)})+c(v_{p'(j)}, v_{p'(j)+1})+s(v_{p'(j)+1}, v_1) \leqq 2s_{\max}.$$

Hence

$$\min\{s(v_1, v_{p'(j)})+r_j, c(v_{p'(j)}, v_{p'(j)+1})-r_j+s(v_{p'(j)+1}, v_1)\} \leqq s_{\max}.$$

Similarly,

$$\min\{s(v_1, v_{p'(j-1)})+r_{j-1}, c(v_{p'(j-1)}, v_{p'(j-1)+1})-r_{j-1}+s(v_{p'(j-1)+1}, v_1)\} \leqq s_{\max}.$$

The worst case for the jth subroute R_j is when it starts from v_1, reaches $v_{p'(j-1)}$, continues to $v_{p'(j)+1}$ along R and finally back to v_1. But in this case $s(v_1, v_{p'(j-1)})+r_{j-1} \leqq s_{\max}$ and $c(v_{p'(j)}, v_{p'(j)+1})-r_j+s(v_{p'(j)+1}, v_1) \leqq s_{\max}$. Thus $c(R_j) \leqq s_{\max}+(1/k)(L-2s_{\max})+s_{\max}$. Since $L \leqq kC_k^*$ and $s_{\max} \leqq (1/2)C_k^*$,

$$c(R_j) \leqq 2s_{\max}(1-1/k)+C_k^* \leqq C_k^*(1-1/k)+C_k^* = C_k^*(2-1/k).$$

Other cases for R_j can easily be shown to satisfy the above inequality. Hence, we obtain $\hat{C}_k/C_k^* \leqq 2-1/k$. □

Finally, we present a tour splitting algorithm for the k-stacker-cranes problem. The algorithm uses the same methods as k-SPLITOUR and k-POSTMEN with differences in detail. If a split point falls on an edge, a similar procedure is used as in k-SPLITOUR. If the split point falls on an arc, then a similar procedure as in k-POSTMEN is used. $R=(v_1, v_{i2}, \cdots, v_{im}, v_1)$, $R_{v_{in}}$, L, $c_{\max}$ and $L_j=(j/k)(L-2c_{\max})+c_{\max}$ are as previously defined.

ALGORITHM k-CRANES.

1. Find a 1-tour $R=(v_1, v_{i2}, \cdots, v_{im}, v_1)$ for the single crane problem with $c(R)=L$, where v_1 is the initial vertex.
2. For each j, $1 \leqq j < k$, find the last vertex $v_{p'(j)}$ such that $c(R_{v_{p'(j)}}) \leqq L_j$.

3. Let $r_j = L_j - c(R_{v_{p'(j)}})$. For each j, $1 \leqq j < k$, if $(v_{p'(j)}, v_{p'(j)+1})$ is an edge, then $v_{p(j)} = v_{p'(j)}$, and the terminal vertex of R_j is $v_{p'(j)}$ and the initial vertex of R_{j+1} is $v_{p(j)+1}$. Suppose $(v_{p'(j)}, v_{p'(j)+1})$ is an arc. If

$$c(v_1, v_{p'(j)}) + r_j \leqq c(v_{p'(j)}, v_{p'(j)+1}) - r_j + c(v_{p'(j)+1}, v_1),$$

then $v_{p(j)} = v_{p'(j)}$ and the initial vertex of R_{j+1} is $v_{p(j)}$. Also if $(v_{p(j)-1}, v_{p(j)})$ is an arc, then the terminal vertex of R_j is $v_{p(j)}$, and if it is an edge then the terminal vertex of R_j is $v_{p(j)-1}$. If

$$c(v_1, v_{p'(j)}) + r_j > c(v_{p'(j)}, v_{p'(j)+1}) - r_j + c(v_{p'(j)+1}, v_1),$$

then $v_{p(j)} = v_{p'(j)+1}$ and the terminal vertex of R_j is $v_{p(j)}$. Also, if $(v_{p(j)}, v_{p(j)+1})$ is an arc, then the initial vertex of R_{j+1} is $v_{p(j)}$ and otherwise the initial vertex of R_{j+1} is $v_{p(j)+1}$.

4. For each j, $1 \leqq j \leqq k$, construct the jth subtour R_j by connecting v_1 to the initial vertex of R_j and the terminal vertex of R_j to v_1 with direct edges.

THEOREM 6. *Algorithm* k-CRANES *produces* $\hat{C}_k$ *such that*

$$\hat{C}_k / C_k^* \leqq e + 1 - 1/k$$

where e *is the bound for a* 1*-crane algorithm.*

Proof. A proof which is a combination of the proofs of Theorems 4 and 5 applies. □

COROLLARY 2. *There is an approximation algorithm of* $O(\max\{|V|^3, |A|^3\})$ *time complexity for* k-SCP *such that*

$$\hat{C}_k / C_k^* \leqq 14/5 - 1/k.$$

Proof. In Step 1, a 1-tour may be obtained by using the algorithm in § 2, for which $e = 9/5$. □

4. Conclusion. In developing an approximation algorithm for the stacker-crane problem, we have applied a mixed strategy approach. Our algorithm consists of two efficient algorithms, each of which handles certain extreme cases well, so that the overall behavior of the algorithm is improved. We have formulated k-person routing problems as minimax problems. We have presented a tour-splitting heuristic, a method by which approximate solutions for multirouting problems can be obtained from good approximate solutions of simple routing problems.

REFERENCES

[1] M. BELLMORE AND S. HONG, *Transformation of the multisalesmen problem to the standard traveling salesman problem*, J. Assoc. Comput. Mach., 21 (1974), pp. 500–504.

[2] N. CHRISTOFIDES, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Management Sciencies Research Report No. 388, Carnegie–Mellon University, Pittsburgh, PA, (1976).

[3] S. A. COOK, *The complexity of theorem proving procedures*, 3rd Symposium on Theory of Computing (1971), pp. 151–158.

[4] J. EDMONDS AND E. L. JOHNSON, *Matching, Euler tours and the Chinese postman*, Math. Programming, 5 (1973), pp. 88–124.

[5] R. FLOYD, *Algorithm 97, shortest path*, Comm. ACM, 5 (1962), p. 345.

[6] H. GABOW AND E. L. LAWLER, *An efficient implementation of Edmonds' algorithm for maximum weight matching on graphs*, TR CU-CS-075-75, Dept. of Computer Science, Univ. of Colorado, 1975.

[7] M. R. GAREY AND D. S. JOHNSON, *Approximation Algorithms for Combinatorial Problems: An Annotated Bibliography*, Algorithms and Complexity: Recent Results and New Directions, J. F. Traub, ed., Academic Press, New York, 1976, pp. 41–52.

[8] O. H. IBARRA AND C. E. KIM, *Fast approximation algorithms for the knapsack and sum of subset problems*, J. Assoc. Comput. Mach., 22 (1975), pp. 463–468.
[9] D. S. JOHNSON, *Approximation algorithms for combinatorial problems*, J. Comput. Systems Sci., 9 (1974) pp. 256–278.
[10] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher eds., Plenum Press, New York, 1972, pp. 85–104.
[11] C. E. MILLER, A. W. TUCKER AND R. A. ZEMLIN, *Integer programming formulation of traveling salesman problem*, J. Assoc. Comput. Mach., 7 (1960), pp. 362–329.
[12] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Some complexity results for the traveling salesman problem*, 8th Symposium on Theory of Computing, 1976, pp. 1–9.
[13] D. J. ROSENKRANTZ, private communication, 1976.
[14] D. J. ROSENKRANTZ, R. E. STEARNS AND P. M. LEWIS, *Approximate algorithms for the traveling salesperson problem*, 15th Symposium on Switching and Automata Theory (1974), pp. 33–42.
[15] S. K. SAHNI, *Approximate algorithms for the* 0/1 *knapsack problem*, J. Assoc. Comput. Mach., 22 (1975), pp. 115–124.
[16] S. K. SAHNI AND T. GONZALEZ, *P-complete approximation problems*, J. Assoc. Comput. Mach., 23 (1976), pp. 555–565.

SIAM J. COMPUT.
Vol. 7, No. 2, May 1978

RUDIMENTARY PREDICATES AND RELATIVE COMPUTATION*

CELIA WRATHALL†

Abstract. A class of languages RUD derived from the class of rudimentary relations is studied. Two characterizations of RUD are established, one using linear-time relative computation and the other using language-theoretic operations. Also, some connections between RUD and classes of languages defined by resource-bounded Turing machines are given.

Key words. rudimentary predicates, formal languages, relative acceptance by oracle machines, characterization using closure properties

1. Introduction. The rudimentary relations were defined in [23] as a string analogue of the constructive arithmetic relations. The class of rudimentary relations is the smallest class containing the concatenation relation (which holds for (x, y, z) if and only if $xy = z$) and closed under the Boolean operations, explicit transformation and linear-bounded quantification, while the constructive arithmetic relations are defined from the addition and multiplication relations using the same operations. When numbers are represented by strings, the classes of rudimentary and constructive arithmetic relations are equal [2]. In this paper, the rudimentary relations are viewed as a class RUD of languages through an encoding of relations to languages, and two characterizations of RUD are established. These characterizations develop a natural analogy between the class of rudimentary relations and the class of arithmetical relations, which may be defined as the smallest class containing the recursive relations and closed under the Boolean operations, explicit transformation and quantification [21].

Machine models for language acceptance generally operate by using symbol-by-symbol scanning and manipulation of strings, and the basic operation on strings is concatenation. Thus the class RUD, based on concatenation, is a natural candidate for a class of languages to study in connection with classes of languages defined by resource-bounded Turing machines or other automata, or closure properties.

Resource-bounded relative computation has arisen recently as a method of formalizing the notion of "efficient" reduction of one decision problem to another. This process can be viewed both as a restricted Turing reducibility [21] and as a means for transforming languages or for generating a class of languages from a given language. Taking the latter view, it is shown here that RUD is the smallest (nonempty) class of languages that is closed under relative acceptance in linear time by nondeterministic "oracle machines" (Theorem 3.10). Notice that the arithmetical sets may be defined in this way using unrestricted relative acceptance [21]. Further, the class RUD is shown to be the smallest class containing the regular sets and the language $\{0^n 1^n : n \geqq 0\}$ and closed under certain operations (Theorem 3.12). These characterizations of RUD give rise to conditions for the rudimentary relations to be equal to Grzegorczyk's class $\mathscr{E}_*^2$ [11].

To simplify the proofs of the characterizations of RUD, a sequence of classes of languages σ_k, $k \geqq 0$, is defined, with $\text{RUD} = \bigcup_k \sigma_k$. Here σ_0 is the class containing

* Received by the editors October 25, 1976, and in final revised form August 18, 1977.

† Department of System Science, University of California, Los Angeles, California. Now at Department of Mathematics, University of California, Santa Barbara, California 93106. The results presented here represent a portion of the author's Ph.D. dissertation at Harvard University [26] which was supervised by Professor Ronald V. Book. The work was supported in part by the National Science Foundation under Grants MCS 76-05744 and DCR 74-15091.

only the empty set and a language is in σ_{k+1} if it can be recognized nondeterministically in linear time relative to some language in σ_k. The structure $\sigma_0 \subseteq \sigma_1 \subseteq \sigma_2 \subseteq \cdots$ is called the "linear hierarchy." The classes in the linear hierarchy can also be defined using alternations of suitably bounded quantifiers (Theorem 4.2).

In § 2, notation is reviewed and the definitions of the rudimentary relations and of oracle machines are given. Section 3 contains the facts about oracle machines and about the class RUD of rudimentary languages that are used in proving the characterizations, followed by the proofs of the characterizations. In § 4, further results about the linear hierarchy are presented.

2. Preliminaries. This section begins with a review of some definitions and notation from formal language theory and of notation for classes of languages accepted by resource-bounded Turing machines. The definitions of the class of rudimentary relations and a derived class RUD of languages are then discussed, and certain basic properties of RUD are established. Finally, oracle machines, the model used here for time-bounded relative computation, are described.

2.1. If S is a finite set of symbols, called an *alphabet*, then S^* denotes the free monoid generated by the symbols in S. The elements of S^* are strings (finite sequences) of symbols from S; the operation in S^* is termed *concatenation* and is denoted by juxtaposition of the strings. The identity element of S^* is the empty string, denoted by e. Thus $S^* = \{e\} \cup \{s_1 \cdots s_n : n \geqq 1,\ s_1, \cdots, s_n \in S\}$. If $n \geqq 1$ and $x = s_1 \cdots s_n$ is a string in S^* ($s_i \in S$ for $1 \leqq i \leqq n$), then the *length* of x, denoted $|x|$, is n; $|e| = 0$. A *language* is a subset of S^* from some alphabet S.

The term "homomorphism" is used for monoid homomorphisms $h: S^* \to T^*$ where S and T are alphabets. If $L \subseteq S^*$, then $h(L) = \{h(x) : x \in L\}$ is the image of L under the homomorphism h. Certain restricted types of homomorphisms are of interest:

(i) h is *length-preserving* if for all $a \in S$, $|h(a)| = |a| = 1$;

(ii) h is *nonerasing* if for all $a \in S$, $|h(a)| \geqq 1$; and

(iii) h *performs linear erasing on a language* $L \subseteq S^*$ if there is constant k such that for all $w \in L$, $|w| \leqq k \cdot \max\{|h(w)|, 1\}$.

If $h: S^* \to T^*$ is a homomorphism, then the "inverse homomorphism" h^{-1} is the mapping of T^* to subsets of S^* defined by $h^{-1}(y) = \{x \in S^* : h(x) = y\}$. If $L \subseteq T^*$, then $h^{-1}(L) = \{x \in S^* : h(x) \in L\}$.

If L_1, L_2 are languages, then the *product* of L_1 with L_2 is the language $L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$. The Kleene * of L_1 is the language $L_1^* = \{e\} \cup \{x_1 x_2 \cdots x_k : k \geqq 1, x_i \in L \text{ for } 1 \leqq i \leqq k\}$. The Kleene + of L_k is $L_1^+ = L_1^* - \{e\}$.

A family of languages $\mathscr{L}$ is closed under complementation if whenever $L \in \mathscr{L}$ and S is any alphabet such that $L \subseteq S^*$ also $S^* - L = \{x \in S^* : x \notin L\}$ is in $\mathscr{L}$. If $\mathscr{L}$ is a family of languages, then co-$\mathscr{L}$ denotes the family containing exactly the complements of languages in $\mathscr{L}$; that is, co-$\mathscr{L} = \{S^* - L : L \subseteq S^* \text{ in } \mathscr{L}\}$.

The family of *regular sets* is the smallest family of languages containing the finite languages and closed under the operations of union, product and Kleene *. The reader is assumed to be familiar with the basic properties of the regular sets, and of the context-free languages (see, e.g., [13]).

The *cross-product* of two sets U and V is denoted by $U \times V$. The cross-product of V with itself m times is $[V]^m = \{(v_1, \cdots, v_m) : v_i \in V \text{ for } 1 \leqq i \leqq m\}$; thus $[S^*]^m$ is the set of m-tuples of strings over an alphabet S. A subset of $[S^*]^m$ is an m-ary string relation.

In order to use the tools of formal language theory to investigate string relations, we combine a tuple of strings into a single string, as follows. Suppose S is an alphabet and # is a symbol not in S. If $x_1, \cdots, x_m$ are strings in S^* ($m \geqq 1$), then $\langle x_1, \cdots, x_m \rangle$ is a string over the alphabet $[S \cup \{\#\}]^m$ of length $n = \max\{|x_i| : 1 \leqq i \leqq m\}$ given by $\langle x_1, \cdots, x_m \rangle = z_1 \cdots z_n$ where, for $1 \leqq j \leqq n$, $z_j = [z_{j1}, \cdots, z_{jm}]$, each $z_{jk} \in S \cup \{\#\}$ and, for $1 \leqq i \leqq m$, $z_{1i} z_{2i} \cdots z_{ni} = x_i \#^{n-|x_i|}$. For example, suppose $S = \{0, 1\}$; then $\langle 000, 101, 11 \rangle = [0, 1, 1][0, 0, 1][0, 1, \#]$. The intention of this encoding, due to Myhill [17], is to describe the writing of m strings on m "tracks" of a Turing tape, so this example should be read as

$$\langle 000, 101, 11 \rangle = \begin{matrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & \# \end{matrix}.$$

If R is an m-ary relation over S, the language encoding of R is $\theta(R) = \{\langle x_1, \cdots, x_m \rangle : (x_1, \cdots, x_m) \in R\}$. Notice that if $m = 1$ (i.e., if R is a language), then $\theta(R) = R$.

The choice of this "parallel" encoding was based on technical considerations; the results here will hold for other reasonable encodings, e.g., for the sequential encoding $\sigma(R) = \{x_1 \# \cdots x_m \# : (x_1, \cdots, x_m) \in R\}$.

The model for Turing acceptor used here has a (one- or two-way) read-only input tape and multiple work tapes (see, e.g., [13]). It may be deterministic or nondeterministic. The language accepted by a Turing machine M is denoted by $L(M)$.

Let $t: N \to N$ and $s: N \to N$ be nondecreasing functions, with $t(n) \geqq n$ for all $n \in N$. (Here N denotes the natural numbers.) A Turing acceptor is said to *operate in time* $t(n)$ if for every input string x every computation of the machine on x takes at most $t(|x|)$ steps. A Turing acceptor is said to *operate in space* $s(n)$ if for any input x no more than $s(|x|)$ tape squares are visited on any one of the work tapes during any computation on x. We use $\log n$ to denote the function whose value at n is the length of the binary representation of n.

The notation used here for families of languages defined by resource-bounded Turing acceptors is as follows. For a time bounding function $t(n)$:

$$\text{DTIME}(t(n)) = \{L(M) | M \text{ is a deterministic Turing acceptor that operates in time bound } t(n)\};$$

$$\text{NTIME}(t(n)) = \{L(M) | M \text{ is a nondeterministic Turing acceptor that operates in time bound } t(n)\}.$$

It is known [4] that for any constant c and any Turing acceptor that operates in time cn (i.e., in linear time) there is a nondeterministic Turing acceptor that accepts the same language and operates in time n (i.e., in real-time); hence $\text{NTIME}(n) = \bigcup \{\text{NTIME}(cn) : c \geqq 1\}$. Also, define $\text{DTIME}(\text{lin}) = \bigcup \{\text{DTIME}(cn) | c \geqq 1\}$.

For a space bounding function $s(n)$:

$$\text{DSPACE}(s(n)) = \{L(M) | M \text{ is a deterministic Turing acceptor that operates in space bound } s(n)\};$$

$$\text{NSPACE}(s(n)) = \{L(M) | M \text{ is a nondeterministic Turing acceptor that operates in space bound } s(n)\}.$$

The class $\text{DSPACE}(n)$ is equal to Grzegorczyk's class $\mathscr{E}_*^2$ [19]; that is, it contains exactly the encodings of those relations.

2.2. The class of rudimentary relations (or attributes) was first defined by Smullyan [23]. The following definition, in a more general form than the original, is based on that given in [15].

DEFINITION. 1) The following operations are the *rudimentary operations.*

(i) *Boolean operations*: The Boolean operations are union, intersection and difference of relations over the same alphabet.

(ii) *Explicit transformation*: An explicit transformation of a relation R is obtained by adding redundant variables, identifying or permuting variables, or substituting a string for a variable. That is, $Q \subseteq [S^*]^n$ is defined by explicit transformation from $R \subseteq [S^*]^m$ if and only if $(x_1, \cdots, x_n) \in Q \Leftrightarrow (t_1, \cdots, t_m) \in R$ where, for $1 \leqq i \leqq m$, t_i is a string containing symbols from S or the variables $x_1, \cdots, x_n$ (or both). For example, t_1 might be x_2 or a string $w \in S^*$ or $x_1 w x_2$.

(iii) *Bounded existential quantification*: Suppose $R \subseteq [S^*]^{n+1}$, $n \geqq 0$. A relation $Q \subseteq [S^*]^{n+1}$ is defined by bounded existential quantification from R if and only if $Q = \{(x_1, \cdots, x_n, y)$: for some $z \in S^*$ such that $|z| \leqq |y|$, $(x_1, \cdots, x_n, z) \in R\}$.

2) For an alphabet S, the *concatenation* relation C_S on S^* is defined by $C_S = \{(x, y, z) : x, y, z \in S^*, xy = z\}$. The class of *rudimentary relations over S* is the smallest class of string relations containing C_S and closed under the rudimentary operations. A relation will be said to be *rudimentary* if it is rudimentary over some alphabet S.

The definition of the rudimentary relations in [23] restricts them to the alphabet $\{1, 2\}$ and in [2], for each $m \geqq 1$, separate classes of m-rudimentary relations are defined; in both these definitions, the operation of explicit transformation is restricted to a simpler form, in which each term t_i is either a constant string or one of the variables. It is shown in [15] that, if S has m symbols, then the class of relations that are rudimentary over S (as defined above) coincides with the m-rudimentary relations. Strings on an alphabet S with m symbols may be viewed as the m-adic notations of natural numbers, giving rise to a one-to-one correspondence between S^* and N. Under this correspondence, the class of rudimentary relations is equal to the class of constructive arithmetic relations [2], [23] and is contained in Grzegorczyk's class $\mathscr{E}^0_*$ [11]. (It is not known whether the containment $\mathscr{E}^0_* \subseteq \mathscr{E}^2_*$ is proper.)

DEFINITION. RUD is the class of language-encodings of rudimentary relations: $\text{RUD} = \{\theta(R) : R \text{ a rudimentary relation}\}$.

PROPOSITION 2.1 [15]. *A string relation R is rudimentary if and only if the language $\theta(R)$ is rudimentary. Hence, the class* RUD *is exactly the class of unary rudimentary relations ("rudimentary events").*

Proof. For $m \geqq 1$, let $T_m = \{(x_1, \cdots, x_m, z) : z = \langle x_1, \cdots, x_m \rangle\}$ (where some fixed alphabet is assumed). Then T_m is rudimentary for each m; the proof relies on the fact that such relations as "$|x| = |y|$" and "a is the $|x|$th symbol in y" are rudimentary. Recall that $|\langle x_1, \cdots, x_m \rangle| = \max\{|x_i| : 1 \leqq i \leqq m\}$. For any relation R, $\theta(R) = \{z$: there exists $x_1, \cdots, x_m$ with $|x_i| \leqq |z|$ for $1 \leqq i \leqq m$ such that $(x_1, \cdots, x_m, z) \in T_m$ and $(x_1, \cdots, x_m) \in R\}$ and $R = \{(x_1, \cdots, x_m)$: there exists z such that $|z| \leqq \max_i |x_i|$, $z \in \theta(R)$ and $(x_1, \cdots, x_m, z) \in T_m\}$. The types of quantification used in these expressions can be replaced with bounded existential quantification (and explicit transformation and union); therefore R and $\theta(R)$ can be defined from each other and T_m by use of rudimentary operations. □

The following proposition expresses the effect on $\theta(R)$ of applying rudimentary operations to a relation R.

PROPOSITION 2.2. *If a relation Q is defined using rudimentary operations from a relation R, then $\theta(Q)$ can be defined from $\theta(R)$ and languages in* DTIME(lin) *by application of Boolean operations, length-preserving homomorphism and inverse homomorphism. Specifically:*

1) *If* $R, R' \subseteq [S^*]^n$ *and* $\# \notin S$, *then* $\theta(R \cup R') = \theta(R) \cup \theta(R')$, $\theta(R \cap R') = \theta(R) \cap \theta(R')$ *and* $\theta(R - R') = \theta(R) - \theta(R')$;

2) *If* Q *is defined from* R *by bounded existential quantification, then there exist a regular set* L_0, *a homomorphism* h *and a length-preserving homomorphism* h' *such that* $\theta(Q) = h'(L_0 \cap h^{-1}(\theta(R)))$; *and*

3) *If* Q *is an explicit transformation of* R, *then there exist a language* $L_1 \in$ DTIME(lin), *a regular set* L_2 *and homomorphisms* h_1 *and* h_2, *with* h_1 *length-preserving, such that* $\theta(Q) = h_1(L_1 \cap h_2^{-1}(\theta(R))) \cup L_2$.

Proof. Verification of 1) is straightforward.

For 2), suppose that Q is defined by bounded existential quantification from R, with $Q, R \subseteq [S^*]^{n+1}$ and $\theta(Q), \theta(R) \subseteq T^*$ where $T = [S \cup \{\#\}]^{n+1}$. Let $L_0 = \{\langle x_1, \cdots, x_n, y, z\rangle : |z| \leqq |y|, x_i, y, z \in S^*\}$; then L_0 is a regular set. Let $h: ([S \cup \{\#\}]^{n+2})^* \to T^*$ be the homomorphism determined by defining $h([b_1, \cdots, b_{n+2}]) = e$ if $b_1 = \cdots = b_n = b_{n+2} = \#$ and $h([b_1, \cdots, b_{n+2}]) = [b_1, \cdots, b_n, b_{n+2}]$ otherwise. Let h': $([S \cup \{\#\}]^{n+2})^* \to T^*$ be the length-preserving homomorphism determined by defining $h'([b_1, \cdots, b_{n+2}]) = [b_1, \cdots, b_{n+1}]$. Then $h(\langle x_1, \cdots, x_n, y, z\rangle) = \langle x_1, \cdots, x_n, z\rangle$ and, when $|z| \leqq |y|$, $h'(\langle x_1, \cdots, x_n, y, z\rangle) = \langle x_1, \cdots, x_n, y\rangle$; so that $\theta(Q) = h'(L_0 \cap h^{-1}(\theta(R)))$.

For 3), suppose $Q \subseteq [S^*]^m$ is an explicit transformation of $R \subseteq [S^*]^n$. Then there are terms $t_1, \cdots, t_n$ formed by concatenation from S^* and $x_1, \cdots, x_m$ such that $(x_1, \cdots, x_m) \in Q$ if and only if $(t_1, \cdots, t_n) \in R$. From the form of the terms, for each i, $1 \leqq i \leqq n$, there are constants $c_0(i), \cdots, c_m(i)$ such that $|t_i| \leqq c_0(i) + c_1(i)|x_1| + \cdots + c_m(i)|x_m|$. Let $k = \max\{c_0(i) + \cdots + c_m(i) : 1 \leqq i \leqq n\}$. For each string w in S^* with $1 \leqq |w| \leqq k$, let $[w]$ be a new symbol; let U be the set of these symbols and let $V \subseteq U$ be the set $\{[w] : w \in S^*, |w| = k\}$. Let $g: (U \cup \{\#\})^* \to S^*$ be the homomorphism determined by defining $g(\#) = e$ and $g([w]) = w$. Let $L_1 = \{\langle x_1, \cdots, x_m, z_1, \cdots, z_n\rangle$: for $1 \leqq i \leqq n$, $|z_i| \leqq |\langle x_1, \cdots, x_m\rangle|$, $g(z_i) = t_i(x_1, \cdots, x_m)$ and either $z_i = e$ or $z_i \in V^*U\}$. Since each term is a concatenation of some of the x_i's and strings in S^*, L_1 can be accepted in linear time by a deterministic Turing machine, so $L_1 \in$ DTIME(lin). Let $T = [S \cup \{\#\}]^m \times [U \cup \{\#\}]^n$. Let $h_1: T^* \to ([S \cup \{\#\}]^m)^*$ and $h_2: T^* \to ([S \cup \{\#\}]^n)^*$ be the homomorphisms determined by defining for $a_1, \cdots, a_m \in S \cup \{\#\}$ and $b_1, \cdots, b_n \in U \cup \{\#\}$, $h_1([a_1, \cdots, a_m, b_1, \cdots, b_n]) = [a_1, \cdots, a_m]$ and $h_2([a_1, \cdots, a_m, b_1, \cdots, b_n]) = \langle g(b_1), \cdots, g(b_n)\rangle$. Notice that h_1 is length-preserving and, when $y = \langle x_1, \cdots, x_m, z_1, \cdots, z_n\rangle$ is in L_1, $h_1(y) = \langle x_1, \cdots, x_m\rangle$ and $h_2(y) = \langle g(z_1), \cdots, g(z_n)\rangle$. Let $L_2 = \{e\} \cap \theta(Q)$, so that L_2 is either $\{e\}$ or $\varnothing$ and is therefore a regular set. Then $\theta(Q) = h_1(L_1 \cap h_2^{-1}(\theta(R))) \cup L_2$. □

2.3. We now turn to an informal definition of the model for relative computation that will be used to characterize RUD. These "oracle machines" differ from the "query machines" of [1], [7] only in that the oracle tape is erased after an oracle call; the definition is essentially that used in [16].

DEFINITION. An *oracle machine* is a multitape Turing acceptor with an added dynamic capability. A computation of an oracle machine M depends on both an input string x and an *oracle set* A, which may be any language over the tape alphabet of M. The machine M has three distinguished states $q_?$, *Yes* and *No*, along with its initial and final states, and one of its work tapes is distinguished as the oracle tape. At any point during a computation of M on x relative to A, there are two possibilities:

(i) The current state of M is not its query state $q_?$ (although it might be one of the response states *Yes*, *No*). In this case, the next step of the computation is determined by the transition function of M, as for an ordinary Turing acceptor.

During such steps, M can read from and write on its oracle tape, as well as its other tapes.

(ii) M has entered its query state $q_?$, in order to make an oracle call. In this case, the next step is determined by the string on the oracle tape and the oracle set: if the (nonblank) contents of the oracle tape is the string z, then the next state is *Yes* if $z \in A$ and *No* if $z \notin A$. During a step that is an oracle call, the oracle tape is erased (i.e., reset to blanks) but the configuration of the other work tapes and the input tape is unchanged.

The oracle machine M is deterministic if its transition function allows at most one move at any step, nondeterministic otherwise. The transition function is undefined for the query state, so moves from that state are uniquely determined by A. M is said to *accept x relative to A* if and only if some computation of M on x relative to A reaches an accepting state. Let $M(A)$ denote the set of strings accepted by M relative to A.

DEFINITION. Suppose $t: N \to N$ is a nondecreasing function which satisfies $t(n) \geqq n$. An oracle machine M is said to *operate in time $t(n)$* if, for any input x and any oracle set A, every computation of M on x relative to A halts in at most $t(|x|)$ steps. (An oracle call costs one step.) Notice that the property of operating in time $t(n)$ is independent of the oracle set.

Time-bounded oracle machines share many of the properties of other resource-bounded abstract automata and proofs about Turing machines can often be easily extended to apply to them (e.g., for closure properties, for separation of classes based on increasing time bounds). Attention here is restricted to linear time bounds.

DEFINITION. 1) An oracle machine is termed a *linear-time* oracle machine if it operates in time $cn + d$ for some constants c, d.

2) If $\mathscr{L}$ is a class of languages, then $\mathrm{NL}(\mathscr{L}) = \{M(A) : A \in \mathscr{L}, M \text{ a nondeterministic linear-time oracle machine}\}$.

It is not hard to see that $\mathrm{NL}(\{\varnothing\}) = \mathrm{NTIME}(n)$; this will be discussed in Proposition 3.1.

We view $\mathrm{NL}(\cdot)$ as an operator that extends a class $\mathscr{L}$ by adding to it the languages that can be accepted in linear time relative to some language in $\mathscr{L}$. The following definition provides the notation for iterated application of $\mathrm{NL}(\cdot)$.

DEFINITION. Let $\mathscr{L}$ be a class of languages. Define $\mathrm{NL}^0(\mathscr{L}) = \mathscr{L}$, and for $k \geqq 0$, $\mathrm{NL}^{k+1}(\mathscr{L}) = \mathrm{NL}(\mathrm{NL}^k(\mathscr{L}))$. Define $\mathrm{NL}^*(\mathscr{L}) = \bigcup \{\mathrm{NL}^k(\mathscr{L}) : k \geqq 0\}$. That is, $\mathrm{NL}^*(\mathscr{L})$ is the smallest class of languages $\mathscr{C}$ satisfying $\mathscr{L} \subseteq \mathscr{C}$ and $\mathrm{NL}(\mathscr{C}) \subseteq \mathscr{C}$.

If $\mathscr{L}$ is a nonempty class, then $\varnothing \in \mathrm{NL}(\mathscr{L})$, so if $\mathrm{NL}(\mathscr{L}) \subseteq \mathscr{L}$ and $\mathscr{L}$ is nonempty, then $\mathrm{NL}^*(\{\varnothing\}) \subseteq \mathscr{L}$. Therefore $\mathrm{NL}^*(\{\varnothing\})$ is the smallest nonempty class that is closed under $\mathrm{NL}(\cdot)$.

3. Characterizations of the rudimentary languages. Two characterizations of RUD are proved in this section. The first characterization states that $\mathrm{RUD} = \mathrm{NL}^*(\{\varnothing\})$; that is, RUD is the smallest nonempty class that is "closed under" $\mathrm{NL}(\cdot)$. In order to prove that $\mathrm{RUD} \subseteq \mathrm{NL}^*(\{\varnothing\})$, it is sufficient to show that languages encoding concatenation are in $\mathrm{NL}^*(\{\varnothing\})$ and that $\mathrm{NL}^*(\{\varnothing\})$ is closed under the operations used (in Proposition 2.2) to express the effect of the rudimentary operations. For the reverse containment, an induction proof is used to show that for all k, $\sigma_k \subseteq \mathrm{RUD}$ where $\sigma_k = \mathrm{NL}^k(\{\varnothing\})$. The induction step is based on a representation of the languages accepted by time-bounded oracle machines that yields a simple representation of σ_{k+1} in terms of complementation and length-preserving homomorphism applied to languages in σ_k. The second characterization states that RUD is the smallest class containing the regular sets and the language $\{0^n 1^n : n \geqq 0\}$ and closed under the Boolean operations, inverse homomorphism and length-

preserving homomorphism. Notice that both of the classes proven equal to RUD have inductive definitions, as the smallest class containing certain basis sets and closed under certain operations. Since the basis sets in each case are simple, we may take the fact that these classes are equal to RUD as providing information about the expressive power of the operations.

We begin with a definition of classes of languages that decompose NL*($\{\varnothing\}$).

DEFINITION. Define $\sigma_0 = \{\varnothing\}$. For $k \geqq 0$, define $\sigma_{k+1} = \mathrm{NL}(\sigma_k)$. That is, σ_{k+1} consists of the languages accepted by nondeterministic linear-time oracle machines relative to languages in σ_k.

It is clear from this definition that $\sigma_k = \mathrm{NL}^k(\{\varnothing\})$ for $k \geqq 0$; therefore $\bigcup_k \sigma_k = \mathrm{NL}^*(\{\varnothing\})$. The collection of classes $\{\sigma_0, \sigma_1, \sigma_2, \cdots\}$ will be referred to as the "linear hierarchy". The linear hierarchy bears the same relationship to the class NTIME(n) as the "polynomial-time hierarchy" [24] does to $\bigcup_k \mathrm{NTIME}(n^k)$. The classes σ_k for $k \geqq 1$ remain the same if σ_0 is taken to be DTIME(lin) rather than $\{\varnothing\}$.

Except for the trivial case $k = 0$, it is not known whether $\sigma_k \subsetneq \sigma_{k+1}$. A proof of either proper containment or equality must rely on properties specific to the classes rather than only on general properties of the operator $\mathrm{NL}(\cdot)$ since classes of recursive languages $\mathscr{C}_1$ and $\mathscr{C}_2$ can be found such that $\mathscr{C}_1 = \mathrm{NL}(\mathscr{C}_1)$ but $\mathscr{C}_2 \subsetneq \mathrm{NL}(\mathscr{C}_2)$. (The class $\mathscr{C}_2$ can be constructed by employing techniques used in [1] for polynomial time.)

The following proposition contains some basic properties of the classes in the linear hierarchy.

PROPOSITION 3.1.

1) $\sigma_1 = \mathrm{NTIME}(n)$.

2) *For all* $k \geqq 0$, $\sigma_k \cup \text{co-}\sigma_k \subseteq \sigma_{k+1}$.

3) *For each* $k \geqq 1$, σ_k *is closed under union, intersection, product, Kleene* *, *inverse homomorphism and linear erasing homomorphism.*

Notice that closure under complementation for σ_k is not asserted in 3); this closure property would imply that the linear hierarchy collapses at the kth level, i.e., that $\sigma_k = \bigcup_j \sigma_j$ (see Proposition 4.1).

Proof. 1) A Turing acceptor may be viewed as an oracle machine that never consults its oracle, so that $\mathrm{NTIME}(n) \subseteq \mathrm{NL}(\{A\})$ for any language A. In particular, $\mathrm{NTIME}(n) \subseteq \mathrm{NL}(\{\varnothing\}) = \sigma_1$. On the other hand, from an oracle machine M_1, a Turing machine M_2 can be easily constructed that will assume a response of "no" to any oracle call of M_1 and otherwise act like M_1, so that $L(M_2) = M_1(\varnothing)$. The Turing machine M_2 will also operate in the same time bound as M_1. Therefore $\sigma_1 = \mathrm{NL}(\{\varnothing\}) \subseteq \mathrm{NTIME}(n)$.

2) For any alphabet S, deterministic oracle machines D_1 and D_2 can be constructed that both operate in time $n+1$ and are such that for any $L \subseteq S^*$, $D_1(L) = D_2(S^* - L) = L$. If $L \in \sigma_k$ then $L = D_1(L) \in \sigma_{k+1}$, and if $L \in \text{co-}\sigma_k$ then $S^* - L \in \sigma_k$ so that $L = D_2(S^* - L) \in \sigma_{k+1}$.

3) Closure under Kleene *, inverse homomorphism and linear erasing homomorphism follows from simple extensions of the usual Turing machine constructions for these operations. The operations of union, intersection and product differ from the other three in that (possibly) two oracle sets are involved. If, however, M_1 and M_2 are nondeterministic linear-time oracle machines with a common oracle set C, then it is again easy to extend the usual constructions to yield, for example, a nondeterministic linear-time oracle machine M_3 such that $M_3(C) = M_1(C) \cap M_2(C)$. Also, if A is an oracle set for M_1 and B is some other language, then a nondeterministic linear-time oracle machine M_1' can be constructed such that $M_1(A) = M_1'(\mathrm{¢}A \cup \$B)$ where ¢, \$ are two new symbols: M_1' acts like M_1 except for marking

the strings on its oracle tape with ¢. Combining these ideas, we can conclude that if $\mathscr{C}$ is either a singleton class or a class that is closed under product (with regular sets) and union, then NL($\mathscr{C}$) is closed under union, intersection and product. Since $\sigma_0 = \{\varnothing\}$ is a singleton, an induction proof will show that each σ_k is closed under union, intersection and product. □

The properties given in Proposition 3.1 can be restated as properties of the union of the linear hierarchy. The following corollary also holds for NL*($\{A\}$) where A is any language and for NL*($\mathscr{C}$), if $\mathscr{C}$ is closed under union and product.

COROLLARY 3.2. NL*($\{\varnothing\}$) *is closed under the Boolean operations, product, Kleene *, inverse homomorphism and linear erasing homomorphism, and contains* NTIME(n).

The next corollary is of more immediate interest for the characterization of RUD. It may be proved by combining Proposition 3.1 with the relationship between rudimentary operations and language operations as given in Proposition 2.2.

COROLLARY 3.3. *The class of string relations* $\{R: \theta(R) \in \text{NL}^*(\{\varnothing\})\}$ *is closed under the rudimentary operations. Suppose* R, R' *are relations such that* $\theta(R)$, $\theta(R') \in \sigma_k$. *Then*

1) $\theta(R \cup R') \in \sigma_k$, $\theta(R \cap R') \in \sigma_k$ *and* $\theta(R - R') \in \sigma_{k+1}$;

2) *if* Q *is defined from* R *by bounded existential quantification, then* $\theta(Q) \in \sigma_k$; *and*

3) *if* Q *is an explicit transformation of* R, *then* $\theta(Q) \in \sigma_k$.

The closure properties in Proposition 3.1 have simple proofs because the classes in the linear hierarchy were defined using automata. However, for an induction proof that $\sigma_k \subseteq \text{RUD}$ for each k, it is useful to have a definition of σ_{k+1} from σ_k using operations on languages. To achieve this, we first establish a representation of languages accepted by time-bounded oracle machines.

THEOREM 3.4. 1) *Suppose* M *is a nondeterministic linear-time oracle machine and* A *is an oracle set for* M. *Then there exist a deterministic linear-time oracle machine* D *and a length-preserving homomorphism* h *such that* $M(A) = h(D(A))$.

2) *Let* D *be a deterministic linear-time oracle machine with tape alphabet* S *and let* $A \subseteq S^*$ *be an oracle set for* D. *Then there exist homomorphisms* h_1 *and* h_2, *with* h_1 *length-preserving, and a language* $L \in \text{DTIME(lin)}$ *such that* $D(A) = h_1(L \cap h_2^{-1}(L'))$ *where* $L' = (\#_1 A \cup \#_2(S^* - A))^*$ *with* $\#_1$, $\#_2 \notin S$.

Proof. Some notation will be useful in the proof. If Γ is an alphabet and k is an integer, $k \geqq 1$, let $\Gamma_k = \{[w] : w \in \Gamma^*,\ 1 \leqq |w| \leqq k\}$. That is, for each nonempty string $w \in \Gamma^*$ of length at most k, $[w]$ is a new symbol, and Γ_k is the set of these symbols. For $x \in \Gamma^*$, $x/k \in (\Gamma_k)^*$ is defined as follows. Suppose $|x| = mk + j$, $m \geqq 0$, $0 \leqq j \leqq k-1$. If $j = 0$, then $x/k = [w_1][w_2] \cdots [w_m]$ where $x = w_1 \cdots w_m$ and $|w_i| = k$ for $1 \leqq i \leqq m$; if $j \geqq 1$, then $x/k = [w_1] \cdots [w_m][y]$ where $x = w_1 \cdots w_m y$, $|w_i| = k$ for $1 \leqq i \leqq m$ and $|y| = j$. If $L \subseteq \Gamma^*$, let $L/k = \{x/k : x \in L\}$. Note that Γ^*/k is a regular set.

1) Part 1) is proved by applying to oracle machines a technique used in [5]; the deterministic machine is supplied with both an input string and a "choice" string that describes the transitions made by the nondeterministic machine in an accepting computation on the given input.

Suppose a nondeterministic oracle machine M operates in time $cn + d$ and has input alphabet T. Let $k = c + d$. Suppose M has at most m choices of transition at any step and let $V = \{v_0, v_1, \cdots, v_m\}$ be an alphabet of $m+1$ distinct symbols. ("v_0" represents a call on the oracle, the only move possible from the query state.) Let $\Sigma = T \times (V_k \cup \{\#\}) = \{[b, \#] : b \in T\} \cup \{[b, [w]] : b \in T,\ w \in V^*,\ 1 \leqq |w| \leqq k\}$ where $\# \notin V_k$. Let A be an oracle set for M.

The deterministic oracle machine D will have input alphabet Σ and operate as follows. Given the empty string as input, D follows all the computations of M on e, using its oracle just as M would, and accepts e if and only if M does. Since any computation of M on e (relative to A) can have at most d steps, there are only finitely many computations for D to check. For nonempty input strings, D accepts only strings of the form $\langle x, u/k\rangle \in \Sigma^*$ with $x \in T^+$, $u \in V^+$ and $|u/k| \leqq |x|$. Given such an input, D follows (if possible) the computation of M on x as described by u, again using its tapes as M would. The details of the simulation are straightforward. D can be constructed to operate in linear time; since M operates in time $cn+d$ and $cn+d \leqq kn$ for $n \geqq 1$, for any $x \neq e$, $x \in M(A)$ if and only if there is an accepting computation of M on x relative to A with at most $k|x|$ steps if and only if there is some $u \in V^+$ such that $\langle x, u/k\rangle \in D(A)$. Let $h: \Sigma^* \to T^*$ be the length-preserving homomorphism determined by defining $h([b_1, b_2]) = b_1$ for $b_1 \in T$, $b_2 \in V_k \cup \{\#\}$; then $M(A) = h(D(A))$.

2) Suppose D is a deterministic oracle machine that operates in time $cn+d$ and has input alphabet T and tape alphabet S. Let $\#_1, \#_2 \notin S$ be two new symbols and $U = S \cup \{\#_1, \#_2\}$. Let $\Sigma = T \times (U_k \cup \{\#\})$, where $k = c+d$.

Let $R \subseteq \Sigma^*$ be the regular set $R = \{\langle x, y\rangle : x \in T^+, y \in U^*/k, |x| \geqq |y|\}$. Let $h_2: \Sigma^* \to U^*$ be the homomorphism determined by defining $h_2([b, \#]) = e$ and for $w \in U^*$, $1 \leqq |w| \leqq k$, $b \in T$, $h_2([b, [w]]) = w$. Then if $A \subseteq S^*$ and $L' = (\#_1 A \cup \#_2(S^* - A))^*$, we have $h_2^{-1}(L') \cap R = \{\langle x, u/k\rangle : x \in T^+, u \in L', |x| \geqq |u/k|\}$.

Let D' be the following deterministic Turing acceptor, with input alphabet Σ. D' rejects its input unless it is of the form $\langle x, u/k\rangle$ with $x \in T^+$, $u \in U^*$ and $|u/k| \leqq |x|$. On an input of this form, D' acts as D would on input x, using the information in $u = \#_{i_1} u_1 \cdots \#_{i_m} u_m$ instead of oracle calls; that is, D' checks that D would query its oracle about $u_1, \cdots, u_m$ (in that order) and D' continues from the "yes" state if $i_j = 1$ and from the "no" state if $i_j = 2$, $1 \leqq j \leqq m$. D' accepts $\langle x, u/k\rangle$ if and only if the answers in u lead D to accept x. Now since D operates in time $cn+d$ and the oracle tape is erased after an oracle call, if during a computation on x, D queries its oracle about strings $u_1, \cdots, u_m$, $m \geqq 0$, and receives "answers" $i_1, \cdots, i_m$, then $m + |u_1| + \cdots + |u_m| \leqq c|x| + d \leqq k|x|$ for $x \neq e$; hence if $u = \#_{i_1} i_1 \cdots \#_{i_m} u_m$ then $|u/k| \leqq |x|$. Further, for any oracle set A, the answers in u are correct relative to A if and only if $u \in (\#_1 A \cup \#_2(S^* - A))^* = L'$. Therefore, for any $x \in T^+$, $x \in D(A)$ if and only if there exists $u \in U^*$ such that $\langle x, u/k\rangle \in L(D')$ and $u \in L'$; or $x \in D(A)$ if and only if there exists $y \in U_k^*$ with $|y| \leqq |x|$ such that $\langle x, y\rangle \in L(D') \cap (h_2^{-1}(L') \cap R)$.

Let $h_1: \Sigma^* \to T^*$ be the length-preserving homomorphism determined by defining $h_1([b_1, b_2]) = b_1$. Let $L = (L(D') \cap R) \cup (D(A) \cap \{e\})$; since R and $(D(A) \cap \{e\})$ are regular sets and D' can be constructed to operate in linear time, $L \in \mathrm{DTIME(lin)}$. Then $D(A) = h_1(L \cap h_2^{-1}(L'))$. □

The constructions in the proof of Theorem 3.4 can be made uniform in the oracle set A by allowing the homomorphism h_1 to perform some erasing (or by ignoring the empty string). The uniform constructions can also be applied to oracle machines which do not necessarily operate in linear time, yielding the following generalization of the representation to arbitrary time bounds.

PROPOSITION 3.5. *Suppose M is a nondeterministic oracle machine which operates in time $t(n)$ and has tape alphabet S. Then there exist homomorphisms h_1 and h_2 and a language $L_M \in \mathrm{DTIME(lin)}$ such that for any oracle set $A \subseteq S^*$, $M(A) = h_1(L_M \cap h_2^{-1}((\#_1 A \cup \#_2(S^* - A))^*))$. Further, the homomorphism h_1 has the property that for any $w \in L_M$, $|w| \leqq t(|h_1(w)|)$.*

Proof (sketch). The language L_M is given by: $L_M = \{\langle x, y, z\rangle$: the transitions described in y and the information given in z about the oracle set cause M to accept $x\}$. The homomorphisms h_1 and h_2, then, satisfy $h_1(\langle x, y, z\rangle) = x$ and $h_2(\langle x, y, z\rangle) = z$.

Since M operates in time $t(n)$, if $\langle x, y, z\rangle \in L_M$ then $|z| \leqq |y| \leqq t(|x|)$ so $|\langle x, y, z\rangle| = |y| \leqq t(|x|) = t(|h_1(\langle x, y, z\rangle)|)$. □

Theorem 3.4 is the basis for the following simple characterization of the languages in σ_k $(k \geqq 2)$.

COROLLARY 3.6. *For $k \geqq 2$, a language $L_1 \subseteq \Sigma^*$ is in σ_k if and only if $L_1 = h(\Delta^* - L_2)$ where $L_2 \in \sigma_{k-1}$ and $h: \Delta^* \to \Sigma^*$ is a length-preserving homomorphism.*

Proof. For $k \geqq 2$, let $\mathscr{H}_k = \{h(\Delta^* - L) : L \subseteq \Delta^*$ in σ_{k-1}, h a length-preserving homomorphism$\}$. Since from Proposition 3.1, co-$\sigma_{k-1} \subseteq \sigma_k$ and σ_k is closed under length-preserving homomorphism, for each $k \geqq 2$, $\mathscr{H}_k \subseteq \sigma_k$.

For the reverse containment, suppose $L_1 \in \sigma_k$ for some $k \geqq 2$. Combining the representations given in Theorem 3.4, $L_1 = h_1(L \cap h_2^{-1}(L'))$ where h_1 is a length-preserving homomorphism, h_2 is a homomorphism, $L \in$ DTIME(lin) and $L' = (\#_1 A \cup \#_2(S^* - A))^*$ for some language $A \subseteq S^*$ in σ_{k-1}. To show that $L_1 \in \mathscr{H}_k$, it is therefore sufficient to prove: (i) $\sigma_{k-1} \cup$ co-$\sigma_{k-1} \subseteq \mathscr{H}_k$, (ii) DTIME(lin) $\subseteq \mathscr{H}_k$ and (iii) $\mathscr{H}_k$ is closed under union, intersection, product, Kleene *, inverse homomorphism and length-preserving homomorphism.

(i) It is apparent from the definition that co-$\sigma_{k-1} \subseteq \mathscr{H}_k$. To see that $\sigma_{k-1} \subseteq \mathscr{H}_k$, suppose $A \in \sigma_{k-1}$; then since $k \geqq 2$, $A = M(B)$ for some language $B \in \sigma_{k-2}$ and some nondeterministic linear-time oracle machine M. From Theorem 3.4 part 1), there is a length-preserving homomorphism h and a deterministic linear-time oracle machine D such that $A = h(D(B))$. Suppose $D(B) \subseteq T^*$; then a straightforward construction will yield a deterministic linear-time oracle machine $\bar{D}$ such that $\bar{D}(B) = T^* - D(B)$. Since $B \in \sigma_{k-2}$ and $\bar{D}$ is a linear-time oracle machine, $\bar{D}(B) \in \sigma_{k-1}$, so $A = h(T^* - \bar{D}(B)) \in \mathscr{H}_k$.

(ii) Since DTIME(lin) $\subseteq$ NTIME$(n) = \sigma_1$ and $\sigma_1 \subseteq \sigma_{k-1} \subseteq \mathscr{H}_k$ for $k \geqq 2$, DTIME(lin) $\subseteq \mathscr{H}_k$.

(iii) The following fact about classes of languages that possess certain closure properties is proved in [9] (see Theorems 1.2 and 2.2).

Claim. If $\mathscr{C}$ is a class of languages containing the regular sets and closed under union, intersection, product, Kleene *, inverse homomorphism and nonerasing homomorphism, then the class $\{h(L) : L \in$ co-$\mathscr{C}$, h a length-preserving homomorphism$\}$ is closed under the same six operations.

The properties of the classes σ_k given in Proposition 3.1 ensure that this claim applies to them, so $\mathscr{H}_k$ is closed under the required operations for each $k \geqq 2$. □

We now consider some properties of the rudimentary relations.

PROPOSITION 3.7 [18]. *If R is a string relation such that $\theta(R) \in$ NSPACE(log n), then R is rudimentary.*

Proof. If $R \subseteq [S^*]^m$ and $\# \notin S$, let $\sigma(R) = \{x_1 \# x_2 \# \cdots x_m \# : (x_1, \cdots, x_m) \in R\}$. It is shown in [18] that R is rudimentary if there are constants k $(k \geqq 1)$ and ε $(0 < \varepsilon < 1)$ such that $\sigma(R)$ is accepted in time n^k and space n^ε by a nondeterministic Turing machine with two-way read-only input and one work tape (to which the space bound applies). In the proof, rudimentary predicates are constructed which represent sequences of steps of the Turing machine on a given input.

It is easy to see that $\sigma(R) \in$ NSPACE(log n) if and only if $\theta(R) \in$ NSPACE(log n) and that any language in NSPACE(log n) can be accepted by a device which satisfies the conditions of the theorem cited above; therefore, if $\theta(R) \in$ NSPACE(log n), then R is rudimentary. □

COROLLARY 3.8. RUD *is closed under nonerasing homomorphism and inverse homomorphism.*

Proof. Suppose $h: S^* \to T^*$ is a nonerasing homomorphism and $L \subseteq S^*$ is in RUD. Let $R_1 = \{(x, y) : h(y) = x,\ x \in T^*, y \in S^*\}$ and $R_2 = \{(x, y) : y \in L,\ x \in T^*\}$. Then

$\theta(R_1) \in \mathrm{DSPACE}(\log n)$ and R_2 is an explicit transformation of L, so $R_1 \cap R_2$ is rudimentary. Since h is nonerasing, for $y \in S^*$, $|y| \leqq |h(y)|$ so $h(L) = \{x \in T^*$: for some $y \in S^*$ with $|y| \leqq |x|$, $(x, y) \in R_1 \cap R_2\}$ is rudimentary.

If $g: S^* \to T^*$ is an arbitrary homomorphism and $L \subseteq T^*$ is in RUD, let $Q_1 = \{(x, y) : y = g(x), y \in T^*, x \in S^*\}$ and $Q_2 = \{(x, y) : x \in S^*, y \in L\}$. As before, $Q_1 \cap Q_2$ is rudimentary and therefore $Q_3 = \{(x, z)$: for some $y \in T^*$ with $|y| \leqq |x|$, $(x, y) \in Q_1 \cap Q_2\}$ is rudimentary. Let $m = \max\{|g(a)| : a \in S\}$; then $|g(x)| \leqq m|x|$ for any $x \in S^*$ and so $g^{-1}(L) = \{x : (x, x^m) \in Q_3\}$ and $g^{-1}(L) \in \mathrm{RUD}$. □

COROLLARY 3.9. 1) *The class of context-free languages is contained in* RUD.

2) $\mathrm{NTIME}(n) \subseteq \mathrm{RUD}$.

Proof. Part 1) is also given in [15], [29]. A simple proof uses Proposition 3.7 and Corollary 3.8. If L is any context-free language, then there exist a length-preserving homomorphism h_1, a homomorphism h_2 and a regular set L' such that $L = h_1(L' \cap h_2^{-1}(D_2))$ where D_2 is the Dyck set on two letters. (A proof of this form of the Chomsky–Schützenberger theorem may be found in [3].) Any regular set is in $\mathrm{DSPACE}(\log n)$ and hence is rudimentary; also $D_2 \in \mathrm{DSPACE}(\log n)$ [20], so D_2 is rudimentary. Since RUD is closed under intersection, length-preserving homomorphism and inverse homomorphism, any context-free language must therefore be in RUD.

Part 2) follows from 1) since (with the use of [4]) for any language $L \in \mathrm{NTIME}(n)$ there exist a length-preserving homomorphism h and (deterministic) context-free languages L_1, L_2, L_3 such that $L = h(L_1 \cap L_2 \cap L_3)$. □

Since the class of context-free languages is not closed under intersection, proper containment holds in part 1) of this corollary. Whether the containment $\mathrm{NTIME}(n) \subseteq \mathrm{RUD}$ is proper is not known; it is proper if and only if $\mathrm{NTIME}(n)$ is not closed under complementation.

All the preliminary results necessary for the characterizations of RUD have now been established.

THEOREM 3.10. $\mathrm{RUD} = \mathrm{NL}^*(\{\varnothing\})$. *That is,* RUD *is the smallest nonempty class of languages* $\mathscr{C}$ *that satisfies* $\mathrm{NL}(\mathscr{C}) \subseteq \mathscr{C}$.

Proof. To see that $\mathrm{RUD} \subseteq \mathrm{NL}^*(\{\varnothing\})$, first notice that for any concatenation relation C, $\theta(C)$ can be accepted by a (deterministic) Turing machine that operates in linear time, so $\theta(C) \in \mathrm{NTIME}(n) \subseteq \mathrm{NL}^*(\{\varnothing\})$. From the definition of RUD and the properties of $\mathrm{NL}^*(\{\varnothing\})$ given in Corollary 3.3, we can conclude that $\mathrm{RUD} \subseteq \mathrm{NL}^*(\{\varnothing\})$.

For the reverse containment, from Corollary 3.9, $\sigma_1 = \mathrm{NTIME}(n) \subseteq \mathrm{RUD}$. Continuing by induction, suppose $\sigma_k \subseteq \mathrm{RUD}$. From Corollary 3.6, $\sigma_{k+1} = \{h(L) : L \in \text{co-}\sigma_k$, h a length-preserving homomorphism$\}$ and RUD is closed under complementation and (from Corollary 3.8) length-preserving homomorphism. Therefore σ_{k+1} is also contained in RUD. □

While it is known [17] that $\mathrm{RUD} \subseteq \mathrm{DSPACE}(n)$, whether this containment is proper remains open. Theorem 3.10 gives rise to the following conditions for the containment to be proper.

THEOREM 3.11. 1) $\mathrm{RUD} = \mathrm{DSPACE}(n)$ *if and only if there is some* $k \geqq 1$ *such that* $\sigma_k = \mathrm{DSPACE}(n)$.

2) *If the containments* $\sigma_1 \subseteq \sigma_2 \subseteq \cdots$ *are all proper, then the rudimentary relations are properly contained in* $\mathscr{E}_*^2$.

Proof. Part 2) follows easily from 1) since $\mathrm{DSPACE}(n)$ consists of the languages that encode the $\mathscr{E}_*^2$ relations [19].

The "if" direction of 1) is obvious, since each σ_k is contained in RUD. For the other direction we use the fact that the class $\mathrm{DSPACE}(n)$ is "principal": there is a

language $L_1 \in \text{DSPACE}(n)$ with the property that for any $L \in \text{DSPACE}(n)$ there is a homomorphism h such that $L - \{e\} = h^{-1}(L_1)$ [25]. If $\text{RUD} = \text{DSPACE}(n)$, then $L_1 \in \text{RUD} = \bigcup_j \sigma_j$ so there is some $k \geqq 1$ such that $L_1 \in \sigma_k$. Then since σ_k is closed under inverse homomorphism and union with $\{e\}$, $\text{DSPACE}(n) \subseteq \sigma_k$, so $\text{DSPACE}(n) = \sigma_k$. □

Recall that the question of whether $\mathscr{E}_*^0 \subsetneq \mathscr{E}_*^2$ remains open. Since $\text{RUD} \subseteq \mathscr{E}_*^0 \subseteq \mathscr{E}_*^2 = \text{DSPACE}(n)$, the equality $\text{RUD} = \text{DSPACE}(n)$ would imply $\mathscr{E}_*^0 = \mathscr{E}_*^2$.

The parallel between the arithmetical sets and RUD suggested by the use of a restricted Turing reducibility in Theorem 3.10 extends to another characterization of the two classes. Based on [12], the arithmetical sets may be viewed as the smallest Boolean-closed full AFL containing the language $\{0^n 1^n : n \geqq 0\}$, where a "full AFL" [8] is a class of languages closed under union, intersection with regular sets, product, Kleene *, inverse homomorphism and (unrestricted) homomorphism. Restricting the homomorphisms used to those that are length-preserving yields the following characterization of RUD.

THEOREM 3.12. RUD *is the smallest class of languages containing the language* $L_0 = \{0^n 1^n : n \geqq 0\}$ *and the regular sets and closed under the Boolean operations, inverse homomorphism and length-preserving homomorphism.*

Proof. Let $\mathscr{L}_0$ denote the class of languages described in the statement of the theorem. From Corollary 3.9, the basis sets for $\mathscr{L}_0$ are rudimentary, since L_0 is a context-free language and all regular sets are context-free. From the definition and Corollary 3.8, RUD is closed under the Boolean operations, inverse homomorphism and length-preserving homomorphism. Therefore $\mathscr{L}_0 \subseteq \text{RUD}$.

It is proved in [27] that any context-free language can be defined from L_0 and regular sets by use of the Boolean operations, inverse homomorphism and length-preserving homomorphism, so any context-free language is in $\mathscr{L}_0$. If L is a language in $\text{DTIME(lin)} \subseteq \text{NTIME}(n)$, then there exist a length-preserving homomorphism h and context-free languages L_1, L_2, L_3 such that $L = h(L_1 \cap L_2 \cap L_3)$ [4], so $L \in \mathscr{L}_0$ and therefore $\text{DTIME(lin)} \subseteq \mathscr{L}_0$. Now the encoded concatenation relation $\theta(C_S)$ is in DTIME(lin) for any alphabet S and, since $\mathscr{L}_0$ is closed under the Boolean operations, inverse homomorphism and length-preserving homomorphism and contains DTIME(lin), from Proposition 2.2 the class of relations $\{R : \theta(R) \in \mathscr{L}_0\}$ is closed under the rudimentary operations. This class of relations therefore contains the rudimentary relations and so $\text{RUD} \subseteq \mathscr{L}_0$. □

A version of this characterization with the context-free languages as the basis sets is proved in [29]. Either nonerasing or linear-erasing homomorphism may also be used in Theorem 3.12.

4. The linear hierarchy. In this section, some further properties of the linear hierarchy are considered.

We first establish conditions for the linear hierarchy to be finite. By Corollary 3.11, if it is infinite then RUD is properly contained in $\text{DSPACE}(n)$.

PROPOSITION 4.1. *For all* $k \geqq 1$, *the following are equivalent*:

(i) $\sigma_k = \bigcup_j \sigma_j$.

(ii) co-σ_k *is closed under nonerasing homomorphism.*

(iii) σ_k *is closed under complementation.*

Proof. If (i) is true, then $\sigma_k = \text{RUD}$ so also co-$\sigma_k = \text{RUD}$ and, since RUD is closed under nonerasing homomorphism, (ii) is true. Recall from Corollary 3.6 that $\sigma_{k+1} = \{h(L) : L \in \text{co-}\sigma_k, h \text{ a length-preserving homomorphism}\}$; therefore if (ii) is true then $\sigma_{k+1} \subseteq \text{co-}\sigma_k$ and so $\sigma_k \subseteq \text{co-}\sigma_k$. This containment implies that $\text{co-}\sigma_k \subseteq \sigma_k$, i.e., that σ_k

is closed under complementation, since σ_k is closed under union with and intersection with regular sets. (Note that if $L_1 = \Sigma^* - L_2$ and $L_2 = \Delta^* - L_3$, then $L_1 = (\Sigma^* - \Delta^*) \cup (\Sigma^* \cap L_3)$.) Finally, if (iii) is true then σ_k is closed under both complementation and length-preserving homomorphism, so (again with the use of Corollary 3.6) $\sigma_{k+1} \subseteq \sigma_k$. From this, an induction argument will show that, for all j, $\sigma_j \subseteq \sigma_k$ and (i) is true. □

The next theorem gives a characterization of the classes in the linear hierarchy based on the number of alternations of (suitably bounded) quantifiers. The class RUD is easily seen to be closed under both the following forms of quantification; they are introduced in order to state the characterization more easily.

Notation. Let P be a string predicate. The abbreviation $(\exists y)_x P(x, y)$ is used for the expression $(\exists y)[|y| \leqq |x| \text{ and } P(x, y)]$. Dually, $(\forall y)_x P(x, y)$ abbreviates $(\forall y)$ [*if* $|y| \leqq |x|$ *then* $P(x, y)$]. In both cases, the quantifiers are assumed to range over strings on the appropriate alphabet.

Notice that for $L \subseteq S^*$, $L' \subseteq T^*$, if $x \in L \Leftrightarrow (\exists y)_x[\langle x, y\rangle \in L']$ then $x \in S^* - L \Leftrightarrow (\forall y)_x[\langle x, y\rangle \in T^* - L']$.

THEOREM 4.2. *For all* $k \geqq 1$: *a language* L *is in* σ_k *if and only if there is some language* $L' \in$ DTIME(lin) *such that*

$$x \in L \Leftrightarrow (\exists y_1)_x (\forall y_2)_x \cdots (Qy_k)_x [\langle x, y, \cdots, y_k\rangle \in L'].$$

The quantifiers in this expression alternate between existential and universal, so that the jth quantification (from the left) is $(\exists y_j)_x$ if j is odd, and $(\forall y_j)_x$ if j is even.

Proof. This theorem is an almost direct consequence of Corollary 3.6, due to the close correspondence between existential quantification and homomorphisms. The constructions for the basis and induction steps are the same and are therefore combined. For notational convenience, let $k \geqq 1$ be odd.

First, suppose that for $L_2 \in$ DTIME(lin), a language $L_1 \subseteq S^*$ is defined by: $x \in L_1 \Leftrightarrow (\exists y_1)_x (\forall y_2)_x \cdots (\exists y_k)_x [\langle x, y_1, \cdots, y_k\rangle \in L_2]$. Let $T = S \cup \{\#\}$ and let $h: ([T]^2)^* \to T^*$ be the length-preserving homomorphism determined by defining for $a, b \in T$, $h([a, b]) = a$. Define $R = \{\langle x, y\rangle : |y| \leqq |x|\}$, a regular set; notice that for $z = \langle x, y\rangle \in R$, $h(z) = x$. Let $L_3 = \{\langle z, y_2, \cdots, y_k\rangle : z$ is of the form $\langle x, y_1\rangle$ and $\langle x, y_1, \cdots, y_k\rangle \in L_2\}$ and let L_4 be the language defined from L_3 by: $z \in L_4 \Leftrightarrow (\exists y_2)_z \cdots (\forall y_k)_z$ $[\langle z, y_2, \cdots, y_k\rangle \notin L_3]$. Then $L_1 = h(R - L_4)$, since for $z_1 = \langle x, y_1\rangle \in R$, $|z_1| = |x|$ and so $z_1 \notin L_4 \Leftrightarrow (\forall y_2)_x \cdots (\exists y_k)_x [\langle x, y_1, y_2, \cdots, y_k\rangle \in L_2]$. If $k > 1$, then (since $L_3 \in$ DTIME(lin)) by the induction hypothesis $L_4 \in \sigma_{k-1}$ so $L_1 \in \sigma_k$. If $k = 1$, then $L_2 = L_3$ and $R - L_4 = R \cap L_2$ is in DTIME(lin) so $L_1 \in$ NTIME$(n) = \sigma_1$.

Conversely, suppose $L_1 \in \sigma_k$. Then for $k = 1$, from [4] $L_1 = h(T^* - L_2)$ where $L_2 \in$ DTIME(lin) and h is a length-preserving homomorphism; for $k > 1$, from Corollary 3.6, $L_1 = h(T^* - L_2)$ where $L_2 \in \sigma_{k-1}$ and h is a length-preserving homomorphism. In either case, we see that there is a language $L_3 \in$ DTIME(lin) such that $z \in L_2 \Leftrightarrow (\exists y_1)_z \cdots (\forall y_{k-1})_z [\langle z, y_1, \cdots, y_{k-1}\rangle \in L_3]$, using the induction hypothesis if $k > 1$ and taking $L_3 = L_2$ if $k = 1$. Let $L_4 = \{\langle x, z, y_1, \cdots, y_{k-1}\rangle : h(z) = x$ and $\langle z, y_1, \cdots, y_{k-1}\rangle \notin L_3\}$. Then $L_4 \in$ DTIME(lin) and (since h is a length-preserving homomorphism) $x \in L_1 \Leftrightarrow (\exists z)_x (\forall y_1)_x \cdots (\exists y_{k-1})_x [\langle x, z, y_1, \cdots, y_{k-1}\rangle \in L_4]$. □

If σ_0 were defined to be DTIME(lin) rather than $\{\varnothing\}$ (which yields the same classes for $k \geqq 1$), then Theorem 4.2 would hold for all $k \geqq 0$.

Each of the classes σ_k is "principal", that is, can be generated from a language in the class by use of certain operations. This property can be used to distinguish between the classes in the linear hierarchy and some classes defined by resource-bounded Turing machines.

THEOREM 4.3. *For all* $k \geqq 1$, *there is a language* $A_k \in \sigma_k$ *with the properties that*: (i) *for every* $L \in \sigma_k$ *there is a homomorphism* h *such that* $L - \{e\} = h^{-1}(A_k)$; *and* (ii) $\sigma_{k+1} = \mathrm{NL}(\{A_k\})$.

The languages $A_1, A_2, \cdots$ are defined uniformly from $A_0 = \varnothing$ with the use of a particular nondeterministic linear-time oracle machine M_0: $A_{k+1} = M_0(A_k)$. The machine M_0 is constructed along the lines of the "universal" Turing machines in [6], [25]; further details may be found in [26]. If we use the closure properties given in Proposition 3.1, part (i) of this theorem implies that for all $k \geqq 1$, $\sigma_k = \{h^{-1}(A_k), h^{-1}(A_k \cup \{e\}) : h \text{ a homomorphism}\}$.

For each k, the language A_k is a "hardest" language for σ_k with respect to deterministic time-bounded or space-bounded recognition, in the same sense that the language exhibited by Greibach [10] is a hardest context-free language. Thus, for example, A_k can be accepted by a deterministic Turing machine in polynomial time if and only if every language σ_k can be so accepted.

Assume the linear hierarchy to be finite, so that $\mathrm{RUD} = \sigma_k$ for some $k \geqq 1$. Since $A_k \in \mathrm{RUD}$, from Theorem 3.12 A_k can be defined from the language $L_0 = \{0^n 1^n : n \geqq 0\}$ by use of some number of applications (say M) of the Boolean operations, inverse homomorphism and length-preserving homomorphism. But from Theorem 4.3, $\sigma_k = \{h^{-1}(A_k), h^{-1}(A_k \cup \{e\}) : h \text{ a homomorphism}\}$. Therefore, if the linear hierarchy is finite, then for any rudimentary relation R, $\theta(R)$ can be defined from L_0 by at most $M+2$ applications of the Boolean operations, inverse homomorphism and length-preserving homomorphism. Conversely, if a language L is obtained from L_0 by use of k of these operations, then $L \in \sigma_k$. Hence the linear hierarchy is infinite if and only if RUD cannot be generated from one language by use of a bounded number of applications of the Boolean operations, inverse homomorphism and length-preserving homomorphism. Also, in view of the proof of Theorem 3.11, RUD is properly contained in DSPACE(n) if and only if DSPACE(n) cannot be generated from L_0 by a bounded number of applications of these operations.

Recall from Proposition 3.7 that DSPACE($\log n$) and NSPACE($\log n$) are contained in the rudimentary relations, hence contained in $\cup \{\sigma_j : j \geqq 1\}$. Whether either of these families of languages is comparable to any σ_k is not known; however, the structure of the classes σ_k revealed in Theorem 4.3 gives partial information on this question.

PROPOSITION 4.4. *For all* $k \geqq 1$, σ_k *is not equal to either* DSPACE($\log n$) *or* NSPACE($\log n$).

Proof. As remarked previously, Theorem 4.3 implies that for all $k \geqq 1$, $\sigma_k = \{h^{-1}(A_k), h^{-1}(A_k \cup \{e\}) : h \text{ a homomorphism}\}$. However, such a representation (as a class generated by a single language under those operations) cannot hold for the classes DSPACE($\log n$) and NSPACE($\log n$): they are each the union of an infinite hierarchy of classes that are closed under inverse homomorphism and union with $\{e\}$ [14], [22]. □

By use of a similar proof, this proposition can be generalized to DSPACE($(\log n)^j$) and NSPACE($(\log n)^j$) for all $j \geqq 1$.

Neither DSPACE($\log n$) nor NSPACE($\log n$) is known to be closed under nonerasing homomorphism; their closure under this operation has the following consequences for the linear hierarchy.

COROLLARY 4.5. 1) *If* NSPACE($\log n$) *is closed under nonerasing homomorphism, then* NTIME(n) *is not closed under complementation.*

2) *If* DSPACE($\log n$) *is closed under nonerasing homomorphism, then the linear hierarchy is infinite (and, in particular,* NTIME(n) *is not closed under complementation).*

Proof. 1) If NTIME(n) is closed under complementation then, from Proposition 4.1, NTIME(n) = RUD, so NSPACE($\log n$) $\subseteq$ NTIME(n). But if NSPACE($\log n$) is closed under nonerasing homomorphism, then NTIME(n) $\subseteq$ NSPACE($\log n$) since NSPACE($\log n$) contains the Dyck sets and regular sets and is closed under intersection and inverse homomorphism. Since NTIME(n) $\neq$ NSPACE($\log n$), a contradiction results if both closure properties are assumed.

2) Suppose DSPACE($\log n$) is closed under nonerasing homomorphism. It is also closed under the Boolean operations and inverse homomorphism and contains $\{0^n 1^n : n \geqq 0\}$; hence from Theorem 3.12, RUD $\subseteq$ DSPACE($\log n$), so RUD = $\cup\{\sigma_j : j \geqq 1\}$ = DSPACE($\log n$). If in addition the linear hierarchy is finite, then there is some k such that RUD = σ_k = DSPACE($\log n$), contradicting Proposition 4.4; therefore the linear hierarchy must be infinite. □

5. Summary. We have seen two characterizations of the rudimentary relations that are restricted versions of characterizations of the arithmetical sets. For the characterization RUD = NL*($\{\varnothing\}$), the analogy between the two classes of relations follows through to the levels of the linear and arithmetical hierarchies, defined either using "r.e. in" or using alternations of quantifiers, with suitable restrictions in the linear case.

A question arises from the characterization of RUD given in Theorem 3.12: can any rudimentary (but nonregular) language be used in place of $\{0^n 1^n : n \geqq 0\}$? Or is there a family of languages lying strictly between the regular sets and RUD that is closed under the Boolean operations, inverse homomorphism and length-preserving homomorphism? A related question is whether any nonregular context-free language generates all the context-free languages under these operations.

Another set of questions concerns the structure of RUD, whether, e.g., a representation as in Theorem 4.3 holds for all of RUD, with some language A in place of A_k. From Proposition 3.1 and Theorem 4.3, each class σ_k is a "principal AFL" [8]; therefore the linear hierarchy is infinite if and only if RUD is not a principal AFL (although it is an AFL). Also, if RUD is not principal, then RUD is properly contained in DSPACE(n).

Two extensions are possible. First, as in [26], [28], the results can be translated into polynomial bounds, to investigate the polynomial hierarchy [24] and the class of "extended" rudimentary relations [2]. Second, the results can be "relativized" by considering an arbitrary language A in place of the empty set. With a natural definition of "rudimentary in A" and RUD(A) (where RUD = RUD($\varnothing$)), it can be shown that RUD(A) = NL*($\{A\}$) and, when A is added to the basis sets, that Theorems 3.12 and 4.2 hold.

REFERENCES

[1] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the P = ?NP question*, this Journal, 4 (1975), pp. 431–444.

[2] J. H. BENNETT, *On Spectra*, Ph. D. dissertation, Princeton Univ., Princeton, NJ, 1962.

[3] R. BOOK, *On the Chomsky–Schützenberger theorem*, Tech. Rep. Dept. of Computer Sci., Yale Univ., New Haven, CT, 1975.

[4] R. BOOK AND S. GREIBACH, *Quasirealtime languages*, Math. Systems Theory, 4 (1970), pp. 97–111.

[5] R. BOOK, S. GREIBACH AND B. WEGBREIT, *Time- and tape-bounded Turing acceptors and AFLs*, J. Comput. System Sci., 4 (1970), pp. 606–621.
[6] R. BOOK, S. GREIBACH, B. WEGBREIT AND O. IBARRA, *Tape-bounded Turing acceptors and principal AFLs*, Ibid., 4 (1970), pp. 622–625.
[7] S. COOK, *The complexity of theorem-proving procedures*, Proc. Third ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–158.
[8] S. GINSBURG AND S. GREIBACH, *Abstract families of languages*, Mem. Amer. Math. Soc., 87 (1969), pp. 1–32.
[9] S. GINSBURG, S. GREIBACH AND J. HOPCROFT, *Pre-AFL*, Ibid., 87 (1969), pp. 41–51.
[10] S. GREIBACH, *The hardest context-free language*, this Journal, 2 (1973), pp. 304–310.
[11] A. GRZEGORCZYK, *Some classes of recursive functions*, Rozprawy Matematyczne, IV (1953), pp. 1–46.
[12] J. HARTMANIS AND J. HOPCROFT, *What makes some language theory problems undecidable*, J. Comput. System Sci., 4 (1970), pp. 368–376.
[13] J. HOPCROFT AND J. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.
[14] O. IBARRA, *On two-way multihead automata*, J. Comput. System Sci., 7 (1973), pp. 28–36.
[15] N. JONES, *Context-free languages and rudimentary attributes*, Math. Systems Theory, 3 (1969), pp. 102–109.
[16] N. LYNCH, *Relativization of the theory of computational complexity*, Project MAC Tech. Rep. 99, Mass. Inst. of Tech., Cambridge, MA, 1972.
[17] G. MYHILL, *Linear bounded automata*, Wright Air Devel. Div., Tech. Note 60–165, Wright-Patterson Air Force Base, OH, 1960.
[18] V. A. NEPOMNYASHCHII, *Rudimentary interpretation of two-tape Turing computation*, Kibernetika (Kiev), 6 (1970), pp. 29–35, Cybernetics, Dec. (1972), pp. 43–50.
[19] R. RITCHIE, *Classes of predictably computable functions*, Trans. Amer. Math. Soc., 106 (1963), pp. 139–173.
[20] R. RITCHIE AND F. SPRINGSTEEL, *Language recognition by marking automata*, Information and Control, 20 (1972), pp. 313–330.
[21] H. ROGERS, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.
[22] J. SEIFERAS, *Nondeterministic time and space complexity classes*, Project MAC Tech. Rep., Mass. Inst. of Tech., Cambridge, MA, 1974.
[23] R. SMULLYAN, *Theory of Formal Systems*, Annals of Mathematics Studies No. 47, Princeton University Press, Princeton, NJ, 1961.
[24] L. STOCKMEYER, *The polynomial-time hierarchy*, Theor. Comput. Sci., 3 (1976), pp. 1–22.
[25] B. WEGBREIT, *A generator of context-sensitive languages*, J. Comput. System Sci., 3 (1969), pp. 456–461.
[26] C. WRATHALL, *Subrecursive predicates and automata*, Ph.D. dissertation, Harvard Univ., Cambridge, MA, 1975; also Res. Rep. 56, Dept. of Computer Sci., Yale Univ., New Haven CT, 1975.
[27] ———, *Characterizations of the Dyck sets*, RAIRO—Informatique Théorique, 11 (1977), pp. 53–62.
[28] ———, *The linear and polynomial-time hierarchies*, in preparation.
[29] Y. YU, *Rudimentary relations and formal languages*, Ph.D. dissertation, Univ. of California, Berkeley, 1970.

SIAM J. COMPUTING
Vol. 7, No. 2, May 1978

FINITE CAPACITY QUEUING SYSTEMS WITH APPLICATIONS IN COMPUTER MODELING*

A. G. KONHEIM† AND M. REISER†

Abstract. A queuing system with a buffer of unlimited capacity in front of a cyclic arrangement of two exponential server queues is analyzed. The main feature of the system is blocking, i.e., when the population in the two queues attains a maximum value M, say, new arrivals are held back in the buffer. The solution is given in form of polynomial equations which require the roots of a characteristic equation. A solution algorithm is provided. The stability condition is given in terms of these roots and also in explicit form. Limiting cases which are of practical interest are discussed. These limiting cases lead to a better understanding of some popular approximation techniques.

Key words. queuing systems, blocking, finite waiting room, generating function method, multiprogramming with job queue

1. Introduction. The system control programs of multiprogrammed computers are typically built around two queues, often called eligible queue and multiprogramming queue. Jobs in the eligible queue are ready to receive service but must wait while jobs in the multiprogramming queue are those actively sharing the resources CPU, channels and I/O devices. A component of the system control program, called the scheduler, regulates the flow of jobs in and out of the multiprogramming queue. This situation is depicted schematically in Fig. 1.

The reason for multiprogramming is to make use of resources which can be utilized in parallel. However, in order to avoid inefficiencies, the number of jobs in the multiprogramming queue, called the level of multiprogramming, needs to be carefully controlled. This control is one of the functions provided by the scheduler.

In this paper, we shall analyze the model of Fig. 2(a) with a fixed upper level of multiprogramming denoted by M. The EXECUTE box of Fig. 1 is replaced by two servers, server 1 representing the CPU and server 2 representing a typical I/O device (such as a paging device, for example). The CPU queue and the I/O queue together make up the multiprogramming queue. After completion of a compute interval of the CPU, a job either leaves (probability $\pi_{1,3}$), joins the I/O device (probability $\pi_{1,2}$) or goes back to the eligible queue once more (probability $1-\pi_{1,3}-\pi_{1,2}$). The cyclic arrangement of CPU and I/O queues corresponds to the alternating sequence of compute and I/O intervals which typically characterizes the behavior of computer programs (also called jobs). We may also allow jobs to depart after having received an I/O service (probability $\pi_{2,3}$). In our model, the scheduler simply checks the level of multiprogramming K, say, upon arrival of a job. If $K<M$, the job is admitted at the CPU; if $K=M$, the job is held back in the eligible queue which is then said to be blocked. Note that this operation of the scheduler is typical for conventional multiprogrammed computer systems where the fixed upper level of multiprogramming corresponds to a certain number of initiators. In the case of virtual memory systems, M is often allowed to fluctuate. However, storage constraints impose an upper level in this case too.

The probability assumptions we make are Poisson arrivals (with rate λ) and independent exponentially distributed service times (rate μ_1, μ_2 respectively).

Traditionally, the EXECUTE box of Fig. 1 is analyzed by closed queuing models, often called central server models [8]. A closed model implies a fixed level of

* Received by the editors February 19, 1976, and in final revised form September 9, 1977.
† IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

multiprogramming. While the closed models were quite successful in their prediction of CPU utilization and throughout they are not capable of analyzing overall delay times. In order to get an estimate of the time spent in the WAIT box of Fig. 1, closed models are sometimes combined in a hierarchical manner [6], [7]. Such hierarchical models, however, are only approximations to the true solution. It is our objective to give an exact and analytical solution. We realize, that our model is still idealized. The Poisson source, the stochastic routing rule, the exponential servers and the limitation to only one job class may all be challenged. However, since a realistic model seems quite intractable, we feel that we have to approach the problem with a series of models, each one typical for one particular aspect. Such analytical solutions, then provide the foundation for and add credibility to approximate models.

Our analytical treatment in fact yields insight into the systems behavior over the entire parameter space. We shall isolate the average number of cycles in the CPU-I/O loop as the most critical parameter. Depending on this parameter, the system either exhibits complete hierarchical decomposability or may not be decomposable at all. We conclude that for the range of parameter values typical for computer systems, the hierarchical modeling technique is quite accurate.

There are only a few solutions to queuing systems with blocking in the literature. A special case of the system in Figs. 2(a), 2(b) has been analyzed in [1]. Results without feedback, but under more general statistical assumptions at server 1 are found in [2] and for the special case of $M = 1$ in [3].

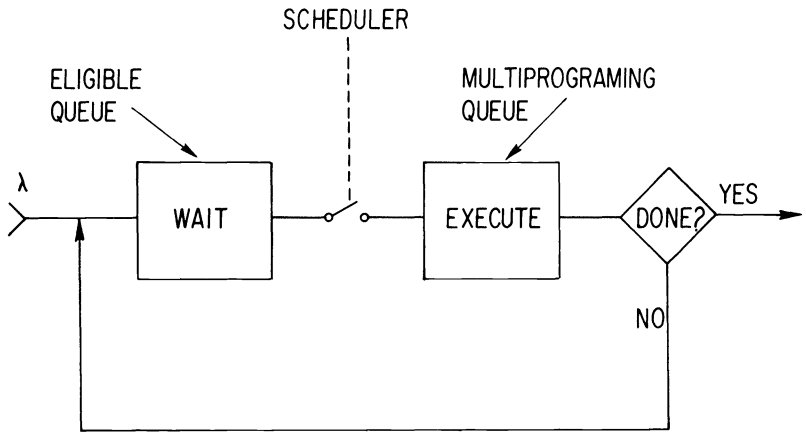


FIG. 1. *Schematic diagram of the structure of a multiprogrammed computer system.*

2. Analytical results.

2.1. Problem formulation. We shall study the exponential service queuing system $\mathcal{S}_M$ of Fig. 2(a). The state at time t is defined to be the vector valued random variable $\mathbf{X}_t = (X_t^{(1)}, X_t^{(2)}, X_t^{(3)})$ where $X_t^{(i)}$ is the number of jobs waiting or in service at the ith queue. Since always either

(i) $0 \leqq X_t^{(1)} + X_t^{(2)} \leqq M$ and $X_t^{(3)} = 0$, or

(ii) $X_t^{(1)} + X_t^{(2)} = M$ and $X_t^{(3)} > 0$,

we can reduce the dimensionality of the state by defining $\mathbf{Y}_t = (Y_t^{(1)}, Y_t^{(2)})$ where $Y_t^{(1)} = X_t^{(1)} + X_t^{(3)}$ and $Y_t^{(2)} = X_t^{(2)}$.

We shall only be interested in the stationary state probabilities $\{p_{i,j}\}$ of the process $\mathbf{Y}$ which are defined by $p_{i,j} = \lim_{t\to\infty} \Pr\{Y_t^{(1)} = i \text{ and } Y_t^{(2)} = j\}$. The process $\mathbf{Y}$ is an irreducible continuous time Markov process: hence $\{p_{i,j}\}$ always exist.

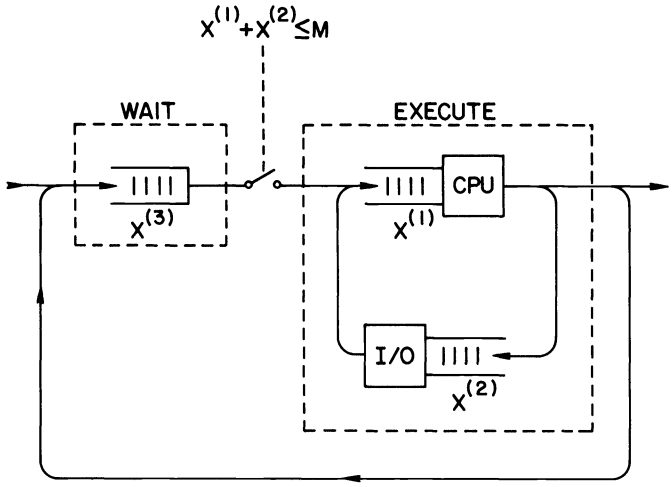


FIG. 2(a). *The queuing system $\mathcal{S}_M$ as it relates to the block diagram of Fig. 1. Note that we may also allow departures from the I/O server (not depicted).*

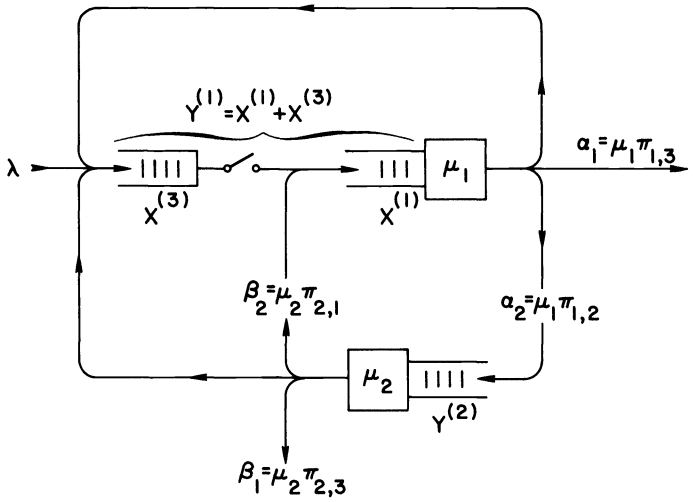


FIG. 2(b). *The queuing system $\mathcal{S}_M$ with full generality of routing. The meaning of the parameters are: μ_1, μ_2: rates of the exponential service processes, $\pi_{1,3}$: probability of leaving after a CPU service, $\pi_{1,2}$: probability of requiring an I/O service after a CPU service, $1-\pi_{1,2}-\pi_{1,3}$: probability of joining the wait queue after a CPU service, $\pi_{2,3}$: probability of leaving after an I/O service, $\pi_{2,1}$: probability of requiring another CPU service after an I/O service, $1-\pi_{2,1}-\pi_{2,3}$: probability of joining the wait queue after I/O service, λ: arrival rate, α_1: rate of departure process from CPU, β_1: rate of departure process from I/O, α_2: rate of arrival process at the I/O server, β_2: rate of process routed from I/O to CPU, M: capacity ($X^{(1)}+X^{(2)}\leqq M$).*

The stationary probabilities satisfy a system of linear equations. For simplicity of notation, we define $\alpha_1 = \mu_1\pi_{1,3}$, $\alpha_2 = \mu_1\pi_{1,2}$, $\beta_1 = \mu_2\pi_{2,3}$ and $\beta_2 = \mu_2\pi_{2,1}$. We will also assume, that the rates μ_1 and μ_2 are normalized such that $\lambda = 1$. There results

$$(1+\alpha_1+\alpha_2+\beta_1+\beta_2)p_{i,j} = p_{i-1,j} + \beta_2 p_{i-1,j+1} + \beta_1 p_{i,j+1} + \alpha_1 p_{i+1,j} + \alpha_2 p_{i+1,j-1} \qquad (0<i<\infty,\ 0<j<M), \tag{1a}$$

$$(1+\beta_1+\beta_2)p_{i,M} = p_{i-1,M} + \alpha_2 p_{i+1,M-1} \qquad (0<i<\infty), \tag{1b}$$

$$(1+\alpha_1+\alpha_2)p_{i,0} = p_{i-1,0} + \beta_2 p_{i-1,1} + \beta_1 p_{i,1} + \alpha_1 p_{i+1,0} \qquad (0<i<\infty), \tag{1c}$$

$$(1+\beta_1+\beta_2)p_{0,j} = \beta_1 p_{0,j+1} + \alpha_1 p_{1,j} + \alpha_2 p_{1,j-1} \qquad (0<j<M), \tag{1d}$$

$$(1+\beta_1+\beta_2)p_{0,M} = \alpha_2 p_{1,M-1}, \tag{1e}$$

$$p_{0,0} = \beta_1 p_{0,1} + \alpha_1 p_{1,0}. \tag{1f}$$

The system (1) is homogeneous and hence always admits the solutions $p_{i,j} = 0$ $0 \leqq i < \infty$ and $0 \leqq j \leqq M$. Under certain conditions, however, a *nonnull absolutely summable solution* exists (i.e., $p_{i,j} \neq 0$ for some i and j and $\sum_{i,j} |p_{i,j}| < \infty$). If such a solution exists the system is said to be *stable*, *unstable* otherwise. If (1) admits a stable solution, then the $\{p_{i,j}\}$ are strictly positive and the proper probabilities are found after normalization of this solution, i.e.,

$$\sum_{i=0}^{\infty}\sum_{j=0}^{\infty} p_{i,j} = 1. \tag{2}$$

The unconstrained solution of $\mathcal{S}_\infty$ is well known. If we set $M = \infty$ and omit equations (1b) and (1e) from the set of equations (1) we find that the probabilities $\{p_{i,j}\}$ satisfy the simple product-form solution

$$p_{i,j} = \begin{cases} (1-\rho_1)(1-\rho_2)\rho_1^i\rho_2^j & \text{if } \rho_1<1 \text{ and } \rho_2<1, \\ 0 & \text{otherwise (for } i \geqq 0,\ j \geqq 0) \end{cases} \tag{3}$$

with

$$\rho_1^{-1} = \alpha_1 + \alpha_2\frac{\beta_1}{\beta_1+\beta_2} \tag{4}$$

and

$$\rho_2^{-1} = \beta_1 + (\beta_1+\beta_2)\frac{\alpha_1}{\alpha_2}. \tag{5}$$

To study (1), we introduce *generating functions* as follows:

$$P_j(z) = \sum_{i=0}^{\infty} p_{i,j}z^i \qquad (0 \leqq j \leqq M), \tag{6}$$

where z is a complex variable of modulus at most one. The linear system (1) translates directly into an equivalent system, namely

$$\Gamma(z)\underline{P}(z) = \Delta(z)\underline{p}_0 \tag{7}$$

where $\underline{P}(z) = [P_0(z), P_1(z), \cdots, P_M(z)]'$, $\underline{p}_0 = [p_{0,0}, p_{0,1}, \cdots, p_{0,M}]'$, $\Gamma(z)$ is the

$(M+1)\times(M+1)$ matrix

$$(8)\qquad \begin{bmatrix} b_0(z) & -c(z) & & & & & \\ -\alpha_2 & b(z) & -c(z) & & & & \\ & -\alpha_2 & b(z) & -c(z) & & 0 & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & 0 & & & -\alpha_2 & b(z) & -c(z) \\ & & & & & -\alpha_2 & b_M(z) \end{bmatrix},$$

$\Delta(z)$ is the $(M+1)\times(M+1)$ matrix

$$(9)\qquad \begin{bmatrix} d(z) & & & & & 0 \\ -\alpha_2 & d(z) & & 0 & & 0 \\ & -\alpha_2 & d(z) & & & \\ & & \cdot & \cdot & & \\ & & & \cdot & \cdot & \\ & 0 & & -\alpha_2 & d(z) & 0 \\ & & & & -\alpha_2 & 0 \end{bmatrix},$$

and $b(z)$, $c(z)$, $d(z)$, $b_0(z)$, $b_M(z)$ polynomials, defined by

$$(10)\qquad b(z)=-\alpha_1+(1+\alpha_1+\alpha_2+\beta_1+\beta_2)z-z^2,$$

$$(11)\qquad c(z)=\beta_1 z+\beta_2 z^2,$$

$$(12)\qquad d(z)=-\alpha_1+(\alpha_1+\alpha_2)z,$$

$$(13)\qquad b_0(z)=-\alpha_1+(1+\alpha_1+\alpha_2)z-z^2$$

and

$$(14)\qquad b_M(z)=(1+\beta_1+\beta_2)z-z^2.$$

Since $\underline{P}(0)=p_0$ we obtain from (7) $[\Gamma(0)-\Delta(0)]p_0=0$. However, we have $\Gamma(0)-\Delta(0)\equiv 0$ which leaves $\underline{p}_0$ undetermined. Clearly, additional conditions are required.

Except for those values z which cause $\det[\Gamma(z)]$ to vanish, we may write the solution of (7) as

$$(15)\qquad \underline{P}(z)=\Gamma^{-1}(z)\Delta(z)\underline{p}_0$$

or formally

$$(16)\qquad P_j(z)=N_j(\underline{p}_0, z)/D_M(z), \qquad 0\leqq j\leqq M,$$

where $D_M(z)=\det[\Gamma(z)]$ is a polynomial of degree $2M$ and $N_j(\underline{p}_0, z)$ is a polynomial of degree at most $2M-1$ which is a linear functional of $\underline{p}_0$. Now by definition, $P_j(z)$ $(0\leqq j\leqq M)$ is required to be analytic inside the complex unit disk $\{z:|z|<1\}$ and continuous on its boundary $\{z:|z|=1\}$. Therefore if the *characteristic equation* $D_M(z)=0$ has any roots in the closed unit disk we require the numerators $N_j(\underline{p}_0, z)$ $(0\leqq j\leqq M)$ to vanish at those points. Thus each such root, ζ say, provides one additional condition $N_j(\underline{p}_0, \zeta)=0$ on $\underline{p}_0$.

We shall subsequently give a recursive derivation of $D_M(z)$ and show that if $\mathcal{S}_M$ is stable we always have the right number of independent equations to determine $\underline{p}_0$.

2.2. Recursive solution. The tridiagonal form of $\Delta(z)$ allows us to simplify the solution of (7). By means of successive backward substitution, we can express $P_j(z)$ $(0 \leqq j < M)$ in terms of $P_M(z)$ and the boundary values $\{p_{0,M-1}, p_{0,M-2}, \cdots p_{0,j}\}$. A straightforward computation yields

$$P_j(z) = A_{M-j}(z)P_M(z) + \sum_{l=j}^{M-1} p_{0,j}A_{l-j}(z) \tag{17}$$

where $\{A_j(z)\}$ are polynomials defined recursively by

$$A_0(z) = 1, \tag{18a}$$

$$A_1(z) = b_M(z)/\alpha_2, \tag{18b}$$

$$A_l(z) = \frac{1}{\alpha_2}[b(z)A_{l-1}(z) - c(z)A_{l-2}(z)] \qquad (1 < l < \infty). \tag{18c}$$

A relation for $P_M(z)$ is found by substituting $P_1(z)$ and $P_0(z)$ as given by (17) into the first equation of (7). We find the relation

$$\begin{aligned} -c(z)P_1(z) + b_0(z)P_0(z) - d(z)p_{0,0} \\ = [-c(z)A_{M-1}(z) + b_0(z)A_M(z)]P_M(z) \\ + [-c(z)A_{M-2}(z) + b_0(z)A_{M-1}(z)]p_{0,M-1} \\ + [-c(z)A_{M-3}(z) + b_0(z)A_{M-2}(z)]p_{0,M-2} \\ + \cdots + [b_0(z) - d(z)]p_{0,0} = 0. \end{aligned} \tag{19}$$

It is easy to see, that polynomials $b_0(z) - d(z)$ and $-c(z)A_l(z) + b_0(z)A_{l-1}(z)$ $(l = 1, 2, \cdots)$ have zeros at $z = 0$ and at $z = 1$. We define therefore

$$D_0(z) = [b_0(z) - d(z)]/(z(z-1)) = 1 \tag{20}$$

and

$$D_l(z) = [b_0(z)A_l(z) - c(z)A_{l-1}(z)]/(z(z-1)) \qquad (0 < l < \infty). \tag{21}$$

Now, we may rewrite (19) as follows

$$P_M(z) = -\frac{p_{0,0}D_0(z) + p_{0,1}D_1(z) + \cdots + p_{0,M-1}D_{M-1}(z)}{D_M(z)}. \tag{22}$$

The polynomials $\{D_j(z)\}$ are linear combinations of the recursively defined polynomials $\{A_j(z)\}$. Hence, they satisfy the same recursion with the following initial conditions

$$D_0(z) = 1, \tag{23a}$$

$$D_1(z) = \frac{1}{\alpha_2}[-(\delta + \alpha_1) + (1 + \alpha_1 + \alpha_2 + \alpha_3 + \beta_1 + \beta_2)z - z^2], \tag{23b}$$

$$D_l(z) = \frac{1}{\alpha_2}[b(z)D_{l-1}(z) - c(z)D_{l-2}(z)] \qquad (1 < l < \infty) \tag{23c}$$

where $\delta = \alpha_1(\beta_1 + \beta_2) + \alpha_2\beta_1$.

Equation (22) expresses $P_M(z)$ in form of a rational function. The analyticity argument yields for each root ζ of $D_M(z) = 0$ which is inside the closed unit disk (i.e.,

$|\zeta| \leqq 1$) a linear relation for the $\{p_{0,j}\}$, viz.

$$\sum_{l=1}^{M-1} p_{0,l} D_l(\zeta) = 0. \tag{24}$$

2.3. The characteristic equation. Further study of the problem requires information about the roots of the characteristic equation $D_M(z) = 0$. We introduce the generating function

$$\mathcal{D}(z, w) = \sum_{l=0}^{\infty} w^l D_l(z) \tag{25}$$

for which we obtain from the recurrence (23)

$$\mathcal{D}(z, w) = \frac{1 - w/\rho_2}{1 - (b(z)/\alpha_2)w + (c(z)/\alpha_2)w^2} = \frac{1 - w/\rho_2}{[1 - w/r_+(z)][1 - w/r_-(z)]} \tag{26}$$

where $r_+(z)$ and $r_-(z)$ are the roots of the quadratic $\alpha_2 - b(z)w + c(z)w^2 = 0$. $D_l(z)$ can be obtained from (26) by partial fractions. In general, such a partial fraction expansion does not yield rational results. There are, however, some special values of z for which closed form expressions for $D_l(z)$ can be obtained. We summarize those results in

LEMMA 1. *For $l = 1, 2, \cdots$ we find*

$$D_l(z) = (-1/\alpha_2)^l z^{2l} + O(z^{2l-1}) \quad \text{as } z \to \infty, \tag{27}$$

$$D_l(0) = (1 + \delta/\alpha_1)(-\alpha_1/\alpha_2)^l, \tag{28}$$

$$D_l(1) = \begin{cases} \dfrac{\rho_1 - 1}{\rho_1 - \rho_2}\left(\dfrac{\rho_1}{\rho_2}\right)^l - \dfrac{\rho_2 - 1}{\rho_1 - \rho_2} & \text{if } \rho_1 \neq \rho_2, \\ (l+1) - l/\rho & \text{if } \rho = \rho_1 = \rho_2, \end{cases} \tag{29}$$

$$D_l(\rho_1^{-1}) = \left[\frac{\beta_2}{\beta_1 + \beta_2}\frac{1}{\rho_1} + \frac{\beta_1}{\beta_1 + \beta_2}\right]^l \tag{30}$$

and

$$D_l(\rho_3^{-1}) = \left[\frac{\beta_2 + 1}{\alpha_1 + \alpha_2}\frac{1}{\rho_3}\right]^l \tag{31}$$

with

$$\rho_3^{-1} = \beta_1 + (1 + \beta_2)\frac{\alpha_1}{\alpha_1 + \alpha_2}. \tag{32}$$

Proof. Equation (27) follows directly from the recursion (23). The proof of (28) to (31) requires straightforward but extremely tedious computations. We found the use of an automatic symbol manipulation system [4] an essential help.

LEMMA 2. *Assume $\rho_1 < 1$ and $\rho_2 < 1$. Then, there exists an integer l^*, say, such that $D_l(1) \geqq 0$ for $0 \leqq l \leqq l^*$ and $D_l(1) < 0$ for $l^* < l < \infty$.*

Proof. Assume that $\rho_1 < \rho_2$. Then $A = (\rho_1 - 1)/(\rho_1 - \rho_2) > 0$, $B = (\rho_2 - 1) \div (\rho_1 - \rho_2) > 0$ and $\gamma = \rho_1/\rho_2 < 0$. $D_l(1) = A\gamma^l - B$ is monotonically decreasing with l. Since $D_l(1) > 0$ and $D_\infty(1) < 0$, an l^* with the properties of Lemma 2 exists. We assume now that $\rho_1 > \rho_2$. Then $A < 0$, $B < 0$ and $\gamma > 0$. $D_l(1) = B - A\gamma^l$ is again monotonically decreasing. The existence of l^* follows from the fact that $D_0(1) = 1 > 0$ and $D_\infty(1) = -\infty < 0$. Finally, it is easy to see, that l^* also exists for $\rho_1 = \rho_2$. Q.E.D.

LEMMA 3. *ρ_2 and ρ_3 are related as follows*:

$$\rho_3 \geqq 1 \Leftrightarrow \rho_2 \geqq 1 \quad \text{and} \quad \rho_3 < 1 \Leftrightarrow \rho_2 < 1.$$

Proof. Define $\phi = \pi_{1,3}/(\pi_{1,3}+\pi_{1,2}) = \alpha_1/(\alpha_1+\alpha_2)$, $\theta = \pi_{2,3}/(\pi_{2,3}+\pi_{2,1}) = \beta_1/(\beta_1+\beta_2)$, $\hat{\mu}_2 = \mu_2(\pi_{2,3}+\pi_{2,1})$ and

$$x(\phi) = \hat{\mu}_2[\theta + \phi/(1-\phi)], \tag{33}$$

$$y(\phi) = \hat{\mu}_2[\theta + (1-\theta)\phi] + \phi. \tag{34}$$

Clearly $\rho_2^{-1} = x(\phi)$ and $\rho_3^{-1} = y(\phi)$. The parameters are constrained by $\hat{\mu}_2 > 0$, $0 \leqq \theta \leqq 1$, $0 \leqq \phi \leqq 1$. The curve $x(\phi)$ vs. ϕ is a hyperbola, $y(\phi)$ vs. ϕ is a straight line. $x(\phi)$ intersects $y(\phi)$ at $\phi_1 = 0$ and $\phi_2 = (1-\beta_1)/(\hat{\mu}_2 + 1 - \beta_1)$. The corresponding function values are $x(\phi_1) = y(\phi_1) = \beta_1$ and $x(\phi_2) = y(\phi_2) = 1$. The assertion of Lemma 2 follows easily by arguments from elementary calculus (which we omit). Q.E.D.

Define $\eta_- = \min\{\rho_1^{-1}, \rho_3^{-1}\}$ and $\eta_+ = \max\{\rho_1^{-1}, \rho_3^{-1}\}$. We are now ready for

THEOREM 1. *Assume $\alpha_1 > 0$. Then $D_M(z) = 0$ has $2M$ real and single roots $\zeta_{M,1}$, $\zeta_{M,2}, \cdots, \zeta_{M,M-1}$, $\eta_{M,1}, \eta_{M,2}, \cdots, \eta_{M,M+1}$ which have the following properties*:

1) *There are M roots in $(0, \eta_-)$ and M roots in (η_+, ∞) which interleave as follows with the roots of $D_{M-1}(z)$*:

$$0 < \zeta_{M,1} < \zeta_{M-1,1} < \zeta_{M,2} < \zeta_{M-1,2} < \cdots < \zeta_{M,M-1} < \eta_{M-1,1} < \eta_{M,1} < \eta_-$$

and

$$\eta_+ < \eta_{M,2} < \eta_{M-1,2} < \cdots < \eta_{M-1,M} < \eta_{M,M+1} < \infty.$$

2) *There are at least $M-1$ roots in $(0, 1)$, viz.*

$$0 < \zeta_{M,1} < \zeta_{M,2} < \cdots < \zeta_{M,M-1} < 1.$$

Proof. 1) Let κ be a quantity sufficiently large. Then by Lemma 1, sign $[D_l(\kappa)] = (-1)^l$, $D_l(\eta_+) > 0$, $D_l(\eta_-) > 0$ and sign $[D_l(0)] = (-1)^l$. Assume that $D_{l-2}(\zeta) \neq 0$. Then, since $c(z) > 0$ for all $0 < z < \infty$ we have sign $[D_l(\zeta)] = -\text{sign}\,[D_{l-2}(\zeta)]$.

2) $D_1(z) = 0$ has two roots $0 < \eta_{1,1} < \eta_-$ and $\eta_+ < \eta_{1,2} < \infty$ since $D_1(0) < 0$, $D_1(\eta_-) > 0$, $D_1(\eta_+) > 0$ and $D_1(\kappa) < 0$.

3) $D_2(z) = 0$ has four roots which interleave with the roots of $D_1(z) = 0$, viz. $0 < \zeta_{2,1} < \eta_{1,1} < \eta_{2,1} < \eta_-$ and $\eta_+ < \eta_{2,2} < \eta_{1,2} < \eta_{2,3} < \infty$. This follows the fact that $D_0(\eta_{1,1}) = D_0(\eta_{1,2}) = 1 > 0$ and hence $D_2(0) > 0$, $D_2(\eta_{1,1}) = -c(\eta_{1,1}) < 0$, $D_2(\eta_-) > 0$; $D_2(\eta_+) > 0$, $D_2(\eta_{1,2}) = -c(\eta_{1,2}) < 0$, $D_2(\kappa) > 0$.

4) The fact that $D_M(z)$ has $2M$ roots which interleave with the roots of $D_{M-1}(z)$ is proved by induction. We assume that $D_{M-1}(z)$ and $D_{M-z}(z)$ have interleaving roots. Then we find the following signature of $D_M(z)$:

$$\begin{aligned}
&D_M(\eta_-) > 0, \\
&D_M(\eta_{M-1,1}) < 0 \qquad \text{since } D_{M-2}(\eta_{M-1,1}) > 0, \\
&D_M(\zeta_{M-1,M-2}) > 0 \quad \text{since } D_{M-2}(\zeta_{M-1,M-2}) < 0, \\
&\quad \vdots \\
&(-1)^M D_M(\zeta_{M-1,1}) < 0 \qquad \text{since } (-1)^M D_{M-2}(\zeta_{M-1,1}) > 0, \\
&(-1)^M D_M(0) > 0
\end{aligned}$$

and

$$
\begin{aligned}
&D_M(\eta_+)>0\\
&D_M(\eta_{M-1,2})<0 \qquad \text{since } D_{M-2}(\eta_{M-1,2})>0,\\
&D_M(\eta_{M-1,3})>0 \qquad \text{since } D_{M-2}(\eta_{M-1,3})<0,\\
&\quad\vdots\\
&(-1)^M D_M(\eta_{M-1,M})<0 \qquad \text{since } (-1)^M D_{M-2}(\eta_{M-1,M})>0,\\
&(-1)^M D_M(\kappa)>0.
\end{aligned}
$$

We have accounted for $2M$ sign changes and hence for $2M$ real and single roots which interleave with those of $D_{M-1}(z)=0$.

5) If $\eta_- \leqq 1$, then assertion 2) of Theorem 1 clearly holds. So let us assume $\eta_- > 1$ which implies $\rho_1 < 1$ and $\rho_3 < 1$ and by Lemma 3 also $\rho_2 < 1$. Now by Lemma 2, we know that an l^* exists, such that $D_l(1) \geqq 0$ for $0 \leqq l \leqq l^*$ and $D_l(1) < 0$ for $l^* < l < \infty$. We now refer to Fig. 3. From the interleaving property and from the signs of $\{D_l(1)\}$ follow that exactly one branch $\{\eta_{l,1},\ l = 1, 2, \cdots\}$ crosses the line $z = 1$. This implies, that the $M-1$ roots $\zeta_{M,1}, \zeta_{M,2}, \cdots, \zeta_{M,M-1}$ are below $z = 1$ and hence in the interval $(0, 1)$. Q.E.D.

If $\alpha_1 = 0$ then $\zeta = 0$ becomes a multiple root of $D_M(z) = 0$. The interleaving property still holds for the remaining nonzero roots. Since this case was treated in detail in [1], we omit its discussion here for the sake of brevity.

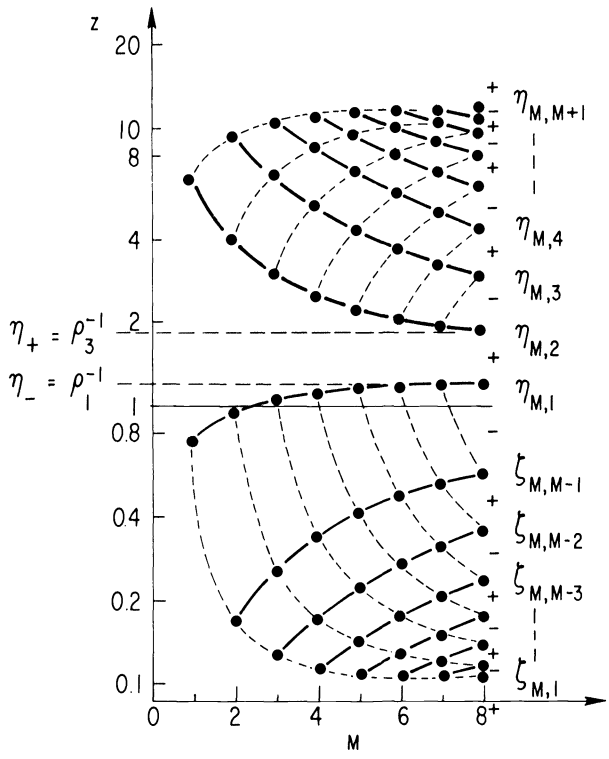


FIG. 3. *Roots of the characteristic equation* $D_M(z) = 0$ *vs. M. The parameter values are* $\alpha_1 = 1.2$, $\beta_1 = 0$, $\beta_2 = 3$.

2.4. The existence of a stable solution. It is the object of this section to investigate conditions under which a stable solution exists and to show that for a stable $\mathcal{S}_M$, the boundary values can be obtained from the following system of linear equations

$$\sum_{l=0}^{M-1} p_{0,l} D_l(\zeta_{M,k}) = 0, \qquad k = 1, 2, \cdots, M-1. \tag{35}$$

Our main result is summarized in

THEOREM 2. *The system $\mathcal{S}_M$ has a stable solution if and only if the characteristic equation $D_M(z)$ has exactly $M-1$ roots $\zeta_{M,1}, \zeta_{M,2}, \cdots, \zeta_{M,M-1}$ in $(0, 1)$. In this case, the system of linear equations* (35) *determines positive boundary values $\{p_{0,j}\}$ up to a constant factor, which follows from normalization.*

Proof. Necessity. 1) Assume that $\mathcal{S}_M$ is stable and that $D_M(z) = 0$ has $K > M-1$ roots in $(0, 1]$. Then (35) is augmented by the conditions

$$\sum_{l=0}^{M-1} p_{0,l} D_l(\zeta_{M,k}) = 0, \qquad k = 1, 2, \cdots, K-M+1. \tag{36}$$

We assert that no positive solution $\{p_{0,l}\}$ exists which contradicts the assumption of stability.

2) Assume on the contrary that a positive solution does exist. Then from the first equation of (36) (i.e., $k = 1$) we have

$$\sum_{l=0}^{M-1} p_{0,l} D_l(z) \to 0 \quad \text{as } z \to \eta_{M,1} \text{ (from above).} \tag{37}$$

But from the interleaving property of the roots and from $D_l(\eta_-) > 0$ we have $D_l(\eta_{M,1}) > 0$ for $0 \leqq l < M$ which contradicts (37).

Sufficiency. 3) We consider now the case that $D_M(z)$ has exactly $M-1$ roots $\zeta_{M,1}, \zeta_{M,2}, \cdots, \zeta_{M,M-1}$ in $(0, 1)$. Then (35) provides $M-1$ equations for M unknowns, hence a nonnull solution always exists.

4) A solution $\underline{p}_0$, say, of (35) makes the $\{P_j(z)\}$ defined by (17) and (22) analytic inside the unit disk and continuous on its boundary. Therefore, values $\{p_{i,j}\}$ can be obtained which are L_1-bounded and which are also a solution to system (1). But by the theory of Markov processes, (Foster's Theorem) a L_1-bounded solution must also be a positive solution.

5) It remains to be shown that $\underline{p}_0 = (p_{0,1}, p_{0,2}, \cdots, p_{0,M-1})$ is determined by (35) up to a constant factor.

6) Assume on the contrary, that another vector $\underline{q} = (q_1, q_2, \cdots, q_{M-1})$ exists which satisfies (35) and which is linearly independent of the vector of possible components $\underline{p}_0$. We assert, that an ε positive and sufficiently small can be chosen such that $(1-\varepsilon)\underline{p}_0 + \varepsilon\underline{q}$ substituted for $\underline{p}_0$ in (22) also defines a positive solution to (1). But this cannot occur since the $\{p_{i,j}\}$ are uniquely determined.

7) To show the existence of such an ε, we rewrite (16) in form of partial fractions, viz.

$$\begin{aligned} P_j(z) &= \sum_{i=0}^{\infty} z^i p_{i,j} = N_j(\underline{p}_0, z)/D_M(z) \\ &= \sum_{i=0}^{\infty} z^i \sum_{k=1}^{M+1} R_{j,k}(\underline{p}_0)\eta_{j,k}^{-1} \end{aligned} \tag{38}$$

where the $\{R_{i,k}(\underline{p}_0)\}$ are linear functionals in $\underline{p}_0$. A continuity argument shows that

$$\sum_{i=0}^{\infty} z^i \sum_{k=1}^{M+1} R_{i,k}((1-\varepsilon)\underline{p}_0+\varepsilon\underline{q})\rho_k^i$$

also defines a positive solution to system (1) for ε sufficiently small.

COROLLARY 1. *If $\rho_1<1$ and $\rho_2<1$, then $\mathcal{S}_M$ is stable iff $D_M(a)<0$.*

Proof. By Lemma 3, $\rho_1<1$ and $\rho_2<1$ implies $\eta_-<1$. Hence there are at most M roots in (0, 1). Thus, stability requires, that the largest of these roots, $\eta_{M,1}$, falls into $1<\eta_{M,1}<\eta_-$. Since $D_M(\eta_-)>0$ this requires $D_M(1)<0$. Q.E.D.

COROLLARY 2. *The stability of $\mathcal{S}_\infty$, i.e., the condition $\rho_1<1$ and $\rho_2<1$ is a necessary condition for stability of $\mathcal{S}_M$. Furthermore, if $\mathcal{S}_\infty$ is stable, there exists an M^* such that $\mathcal{S}_M$ is stable for $M^*<M<\infty$ and unstable for $0\leqq M\leqq M^*$.*

Proof. Assume that $\mathcal{S}_M$ is stable and that $\rho_1\geqq 1$ or $\rho_2\geqq 1$. This implies $\eta_-\leqq 1$ (Lemma 3). But $\eta_-\leqq 1$ implies that there are at least M roots in (0, 1] which contradicts the assumption of stability. The existence of M^* follows from Corollary 1 and Lemma 2.

The property of stability is a condition on the parameters λ, μ_1, μ_2, $\{\pi_{i,j}\}$ and M. We give an explicit form of this condition in

COROLLARY 3. *$\mathcal{S}_M$ is stable if and only if*

$$\lambda_M>\lambda=1$$

where

$$\lambda_M=\begin{cases}\dfrac{\rho_1^M-\rho_2^M}{\rho_1^{M+1}-\rho_2^{M+1}} & \text{if } \rho_1\neq\rho_2,\\[2ex] \dfrac{M}{M+1}\dfrac{1}{\rho} & \text{if } \rho=\rho_1=\rho_2.\end{cases} \tag{39}$$

Proof. Assume $\rho_1\neq\rho_2$. We may rewrite (39) as follows

$$\lambda_M=\frac{1-(\rho_2/\rho_1)^M}{1-(\rho_2/\rho_1)^{M+1}}\frac{1}{\rho_1}=\frac{1-(\rho_1/\rho_2)^M}{1-(\rho_1/\rho_2)^{M+1}}\frac{1}{\rho_2}. \tag{40}$$

Clearly $\lambda_M>1$ implies $\rho_1<1$ and $\rho_2<1$ as $M\to\infty$. Therefore by Corollary 1, $D_M(1)<0$ is a necessary and sufficient condition for stability. A simple computation shows that $D_M(1)$ as given by (29) may be written as

$$D_M(1)=\frac{1-(\rho_1/\rho_2)^{M+1}}{1-\rho_1/\rho_2}[1-\lambda_M]. \tag{41}$$

Since the first term of (41) is always positive

$$D_M(1)<0 \quad \text{implies} \quad 1-\lambda_M<1. \quad \text{Q.E.D.} \tag{42}$$

The region of stability in the (ρ_1, ρ_2)-plane is depicted in Fig. 4 for various values of M.

Corollary 3 has a simple physical interpretation in terms of the system $\mathcal{S}_M$. The quantity λ_M in (39) is the *departure rate of the saturated system* and Corollary 3 states simply that for stability, the arrival rate λ must be smaller than this (maximum) departure rate λ_M. For a more general class of systems without feedback, this interpretation of the stability limit is found in [5].

If the system $\mathcal{S}_M$ is saturated, then each job departing from stage 1 is immediately replaced. Thus, the *two servers act as a detached closed queuing system with*

population M. In this case, λ_M is the throughput of the branch labeled $\pi_{1,3}$ in Fig. 5. It is a simple exercise to derive (39) from the well known solution of the closed exponential server system.

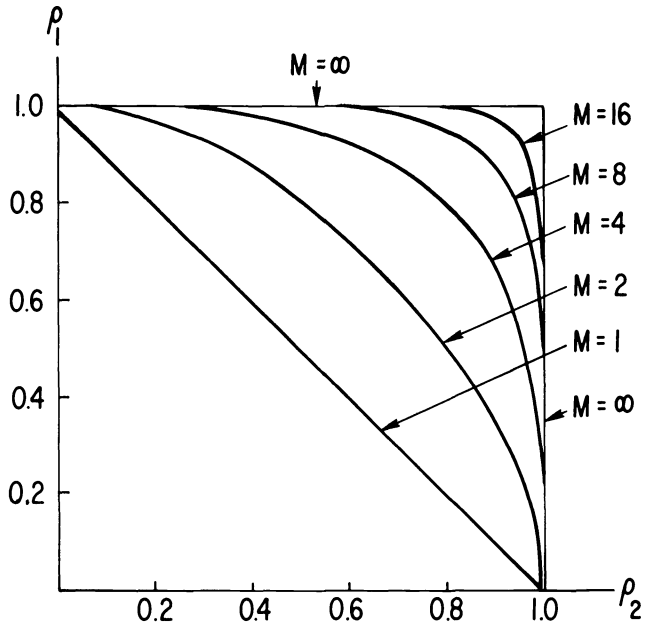


FIG. 4. *Stability chart for various values of M. $\mathcal{S}_M$ is stable, if its operating point (ρ_1, ρ_2) lies towards the origin of the stability curves.*

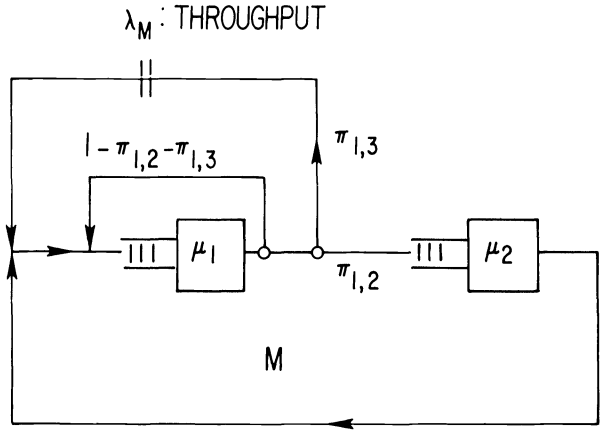


FIG. 5. *A closed queuing system related to the stability condition of Corollary* 3.

2.5. Algorithmic solution. If $\mathcal{S}_M$ is stable, then by (35) we have a set of positive boundary values which make $\{P_j(z): 0 \leqq j \leqq M\}$ analytic inside the unit disk and continuous on its boundary. By considerations given in § 2.1, we know that the functions $\{P_j(z): 0 \leqq j \leqq M\}$ are proper rational functions in z. Hence we may obtain $\{p_{i,j}\}$ by partial fractions as follows

$$p_{i,j} = \sum_{k=1}^{M+1} R_{j,k} \eta_{M,k}^{-i} \qquad (0 \leqq j \leqq M) \tag{43}$$

where $R_{j,k} = N_j(\eta_{M,k})/D'_M(\eta_{M,k})$.

It is not difficult to see, that

$$R_{j,k} = A_{M-j}(\eta_{M,k})R_{M,k} \qquad (0<j<M). \tag{44}$$

We are now ready to summarize the solution in algorithmic form:

Step 1. Calculate the polynomials $\{D_j(z): 0 \leqq j \leqq M\}$ using the recursion (23).

Step 2. Check for stability (i.e., $\rho_1 < 1$, $\rho_2 < 1$ and $D_M(1) < 0$). If $\mathcal{S}_M^*$ is unstable, then stop (Corollary 3).

Step 3. Determine the roots of the characteristic equation $D_M(z) = 0$.

Step 4. Determine a solution $\{p_{0,j}^*,\ 0 \leqq j \leqq M\}$ of (35).

Step 5. Compute $N_M(z) = \sum_{l=0}^{M-1} p_{0,l}^* D_l(z)$.

Step 6. Compute the residues $\{R_{M,k} = -N_M(\rho_{M,k}^{-1})/D'_M(\rho_{M,k}^{-1}): 1 \leqq k \leqq M+1\}$.

Step 7. Compute the polynomials $\{A_j(z): 0 \leqq j \leqq M\}$ using the recursion (18).

Step 8. Compute the residues $\{R_{j,k} = A_{M-j}(\eta_{M,k})R_{M,k}: 0 \leqq j < M\}$.

Step 9. Obtain the normalization constant

$$G = \sum_{j=0}^{M} \sum_{k=1}^{M+1} R_{j,k}/(1-\eta_{M,k}^{-1}).$$

Step 10. The solution is $p_{i,j} = \sum_{k=1}^{M+1} R_{j,k}\eta_{M,k}^{-1}/G$.

It is not difficult to implement this algorithm on a digital computer. Polynomials are conveniently represented as arrays. A set of subroutines for polynomial addition, subtraction, multiplication and division is useful. A simple root finder is sufficient since we know that all roots are real and single. Good use can be made of the interleaving property of the roots. We found APL to be particularly well suited for this kind of problem.

3. Numerical results, special cases and approximate solution.

3.1. Regimes of operation with numerical examples. We now revert to a discussion of our original system $\mathcal{S}_M$ of Fig. 2(a). From the six parameters M, λ, μ_1, μ_2, $\pi_{1,3}$ and $\pi_{1,2}$, only four play an essential role. Clearly, we can always choose units of time such that $\lambda = 1$ (as we have done throughout § 2). Furthermore, the queue size distribution is invariant under the transformation

$$\mu_1 \to \hat{\mu}_1 = \mu_1(\pi_{1,3} + \pi_{1,2}),$$

$$\pi_{1,3} \to \phi = \pi_{1,3}/(\pi_{1,2} + \pi_{1,3})$$

and

$$\pi_{1,2} \to (1-\phi) = \pi_{1,2}/(\pi_{1,2} + \pi_{1,3})$$

since α_1, α_2, β_1 and β_2 are not affected. M, $\hat{\mu}_1$, μ_2 and ϕ define an equivalent system without overall feedback. We call such a feedback loop a nonessential loop.

It will be more convenient to use the quadruplet $(M, \rho_1, \rho_2, \phi)$ instead of $(M, \hat{\mu}_1, \mu_2, \phi)$ since ρ_1, ρ_2 and ϕ are restricted to the interval (0, 1). Subsequently, we consider M to be a fixed parameter. The triplet (ρ_1, ρ_2, ϕ) defines an operating point of $\mathcal{S}_M$. We call $\mathcal{S}_M$ nondegenerate, if its operating point is in the domain

$$\mathcal{O}_M = \{(\rho_1, \rho_2, \phi): \rho_1 > 0, \rho_2 > 0, \lambda_M > 1, 0 < \phi < 1\}.$$

As an example, $\mathcal{O}_4$ is depicted in Fig. 6. Clearly, $\mathcal{O}_M$ contains all possible stable modes of operations. It is often useful, to interpret ϕ as a measure of $\bar{\omega}$, the average number of cycles a job makes between server 1 and server 2. The quantities $\bar{\omega}$ and ϕ are related by

$$\bar{\omega} = (1-\phi)/\phi. \tag{45}$$

We can identify regimes of operations of $\mathcal{S}_M$ according to three criteria, namely

1) The ratio $\gamma = \rho_1/\rho_2$. If γ is small, then the second server is the bottleneck. If γ is around unity, we have a balanced load and if $\gamma \gg 1$ then the first server is the bottleneck.
2) The value of λ_M. If λ_M is close to unity, then the operation point is close to the stability limit and hence $\mathcal{S}_M$ is close to saturation. On the other hand, $\lambda_M \gg 1$ means light load condition.
3) The value of ϕ. ϕ close to zero means a high value of $\bar{\omega}$, the average number of cycles. On the other extreme, ϕ close to unity yields small $\bar{\omega}$.

The expected values of $X^{(1)}$, $X^{(2)}$ and $X^{(3)}$ for the various regimes are given in Tables 1(a), 1(b) for the cases $M = 2$ and $M = 4$. (See Fig. 6 also.) It is interesting to observe, that while $E\{X^{(1)}\}$ and $E\{X^{(2)}\}$ are quite insensitive to changes in ϕ, $E\{X^{(3)}\}$, the average queue length of the buffer may vary strongly with ϕ.

Operating points on the boundary of $\mathcal{O}_M$ define degenerate systems $\mathcal{S}_M$. Meaningful solutions can be defined for degenerate systems only by means of a limiting process. The following is a list of degenerate cases:

(i) $\rho_1 = 0$ or $\rho_2 = 0$. The defining limit is $\mu_1 \to \infty$ or $\mu_2 \to \infty$. The solution becomes that of a simple $M/M/1$ queue.

(ii) $\lambda_M = 1$. $\mathcal{S}_M$ is saturated and no steady state probabilities exist for $\mathbf{X} = (X^{(1)}, X^{(2)}, X^{(3)})$. However, one can still define marginal probabilities for $(X^{(1)}, X^{(2)})$. In a saturated system, the input buffer is nonempty with probability one. Hence each departing job is immediately replaced. Therefore the two queues operate like the closed system of Fig. 5.

(iii) $\phi = 0$. The defining limit is $\phi \to 0$ such that ρ_1 and ρ_2 are constant which implies $\mu_1 \to \infty$ and $\mu_2 \to 0$. As $\phi \to 0$, jobs circle around queue 1 and queue 2 more and more and do that faster and faster. In analogy with the processor shared queue discipline, we call this case "shared processors". We will give the solution in § 3.2.

(iv) $\phi = 1$. The defining limit is $\phi \to 1$ such that ρ_1 and ρ_2 are fixed. This implies $\mu_2 \to 0$. Thus as $\phi \to 1$, the second queue is visited by fewer and fewer jobs but these jobs are likely to be held in service for longer and longer periods of time. We shall give the solution in § 3.3 where we will show that this case is always unstable.

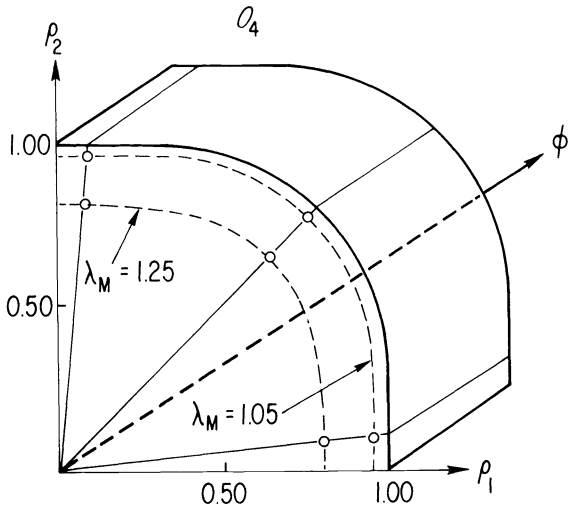


FIG 6. *The domain $\mathcal{O}_4$ of nondegenerate systems with $M = 4$. The points in the (ρ_1, ρ_2) plane are used as operating points in Table 1(b).*

TABLE 1(a)
Average queue lengths of $\mathcal{S}_2$ in various regimes (see Fig. 6).

$M=2$
$\phi=0.1$

λ_M	ρ_1/ρ_2	α_1	α_2	β_2	$E\{X^{(1)}\}$	$E\{X^{(2)}\}$	$E\{X^{(3)}\}$
1.05	10	1.06	9.54	95.36	1.764	.103	18.214
	1	1.57	14.17	14.17	.941	.941	18.926
	0.1	10.60	95.36	9.54	.103	1.764	20.168
1.25	10	1.26	11.35	113.52	1.379	.085	2.603
	1	1.88	16.87	.758	.758	2.778	
	0.1	12.61	113.52	11.35	.085	1.379	2.882

$\phi=0.5$

λ_M	ρ_1/ρ_2	α_1	α_2	β_2	$E\{X^{(1)}\}$	$E\{X^{(2)}\}$	$E\{X^{(3)}\}$
1.05	10	1.06	1.06	10.60	1.764	.103	18.230
	1	1.57	1.57	1.57	.940	.941	23.454
	0.1	10.60	10.60	1.06	.103	1.765	35.814
1.25	10	1.26	1.26	12.61	1.379	.085	2.606
	1	1.88	1.88	1.88	.757	.759	3.440
	0.1	12.61	12.61	1.26	0.85	1.381	5.116

$\phi=0.9$

λ_M	ρ_1/ρ_2	α_1	α_2	β_2	$E\{X^{(1)}\}$	$E\{X^{(2)}\}$	$E\{X^{(3)}\}$
1.05	10	1.06	.12	1.18	1.764	.103	18.370
	1	1.57	.18	.18	.939	.942	64.263
	0.1	10.60	1.18	.12	.102	1.766	176.650
1.25	10	1.26	.14	1.40	1.379	.085	2.625
	1	1.88	.21	.21	.752	.762	9.420
	0.1	12.61	1.40	.14	.085	1.385	25.210

3.2. The degenerate system $\phi=0$. In this section, we consider the case of shared processors, i.e., the limit $\phi\to 0$ such that ρ_1 and ρ_2 are held constant. Since

$$\mu_1=1/(\phi\rho_1) \tag{46}$$

and

$$\mu_2=(1-\phi)/(\mu_2\phi), \tag{47}$$

$\phi\to 0$ implies $\hat{\mu}_1\to\infty$ and $\mu_2\to\infty$. Substituting $\alpha_1=1/\rho_1$, $\alpha_2=(1-\phi)/(\rho_1\phi)$, $\beta_1=0$ and $\beta_2=(1-\phi)/(\rho_2\phi)$ into (26) and taking the limit $\phi\to 0$ yields

$$\mathcal{D}(z,w)=\frac{1-w/\rho_2}{1-(1-\rho_1/\rho_2)zw+(\rho_1/\rho_2)z^2w^2}. \tag{48}$$

The denominator of (48) has the roots $r_+(z)=\rho_2/(\rho_1 z)$ and $r_-(z)=1/z$. The polynomials $D_l(z)$ follow from a partial fraction expansion of (48) as follows:

$$D_l(z)=g_l z^{l-1}(z-\lambda_l) \tag{49}$$

TABLE 1(b)
Average queue lengths of $\mathcal{S}_4$ in various regimes (see Fig. 6).

$M=4$
$\phi=0.1$

λ_M	ρ_1/ρ_2	α_1	α_2	β_2	$E\{X^{(1)}\}$	$E\{X^{(2)}\}$	$E\{X^{(3)}\}$
1.05	10	1.05	9.45	94.51	3.459	.105	16.540
	1	1.31	11.81	11.81	1.833	1.833	17.655
	0.1	10.50	94.51	9.45	.105	3.459	18.377
1.25	10	1.25	11.25	112.51	2.325	.087	1.675
	1	1.56	14.06	14.06	1.351	1.352	1.981
	0.1	12.50	112.51	11.25	.087	2.325	1.860

$\phi=0.5$

λ_M	ρ_1/ρ_2	α_1	α_2	β_2	$E\{X^{(1)}\}$	$E\{X^{(2)}\}$	$E\{X^{(3)}\}$
1.05	10	1.05	1.05	10.50	3.459	.105	16.541
	1	1.31	1.31	1.31	1.830	1.835	22.197
	0.1	10.50	10.50	11.05	.105	3.460	33.067
1.25	10	1.25	1.25	12.50	2.325	.087	1.675
	1	1.56	1.56	1.56	1.346	1.356	2.484
	0.1	12.50	12.50	1.25	.087	2.327	3.344

$\phi=0.9$

λ_M	ρ_1/ρ_2	α_1	α_2	β_2	$E\{X^{(1)}\}$	$E\{X^{(2)}\}$	$E\{X^{(3)}\}$
1.05	10	1.05	.12	1.17	3.459	.105	16.542
	1	1.31	.15	.15	1.823	1.840	63.222
	0.1	10.50	1.17	.12	.105	3.462	165.268
1.25	10	1.25	.14	1.39	2.325	.087	1.675
	1	1.56	.17	.17	1.330	1.367	7.033
	0.1	12.50	1.39	.14	.086	2.331	16.674

where λ_l is given by (39) and

$$g_l=\frac{1-(\rho_1/\rho_2)^{l+1}}{1-(\rho_1/\rho_2)}. \tag{50}$$

It is interesting to note that g_l is the normalization constant for the closed system of Fig. 5 with l jobs. The degree of $D_l(z)$ as given by (49) is only M instead of $2M$. A careful study of the limiting process (which we omit) reveals that as $\phi\to 0$, $\eta_{M,i}\to\infty$ $(2\leqq i\leqq M+1)$. Thus we have the roots $\zeta_{M,i}=0$ $(1\leqq i\leqq M-1)$, $\eta_{M,1}=\lambda_M$ and $\eta_{M,i}=\infty$ $(2\leqq i\leqq M+1)$. Since now $\zeta=0$ is a root of multiplicity $l-1$, the system (35) does not provide a sufficient number of equations. We augment (35) by

$$\sum_{l=0}^{M-1} p_{0,l}D_l^{(k)}(0)=0, \qquad k=1,2,\cdots,M-2, \tag{51}$$

where $D_l^{(k)}(z)$ denotes the kth derivative with respect to z. The system of linear equations (35) and (51) has only two nonzero diagonals. Thus it can be solved in closed form yielding

$$p_{0,j}=\rho_2^j, \qquad 0\leqq j\leqq M. \tag{52}$$

Now we can obtain $\{p_{i,j}\}$ from (17) and (22) by a straightforward (but tedious) computation. There results

$$p_{i,j} = \begin{cases} g^{-1}\rho_1^i\rho_2^j & \text{if } i+j \leqq M, \\ g^{-1}\rho_1^{M-j}\rho_2^j\lambda_M^{M-(i+j)} & \text{if } i+j > M \end{cases} \tag{53}$$

where g is the normalization constant. We shall give an interpretation of (53) in § 3.4. Mean queue sizes for the shared processors case are given in Table 2 for the same parameter values as in Table 1.

TABLE 2

Average queue lengths of $\mathcal{S}_2$ and $\mathcal{S}_4$ in the limit of shared processors (i.e., $\phi \to 0$ such that ρ_1 and ρ_2 remain constant).

$M = 2$
$\phi \to 0$

λ_M	ρ_1/ρ_2	ρ_1	ρ_2	$E\{X^{(1)}\}$	$E\{X^{(2)}\}$	$E\{X^{(3)}\}$
1.05	10	.94	.09	1.764	.103	18.212
	1	.63	.63	.941	.941	18.359
	0.1	.09	.94	1.03	1.764	18.212
1.25	10	.79	.08	1.379	.085	2.603
	1	.53	.53	.758	.758	2.695
	0.1	.08	.79	.085	1.379	2.603

$M = 4$
$\phi \to 0$

λ_M	ρ_1/ρ_2	ρ_1	ρ_2	$E\{X^{(1)}\}$	$E\{X^{(2)}\}$	$E\{X^{(3)}\}$
1.05	10	.95	.10	3.459	.105	16.540
	1	.76	.76	1.833	1.833	17.086
	0.1	.10	.95	.105	3.459	16.540
1.25	10	.80	.08	2.325	.087	1.675
	1	.64	.64	1.352	1.352	1.917
	0.1	.08	.80	.087	2.325	1.675

3.3. The degenerate system $\phi = 1$. The opposite of shared processors is the degenerate system, defined by the limit $\phi \to 1$ such that ρ_1 and ρ_2 are constant. Since $\mu_2 = (1-\phi)/(\rho_2\phi)$, $\phi \to 1$ requires $\mu_2 \to 0$. The limits of $D_1(z)$, $b(z)$ and $c(z)$ are found as follows

$$D_1(z) \to \frac{\rho_1}{\varepsilon}[\rho_1^{-1} + (1+\rho_1^{-1})z - z^2], \tag{54}$$

$$b(z) \to \rho_1^{-1} + (1+\rho_1^{-1})z - z^2, \tag{55}$$

$$c(z) \to 0 \tag{56}$$

where $\varepsilon = 1-\phi$. The recursion (23) yields

$$D_l(z) \to \left(\frac{\rho_1}{\varepsilon}\right)^l [\rho_1^{-1} + (1+\rho_1^{-1})z - z^2]^l. \tag{57}$$

From (57) we find the roots of the degenerate system $\phi = 1$ as follows: $\zeta_{M,i} = 1$ $(1 \leqq i \leqq M+1)$, $\eta_{M,1} = 1$, $\eta_{M,i} = \rho_1^{-1}$ $(2 \leqq i \leqq M+1)$. Since $\eta_{M,1} = 1$, this system is always unstable.

3.4 Hierarchical decomposition, an interpretation of the shared processors case. Define the system population $K = X^{(1)} + X^{(2)} + X^{(3)}$. From (53) we find

$$\Pr\{K = k\} = \begin{cases} g^{-1} g_k \rho_2^k & \text{if } k \leqq M, \\ g^{-1} g_M \rho_2^M \lambda_M^{M-k} & \text{if } k > M. \end{cases} \tag{58}$$

Since $\rho_2 \lambda_l = g_{l-1}/g_l$, we may rewrite (58) as follows

$$\Pr\{K = k\} = \begin{cases} g^{-1} \prod_{l=1}^{k} \dfrac{1}{\lambda_l} & \text{if } k \leqq M, \\ g^{-1} \prod_{l=1}^{M} \dfrac{1}{\lambda_l} \dfrac{1}{\lambda_M} M - k & \text{if } k > M. \end{cases} \tag{59}$$

But (59) is the solution of a $M/M/1$ system with queue-dependent rates $\mu(k)$, which are defined by

$$\mu(k) = \begin{cases} \lambda_k & \text{if } k \leqq M, \\ \lambda_M & \text{if } k > M. \end{cases} \tag{60}$$

Thus for the degenerate system $\mathcal{S}_M$, $\phi = 0$ (shared processors) we find the distribution of the system population identical to the queue size distribution of an equivalent $M/M/1$ system, whose rate function is obtained from the closed system of Fig. 5. The decomposition of $\mathcal{S}_M$ into an open "outer model" (the equivalent $M/M/1$ queue) and a closed "inner model" is shown schematically in Fig. 7. It is easy to obtain the

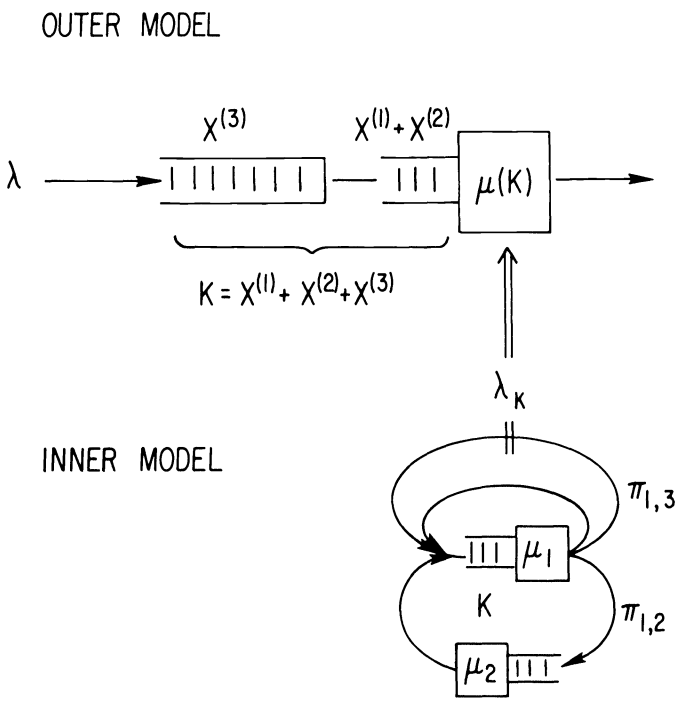


FIG. 7. *Schematic representation of a hierarchical decomposition of* $\mathcal{S}_M$.

marginal distribution $\Pr\{X^{(3)}=i\}$ from (59). The marginal distributions $\Pr\{X^{(1)}=i\}$ and $\Pr\{X^{(2)}=j\}$ are found as a weighted sum of the M closed system solutions.

The decomposition of a queuing system with blocking into an open outer model and a closed inner model is frequently used in practice to get approximate solutions for more complicated networks with blocking [6], [7]. For the example of $\mathcal{S}_M$, we have now given the precise condition under which such approximations are valid, namely that ϕ be small or in other words, that the average number of cycles be large. The accuracy can be estimated from the graphs $E\{X^{(3)}\}$ vs. ϕ which are given in Fig. 8.

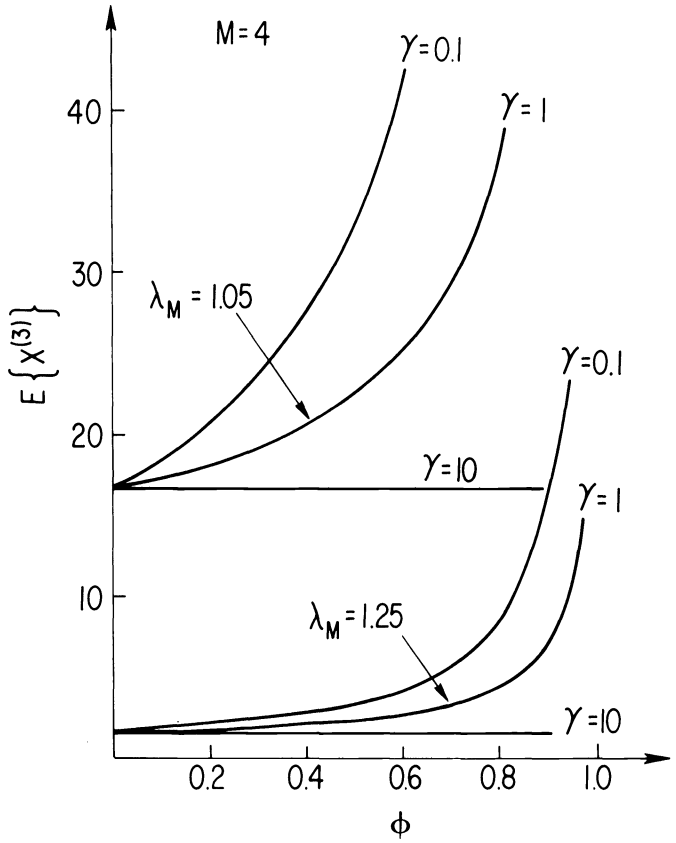


FIG. 8. *Average queue size of the buffer vs. ϕ for $\mathcal{S}_M$. The value for $\phi=0$ is the one obtained by a hierarchical model.*

4. Conclusions. We have given an analytical formulation of the blocking problem in terms of generating functions. This technique is applicable to other blocking systems. For example, the rate of the servers may be made a function of their local queue size. Other blocking mechanisms, too, can be considered such as buffers with resume levels, i.e., service resumes only after the population in the finite capacity subsystem has shrunk below a resume threshold N ($N \leqq M$). The generation function method, however, becomes impractical if the dimensionality of the limited capacity systems goes beyond two (e.g., two I/O servers).

An alternate method is of course a numerical solution of the system of linear equations (1). A suitable truncation of the $Y^{(2)}$ dimension is required. Such a system may be interpreted either as a loss system of a finite source system. We have obtained such numerical solutions by standard elimination methods for block tri-diagonal

systems [9]. Such finite systems do exhibit similar behavior especially as the sensitivity to the parameter ϕ is concerned. Our results will be discussed in detail in a different paper.

We wish to comment, however, that it was the analytical solution which yielded considerably more insight. The so-gained intuition has led us to pose the appropriate questions to a numerical model. It is very unlikely that we would have found the fundamental influence of ϕ by purely empirical methods.

Acknowledgment. The authors are grateful to the help of J. H. Griesmer and R. D. Jenks with the SCRATCHPAD system. Symbolic computations have helped at various places.

REFERENCES

[1] A. G. KONHEIM AND M. REISER, *A queuing model with finite waiting room and blocking*, J. Assoc. Comput. Mach., 23 (1976), p. 328.

[2] M. F. NEUTS, *Two queues in series with a finite intermediate waiting room*, J. Appl. Probability, 5 (1968), p. 123.

[3] B. AVI-ITZHAK AND M. YADIN, *A sequence of two servers with no intermediate queue*, Management Sci., 11 (1965), p. 553.

[4] J. H. GRIESMER AND R. D. JENKS, *Experience with an online symbolic mathematics system*, Proc. of the ONLINE72 Conf., Brunel Univ., Oxbridge, Middlesex, England, 1972; also IBM Rep. RC3925, *The SCRATCHPAD System*, Thomas J. Watson Res. Center, Yorktown Heights, NY.

[5] S. S. LAVENBERG, *Stability and maximum departure rate of certain open queuing networks having finite capacity constraints*, IBM Rep. RJ1625, Thomas J. Watson Res. Center, Yorktown Heights, NY, 1975.

[6] P. J. COURTOIS, *Decomposability, instabilities and saturation in multiprogramming systems*, Comm. ACM, 18 (1975), p. 371.

[7] ———, *Error analysis in nearly-completely decomposable stochastic systems*, Econometrica, 43 (1975), p. 691.

[8] J. P. BUZEN, *Queuing network models for multiprogramming*, Ph.D. dissertation, Harvard Univ., Cambridge, MA, 1971.

[9] R. S. VARGA, *Matrix Iterative Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1962.

SIAM J. COMPUTING
Vol. 7, No. 2, May 1978

PARALLEL COMPUTATIONS IN GRAPH THEORY*

ESHRAT REGHBATI (ARJOMANDI)† AND D. G. CORNEIL‡

Abstract. In parallel computation two approaches are common, namely unbounded parallelism and bounded parallelism. In this paper both approaches will be considered with respect to graph theoretical algorithms. The problem of unbounded parallelism is studied in § 2 where some lower and upper bounds on different graph properties for directed and undirected graphs are presented. In § 3 we mention bounded parallelism and three different K-parallel graph search techniques, namely K-depth search, breadth-depth search, and breadth-first search. Each parallel algorithm is analyzed with respect to the optimal serial algorithm. It is shown that for sufficiently dense graphs the parallel breadth-first search technique is very close to the optimal bound.

Keywords. parallel computation, graph algorithms, graph connectivity, depth-first search, breadth-first search, breadth-depth search, dense graphs

1. Introduction. The problem of parallel computation has been considered from two different points of view. One is to permit some fixed number, say K, of processors to be available. This notion is called bounded parallelism, K-parallelism or K-computation. For a given problem assume that the current best time bound for sequential computation is T steps. Our goal then, is to have a T/K step K-parallel algorithm. Trivially, any K-parallel algorithm which requires fewer than T/K steps yields an improved serial algorithm. The second case, which is referred to as unbounded parallelism, involves an arbitrarily large number of processors being available. In this paper we will be looking at both approaches. Much work in producing and analyzing parallel algorithms has been done in different areas, such as matrix calculations [7], [14], sorting [3], evaluation of polynomials [13] and arithmetic expressions [6], etc. Definitions not given in this paper are standard and may be found in Harary [8]. Throughout the paper, K refers to the number of processors, n refers to the number of nodes of a graph, and all logarithms are to base 2.

In analyzing parallel algorithms it is necessary to clearly define the model of computation. Although many parallel computers (such as the CDC Star 100, Bull Gamma 60, and Illiac IV) could be used as the model of computation, for many problems a more flexible and powerful theoretical model is essential. The model used in this paper for unbounded parallelism is as follows:

The model consists of sufficiently many identical processors and a sufficiently large memory with unrestricted access. Each processor is capable of performing Boolean and comparison operations in unit time. This unit time is called a step. It is assumed that the input data is stored in the memory before the computation starts. Each processor takes its operands from the memory and after a step stores the result in the memory. At any unit of time only one processor may change the contents of any one memory location. The parallel complexity of the computation is the number of steps used to produce the output in the memory.

The above model is very similar to that used by other researchers [7]. The model used for bounded parallelism is a slight modification of the above model. Namely, it is assumed that K identical processors, each being capable of performing arithmetic operations and comparisons, are available.

* Received by the editors December 11, 1975, and in revised form June 10, 1977. This work was supported by the National Research Council of Canada.

† N620R, Department of Computer Science, York University, Downsview, Ontario M3J 1P3.

‡ Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A7.

In § 2 we prove some lower and upper bounds on the complexity of algorithms for determining different graph properties for directed and undirected graphs. All the lower bounds are $\Omega(\log n)$[1] and upper bounds are $O(\log^2 n)$. The number of processors used for achieving these upper bounds is n^3.

Many different serial graph algorithms employ depth-first search. In § 3 we introduce and analyze various K-parallel graph search techniques. Two cases are distinguished, namely h-sparse and h-dense graphs. By an h-dense graph it is meant that the number of edges in the graph is $\geqq h \cdot n/2$. For each of the K-parallel search techniques, a value $f(K)$ is determined so that for h-dense graphs where $h > f(K)$, the particular K-parallel technique is superior to any serial search technique. A graph is called h-sparse if the number of edges is less than $n \cdot h/2$. It is shown that for sufficiently dense graphs two of these parallel search techniques achieve bounds which are very close to optimal. By optimal we mean that if an optimal sequential algorithm takes T units of time, then our K-parallel algorithm takes $T/K +$(small lower order terms) units of time. In K-parallel search algorithms the key point is to assign processors to the edges emanating from a node rather than to different nodes. Namely when the graph is dense enough, a better bound is achieved by keeping the processors busy looking at the edges. In the case of h-sparse graphs since there aren't enough edges emanating from a node to keep the processors busy we have to start looking at more than one node at a particular time. In [1] we have presented a few pathological cases which show how some techniques for assigning processors to the nodes behave badly. Hence the question of handling h-sparse graphs in the case of bounded parallelism remains unsettled.

2. Some lower and upper bounds for unbounded parallelism. Among all the known graph representation structures, the adjacency matrix seems to be a proper data structure in the case of unbounded parallelism. One reason is that checking the existence of an edge is very fast and does not involve any search; another reason is that because of the large amount of parallelism in matrix manipulations we can view the graph as a Boolean matrix and try to interpret graph problems as matrix manipulation problems. In this section it is assumed that the graph is represented by an adjacency matrix A.

In the following we present some lower bounds for the complexity of algorithms for determining different graph properties. In the proof of Theorem 2, algorithms which determine various connectivity properties are presented. The output from all these algorithms is an integer vector INDEX and a Boolean matrix. The nonzero elements of INDEX are row numbers of the Boolean matrix which represent the connected components.

In proving lower bounds for unbounded parallelism, the following result (referred to as the fan-in theorem) is often used:

THEOREM 1 (Munro and Paterson [13]). *Suppose the computation of a single quantity Q requires $q \geqq 1$ binary arithmetic operations. Then the shortest K-computation of Q is at least*

$$\lceil((q+1)-2^{\lceil \log K \rceil})/K\rceil + \lceil \log K \rceil \text{ steps} \quad \text{if } q \geqq 2^{\lceil \log K \rceil},$$

and

$$\lceil \log (q+1) \rceil \quad \text{otherwise.}$$

[1] As defined by Knuth [12]: $\Omega(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants c and n_0 with $g(n) \geqq c \cdot f(n)$ for all $n \geqq n_0$.

Most efficient serial graph algorithms assume that the graph is represented by a list of adjacencies which is normally in the form of a linked list. It is easily seen that $O(n)$ steps are necessary and sufficient for an unbounded parallel algorithm to transform a (linked) list of adjacencies to an adjacency matrix. To transform an adjacency matrix to a (sequential or linked) list of adjacencies $O(\log n)$ steps are necessary and sufficient (the necessity follows from the fan-in theorem).

Many researchers have studied the problem of determining lower bounds for the serial computation of various graph properties under the assumption that the graph is represented by an adjacency matrix. The following result follows immediately from a result by Kirkpatrick [11]:

COROLLARY 1. *To determine any nontrivial digraph (undirected graph) property, in the worst case, at least* $2n-2$ $(\sqrt{2}\,n-2)$ *entries of the adjacency matrix must be examined.*

From the fan-in theorem one immediately sees that in unbounded parallelism a lower bound of $\lceil \log n \rceil + c$ (c a constant) is obtained for all nontrivial graph properties. For many classes of graph properties (which include connectivity in undirected graphs, weak, unilateral, and strong connectivity in digraphs, etc.) various researchers, namely Kirkpatrick [11], Rivest and Vuillemin [15], and Best, van Emde Boas and Lenstra [5] have shown lower bounds of $\Omega(n^2)$. For these properties, the unbounded parallel lower bound is $\lceil 2 \log (n) \rceil - c'$ (c' a constant).

We now turn our attention to determining upper bounds for some connectivity problems.

THEOREM 2. *The following connectivity problems can be reduced in* $O(\log n)$ *time, using* n^3 *processors, to* c *(a constant) transitive closure computations*:

finding connected components in an undirected graph $(c = 1)$,

finding weakly connected components in a digraph $(c = 1)$,

finding strongly connected components in a digraph $(c = 2)$.

COROLLARY 2.1. *The upper bound on the connectivity problems mentioned in Theorem* 2 *is* $O(\log^2 n)$.

Proof. The result follows since the best known upper bound on the transitive closure computation is $O(\log^2 n)$.

Note that any improvement of the bound for transitive closure computation will improve the bound for the connectivity problems. In [2] we have shown that the number of unilaterally connected components may grow exponentially with the number of nodes. Hence the result of Theorem 2 is not applicable to unilateral connectivity.

Proof of Theorem 2. We present algorithms which demonstrate the reduction of connectivity problems to the transitive closure computation.[2] We do not present the proof of correctness or the details of the timing (see [1] for more details).

ALGORITHM 1. Finding connected components in an undirected graph. The first step of the algorithm finds A^*, the reflexive transitive closure of A. A^* contains all the information about the connected components. The rest of the algorithm extracts the connected components from A^*. Namely it finds the distinct rows of A^* and stores the row numbers in INDEX. Each distinct row represents a connected component.

[2] Recently D. S. Hirschberg [9] has also given an $O(\log^2 n)$ algorithm for finding connected components in an undirected graph using $O(n^2)$ processors.

1. Find A^*, the (reflexive) transitive closure of A.

2. Find the position of the first nonzero entry for each row of the matrix A^* and build vector INDEX as follows:

$$\text{INDEX}(i) = \text{the position of the first nonzero entry in row } i.$$

3. Construct an $n \times n$ matrix, M, which is initially set to zero; now set:

$$M(\text{INDEX}(i), i) = 1 \quad \text{for } i = 1, \cdots, n.$$

The column numbers of the 1's in each row of M correspond to the identical rows of A^*.

4. Repeat step 2 for matrix M. If row i is all zeros set $\text{INDEX}(i) = 0$. Since the row numbers of identical rows in A^* are now the nonzero elements of a row in M, this time the nonzero elements of INDEX will be representatives from each connected component. Namely if $\text{INDEX}(i) = j > 0$, then the 1's in the jth row of the transitive closure matrix represent the nodes in one of the connected components.

Algorithm 1 may be used directly for finding the weakly connected components of a digraph.

ALGORITHM 2. Finding strongly connected components. In this algorithm a matrix $\hat{C}$ is constructed. The connected components of $\hat{C}$ are exactly the strongly connected components of A.

1. Find A^*, the transitive closure of A.

2. Construct the cycle matrix C as follows [4]

$$c_{ij} = a_{ij} \times a^*_{ji}.$$

The digraph represented by C contains all the edges of the digraph represented by A, which belong to a directed cycle.

3. Construct matrix $\hat{C}$ as follows:

$$\hat{c}_{ij} = c_{ij} \vee c_{ji}.$$

4. Find the connected components of $\hat{C}$, by using Algorithm 1. The connected components of $\hat{C}$ are exactly the strongly connected components of A.

3. Bounded search techniques. A very powerful technique widely used in serial graph algorithms is depth-first search [16]. In this pattern of search, each time an edge to a new vertex is discovered, the search is continued from the new vertex and is not renewed at the old vertex until all edges from the new vertex are exhausted. This invokes an ordering on the edges which is destroyed if we allow more than one node at a time to be searched or more than one edge from a node to be looked at. Thus depth-first search seems to be inherently serial.

Suppose we are given K processors and a graph problem such as finding the connected components or growing a spanning tree (forest) in a graph. How fast can we solve these problems? In this section we present some bounded search techniques for searching a graph. Throughout this section, it is assumed that the graph is represented by a list of adjacencies (each list uses sequential storage allocation). For analyzing our algorithms some operations are considered as active and we will be concerned only with those operations. Serial and K-parallel search techniques presented in this paper are analyzed and compared by means of counting active operations. One visit to a node will be considered as one active operation. When a node is selected for search, each processor looks at one edge emanating from that node. Each processor determines whether the node it is looking at is a new node, and forms a partial list of all new

nodes which it has found. At various stages during the K-parallel search, the partial lists found by each processor are linked together and added to the list of new nodes. This linking operation is also treated as active. Note that in both serial and parallel computation, the operations involved in the construction of the (partial) list of new nodes kept by each processor and the selection of new nodes for search are not considered to be active. If we had considered these operations to be active, then a lower order term of $c \cdot n$, $c \leqq 2$, would be added to the time. If we count only the active operations, then the serial upper bound on all considered search techniques, using an adjacency list as the graph representation would be $T_1 = \sum_{i=1}^{n} (\deg_i + 1)$. This time bound is optimal. As mentioned previously the best possible bound for a K-parallel algorithm is T_1/K.

In this section we introduce three different graph search techniques, namely K-depth search, breadth-depth search, and breadth-first search. It is shown that for sufficiently dense graphs the breadth-first search algorithm achieves a bound which is very close to optimal.

(a) *K-depth search.* In this technique once a node is picked we simultaneously look at K edges emanating from that node. Then one of the most recently discovered nodes is selected for search and the search continues from there. Namely, we are doing a depth-first search with breadth K at each node. After a node has been looked at for the first time, the corresponding entry of a vector which is initially set to zero is changed to one. Suppose each processor determines whether the node it is interrogating is a new node. We then add the new nodes found at this step to a LIFO list for later search. A node is not deleted from this list until all its adjacencies are exhausted.

Depth-first search has proved to be a very powerful technique in serial computations. Namely for many problems we would like to traverse the graph as deeply as possible by going to a new node each time. Once we have a K-parallel model, it seems natural to use a K-depth search technique. This technique seems more attractive for algorithms which employ depth-first search, though we have not yet found a practical application for it.

Now let us see how fast we can add the new nodes to the list of unexamined nodes. After K edges emanating from a node have been searched, some of the processors have found new nodes. It is obvious that the new nodes found at each step can be added to the list of new nodes in $\lceil \log K \rceil + 1$ steps. The time bound of this search technique if we count only the active operations is as follows:

$$T_k^1 = \sum_{i=1}^{n} \left\lceil \frac{\deg_i + 1}{K} \right\rceil (\lceil \log K \rceil + 1)$$

where the term $(\lceil \log K \rceil + 1)$ is the time spent for adding the new nodes found adjacent to a node being visited, and the term $\lceil (\deg_i + 1)/K \rceil$ is the number of times we visit node i. Hence:

$$T_k^1 \leqq T_1 \frac{\lceil \log K \rceil + 1}{K} + n \cdot (\lceil \log K \rceil + 1). \tag{1}$$

Thus the reduction factor is $(\lceil \log K \rceil + 1)/K$ with an additive term of $n \cdot (\lceil \log K \rceil + 1)$. Note that in this search technique, in order for $(\lceil \log K \rceil + 1)/K$ to be less than 1, K must be $\geqq 4$. This search technique is superior to the serial search techniques for h-dense graphs where $h \geqq \lceil \log K \rceil + 17$. The superiority increases as h increases.

Figure 1 represents graph G and a K-depth search with $K = 2$ performed on it.

(b) *Breadth-depth search.* In part (a) of this section we showed that by using a K-depth search technique, we can reduce the serial time bound by a factor of

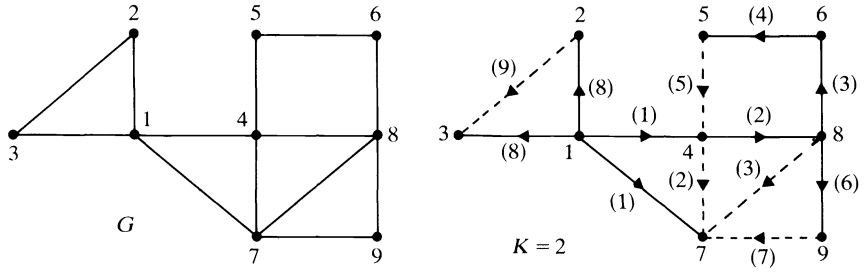


FIG. 1. *K-depth search on G. Edges are numbered in the order in which they are scanned. (Note that the edges that are scanned simultaneously are given the same number.)*

$(\lceil \log K \rceil + 1)/K$. In this section we show that by using a breadth-depth search technique, a reduction factor of $1/K$ can be achieved. In a breadth-depth search technique once we are at a node, we look at all its adjacencies. Then one of the most recently discovered nodes is selected and the search is continued from there. While a node is being searched, each processor keeps track of the new nodes it has found. These partial lists are linked together and added to the list of new nodes when the list of adjacencies for the node currently being searched is exhausted. Counting the active operations we get the following time bound:

$$T_k^2 = \sum_{i=1}^{n} \left(\left\lceil \frac{\deg_i}{K} \right\rceil + 1 + \lceil \log K \rceil + 1 \right)$$

where $\lceil \log K \rceil + 1$ is the time spent on node i for keeping track of the new nodes discovered adjacent to it and $\lceil \deg_i/K \rceil + 1$ is the time spent on node i, once it is selected for search. Simplifying T_k^2 we get the following:

$$T_k^2 \leqq T_1/K + n \cdot (\lceil \log K \rceil + 3). \tag{2}$$

Now the reduction factor is $1/K$ with the additive term of $n \cdot (\lceil \log K \rceil + 3)$. The K-parallel breadth-depth search is superior to the serial search techniques for h-dense graphs where $h \geqq \lceil \log K \rceil + 7$. As before the superiority increases as h increases.

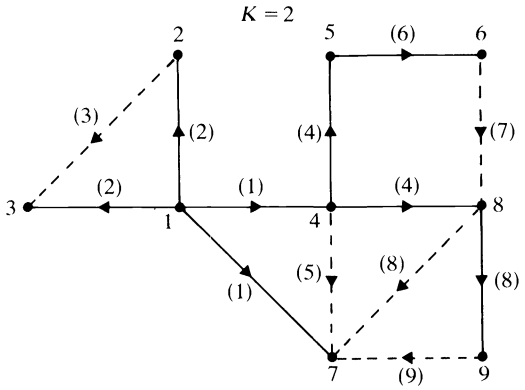


FIG. 2. *Breadth-depth search on the graph of Fig. 1. Edges are numbered in the order in which they are scanned. (Notice that the edges that are scanned simultaneously are given the same number.)*

Figure 2 represents a breadth-depth search performed on the graph of Fig. 1.

Sequential breadth-depth search is widely used in shortest path algorithms (see [10]) and PERT networks. In [1] we have shown, using the above search technique,

that a bound similar to relation (2) above can be obtained for many shortest path and PERT network problems.

(c) *Breadth-first search.* We now show that by using a breadth-first search technique, the additive term in relation (2) can be reduced further. In a breadth-first pattern of search we start from a node and all the processors are employed for exhausting its adjacencies. A node at distance i from the start node is searched before the nodes further from the start node. When the nodes at distance i from the start node are being searched, each processor keeps track of new nodes at distance $i+1$ which it has found. After all the nodes at distance i have been searched, we join the partial lists made by the different processors.

The reason for linking partial lists together is to prevent wasting time when we want to pick a new node for search. Namely, if only a few lists are nonempty then we are wasting time by looking at all the lists in order to find a nonempty one. Now let us ascertain the complexity of this search:

$$T_k^3 = \sum_{i=1}^{n} \left(\left\lceil \frac{\deg_i}{K} \right\rceil + 1 \right) + (\text{the distance of the furthest node from the start node}) \cdot \lceil \log K \rceil$$

where $\lceil \deg_i/K \rceil + 1$ is the time spent on each node, once it is selected for search and $\lceil \log K \rceil$ is the time spent for linking the partial lists of new nodes found by each processor at every distance from the start node. Hence:

$$T_k^3 \leqq T_1/K + L \cdot \lceil \log K \rceil + 2n \tag{3}$$

where L is the distance of the furthest node from the start node. If our graph is sufficiently dense L is very small; in fact, it is not hard to prove that if the degree of each node in our graph is $\geqq K$, then $L \leqq 3 \cdot q - 1$, where $n = q(K+1)+1$ and $q \geqq 1$. (See [1] for a proof.) Hence for sufficiently dense graphs, L is at worst of the order of n/K. Thus in this pattern of search the reduction factor is $1/K$ and the additive term is $L \cdot \lceil \log K \rceil + 2n$. The K-parallel breadth-first search is superior to the serial search techniques for h-dense graphs where $h \geqq \lceil \log K \rceil + 5$. Once again the superiority increases as h increases.

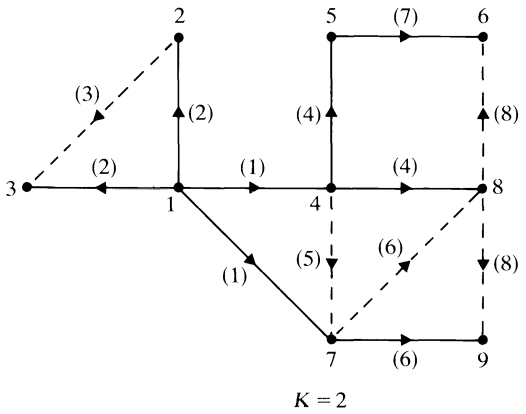


FIG. 3. *Breadth-first search on graph of Fig.* 1. *Edges are numbered in the order in which they are scanned. (Notice that the edges that are scanned simultaneously are given the same number.)*

Figure 3 represents a breadth-first search performed on the graph of Fig. 1.

The breadth-first search technique introduced above can easily be used to find the connected components of an undirected graph and the arc shortest paths in a uniform

positive arc weight network [10]. The time bound of the algorithms for solving these problems is very close to the bound given in relation (3) above. (For more applications of the search techniques introduced in this section see [1].)

REFERENCES

[1] E. ARJOMANDI, *A study of parallelism in graph theory*, Ph.D. thesis, TR 86, Dept. of Computer Science, Univ. of Toronto, 1975.

[2] ———, *On finding all unilaterally connected components*, Information Processing Lett., 5 (May 1976), no. 1, pp. 8–10.

[3] K. E. BATCHER, *Sorting networks and their applications*, Spring Joint Computer Conf., AFIPS Proc., vol. 32, (1968), pp. 307–314.

[4] A. T. Berztiss, *Data Structures: Theory and Practice*, Academic Press, New York 1971.

[5] M. R. BEST, P. VAN EMDE BOAS AND H. W. LENSTRA, JR., *A sharpened version of the Aanderaa–Roseberg conjecture*, TR ZW 30–74, Mathematisch Centrum, Amsterdam, 1974.

[6] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[7] L. CSANKY, *Fast parallel matrix inversion algorithms*, Proceedings of 16th Annual IEEE Symposium on Foundations of Computer Science, Oct. 13–15, IEEE, New York, 1975, pp. 11–12.

[8] F. HARARY, *Graph Theory*, Addison Wesley, Reading, MA, 1971.

[9] D. S. HIRSCHBERG, *Parallel algorithms for the transitive closure and the connected components*, Proceedings of the Eighth Annual ACM Symposium on Theory of Computing, May 3–5, ACM, New York, 1976, pp. 55–57.

[10] D. B. JOHNSON, *Algorithms for shortest paths*, Ph.D. thesis, TR 73–169, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1973.

[11] D. G. KIRKPATRICK, *Determining graph properties from matrix representations*, Proceedings of Sixth Annual ACM Symposium on Theory of Computing, Seattle, Washington, April 30–May 2, ACM, New York, 1974, pp. 84–90.

[12] D. E. KNUTH, *Big omicron and big omega and big theta*, ACM SIGACT NEWS, (April–June, 1976), no. 2, pp. 18–24.

[13] I. MUNRO AND M. PATERSON, *Optimal algorithms for parallel polynomial evaluation*, J. Comput. System Sci., 7 (1973), pp. 189–198.

[14] Y. MURAKA AND D. J. KUCK, *On the time required for a sequence of matrix products*, Comm. ACM, 16 (1973), pp. 22–26.

[15] R. L. RIVEST AND J. VUILLEMIN, *On recognizing graph properties from adjacency matrices*, Theor. Comput. Sci., 3 (December 1976), no. 3, pp. 371–384.

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

SIAM J. COMPUTING
Vol. 7, No. 2, May 1978

ERRATA: ON THE NUMBER OF MULTIPLICATIONS REQUIRED FOR MATRIX MULTIPLICATION*

R. W. BROCKETT† AND D. DOBKIN‡

In the formula marked (*) on page 625 the two terms involving summation should be subtracted rather than added. In the equation which immediately precedes this the two terms in parentheses should be added rather than subtracted.

We are indebted to Dr. Paul Purdom for pointing out this correction.

* This Journal, 5(1976), pp. 624–628. Received by the editors August 24, 1977.
† Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts 02138.
‡ Department of Computer Science, Yale University, New Haven, Connecticut 06520.

SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0001 $01.00/0

DATA MOVEMENT IN ODD-EVEN MERGING*

ROBERT SEDGEWICK†

Abstract. A complete analysis is given of the number of exchanges used by the well-known Batcher's odd-even merging (and sorting) networks. Batcher's method involves a fixed sequence of "compare-exchange" operations, so the number of comparisons required is easy to compute, but the problem of determining how many comparisons result in exchanges has not been successfully attacked before. New results are derived in this paper giving accurate formulas for the worst-case and average values of this quantity.

The worst-case analysis leads to the unexpected result that, asymptotically, the ratio of exchanges to comparisons approaches 1, although convergence to this asymptotic maximum is very slow.

The average-case analysis shows that, asymptotically, only $\frac{1}{4}$ of the comparators are involved in exchanges. The method used to derive this result can in principle be used to get any asymptotic accuracy. The derivation involves principles of the theory of complex functions; in particular, properties of the Γ-function and the generalized Riemann ζ-function are integral to the solution. Intermediate results in the analysis may be applicable to the average-case analysis of other merging methods, and the final portion of the derivation illustrates the utility of the "gamma function" method of asymptotic analysis.

Key words. analysis of algorithms, odd-even merge, merging networks, merge-exchange sort, sorting networks, gamma function, zeta function

1. Introduction. Suppose that we have two sorted arrays $B[1], \cdots, B[N]$ and $C[1], \cdots, C[N]$ which we wish to merge into a single sorted array $A[1], \cdots, A[2N]$, The straightforward algorithm

$i := j := 1$; $B[N+1] := C[N+1] := \infty$;
loop for $k := 1, 2, \cdots, 2N$:
    if $B[i] < C[j]$ **then** $A[k] := B[i]$; $i := i+1$
        else $A[k] := C[j]$; $j := j+1$
repeat

has been shown to be the "best possible" way to solve this problem (see [13, p. 199]) in that it requires the minimum number of comparisons between keys, not counting the ∞ sentinel keys. However, this method may not be appropriate if, for example, we wish to build hardware to do the merging, since it requires space for the output array and its comparison sequence depends on the arrangement of the input.

The "odd-even" merge introduced by K. E. Batcher in 1964 [3], [4] is a well-known method for merging in place with a fixed comparison sequence. To satisfy the in place condition we assume that the first sorted input array is stored in the odd positions $A[1], A[3], \cdots, A[2N-1]$ of the output array, and the second sorted input array is stored in the even positions $A[2], A[4], \cdots, A[2N]$ of the output array. Such files are called 2-*ordered*, and merging is equivalent to sorting 2-ordered files. Then

* Received by the editors March 7, 1977, and in revised form October 3, 1977.

† Computer Science Program and Division of Applied Mathematics, Brown University, Providence, Rhode Island 02912. This work was supported by the National Science Foundation under Grant MC575-23738.

Batcher's method may be implemented as follows:

```
loop for j := 1, 2, ···, N:
    if A[2j−1] > A[2j] then A[2j−1] := A[2j];
repeat;
loop for δ := 2^⌈lgN⌉−1, 2^⌈lgN⌉−2, ···, 1:
    loop for j := 1, 2, ···, N−δ:
        if A[2j] > A[2j+2δ−1] then A[2j] :=: A[2j+2δ−1];
    repeat;
repeat;
```

In this program, notice that the only statements which actually operate on the data are the "compare-exchange" statements of the form

$$\textbf{if } A[2j] > A[2j+2\delta-1] \textbf{ then } A[2j] :=: A[2j+2\delta-1];$$

and these are performed in the same order regardless of the input. Because of this, it is convenient to describe the algorithm as a merging *network* as in Fig. 1, which shows the algorithm operating on a typical 2-ordered file of sixteen numbers. The numbers move from left to right, encountering "compare-exchange" modules on the way. Each module exchanges its inputs, if necessary, to make the larger number appear on the lower line after passing. (Modules which actually perform exchanges are boxed in Fig. 1.) The networks for $N = 1$, 2, 4, 8, and 16 are shown in Fig. 2. Notice that the networks are composed of stages (an initial stage plus one for each value of δ) within which all of the compare-exchanges can be overlapped. This makes Batcher's algorithm particularly useful when parallelism is available.

Figure 3 shows the networks for $N = 1$, 2, 4, 8 and 16 with the comparators arranged somewhat differently to illustrate why the method is called the "odd-even" merge. First the "odd" members of the input files ($A[1]$, $A[5]$, $A[9]$, $\cdots$ and $A[2]$, $A[6]$, $A[10]$, $\cdots$) are merged, and, independently, the "even" members of the input files ($A[3]$, $A[7]$, $A[11]$, $\cdots$ and $A[4]$, $A[8]$, $A[12]$, $\cdots$) are merged. After this, it turns out that a single stage of compare-exchange modules connecting $A[2]$ with $A[3]$, $A[4]$ with $A[5]$, $A[6]$ with $A[7]$, etc., will complete the sort. Batcher gave a complete inductive proof that his method is valid, using this approach [2] (see also Knuth [13, pp. 224–225]). Knuth gives another proof [13, exercise 5.2.2-10] which we shall examine in some detail below.

To determine the running time of a program, we need to be able to determine the frequency of execution of each of its instructions. In the program above, these

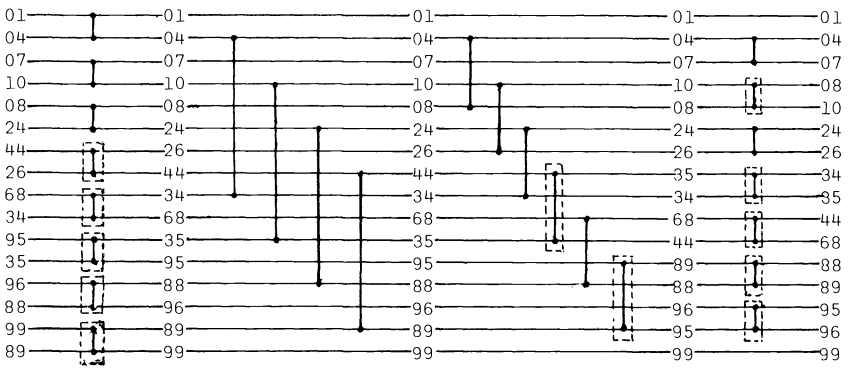


FIG. 1. 2-*sorting a file of* 16 *elements.*

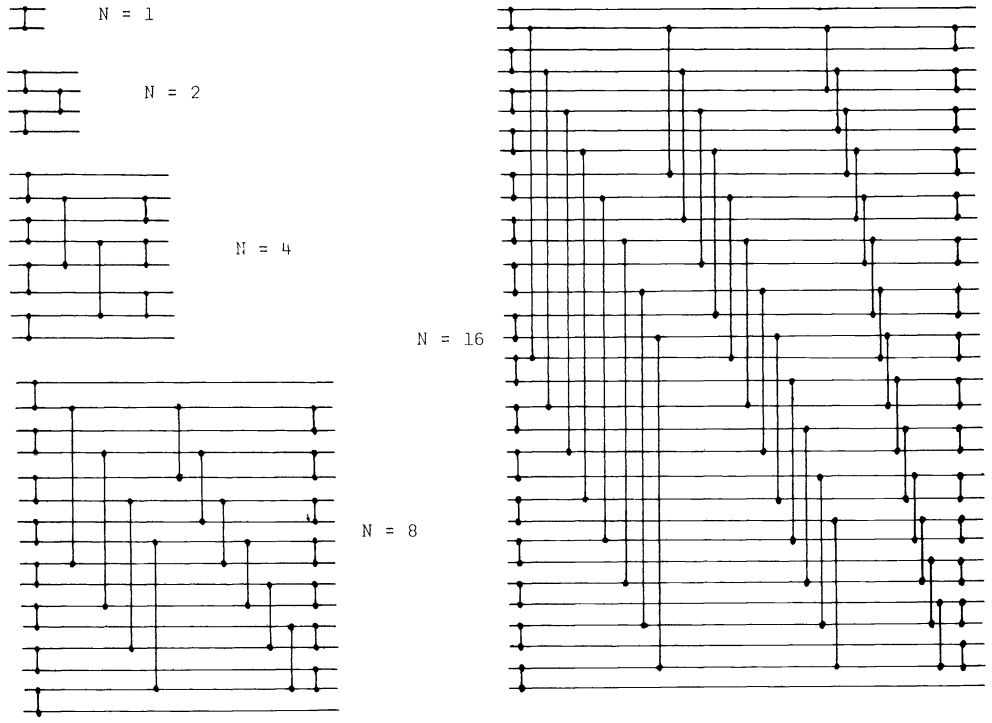


FIG. 2. *Odd-even merging (2-sorting) networks.*

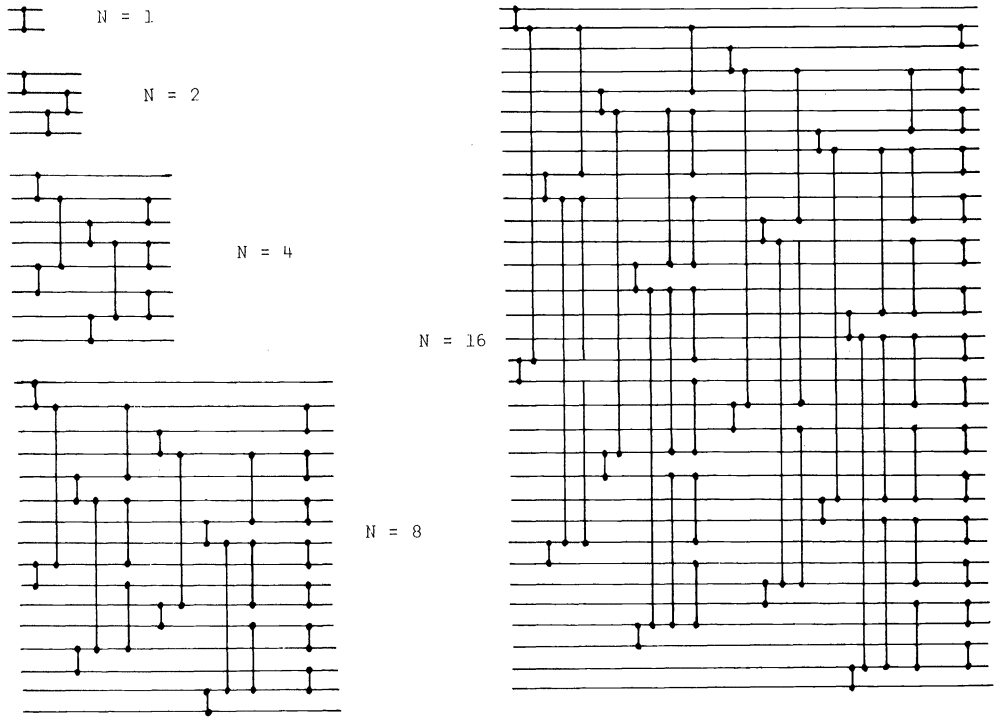


FIG. 3. *Odd-even merging networks (alternate arrangement).*

frequencies can be determined from N, the number of items in each file to be merged, and the following three quantities:

A—the number of stages,

B—the number of exchanges, and

C—the number of comparisons.

More precisely, A is the number of times a new value is assigned to δ, C is the number of times the tests "$A[2j-1] > A[2j]$" and "$A[2j] > A[2j+2\delta-1]$" are performed and B is the number of times these tests are successful.

The values of A and C are clearly independent of the input distribution. They do depend on the number of elements being merged, and we will write A_N and C_N to denote their values for an $N \times N$ merge. On inspection of the program, we see that

$$A_N = \lceil \lg N \rceil + 1 \qquad (\lg N \equiv \log_2 N)$$

and, counting the number of times the loop index j changes, we have

$$C_N = N + \sum_{\lceil \lg N \rceil > k \geq 0} (N - 2^k)$$

which evaluates to

$$C_N = N(\lceil \lg N \rceil + 1) - 2^{\lceil \lg N \rceil} + 1. \tag{1}$$

The number of comparisons is of order $(N \log N)$ so Batcher's merging algorithm cannot compare with the straightforward $O(N)$ algorithm on a serial computer. However, if parallelism is available, the comparisons on each stage can be performed independently, and the merge can be completed in $\lceil \lg N \rceil + 1$ parallel stages. Also, R. W. Floyd [13, p. 230] has shown that any merging method which can be represented as a network must use at least $\frac{1}{2} N \lg N + O(N)$ comparators to 2-sort N elements, so Batcher's method is, in this sense, nearly optimal.

The value of B does depend on the input distribution, and the subject of this paper is the analysis of the maximum and average values of this quantity when a random 2-ordered file is sorted. This is listed as an open problem by Knuth [13, p. 135]. The practical importance of this problem may be limited, since the method is best suited to a parallel implementation, and exchanges might not significantly affect the running time of a truly parallel implementation. However, it is essential to know the value of B for serial implementations, and, as we shall see, the analysis of B is of some theoretical interest. Our understanding of Batcher's method is incomplete without an understanding of how often it does exchanges. Moreover, the methods and results of the analysis may be applicable to the study of other algorithms.

To deal with the number of exchanges, it is useful to examine Knuth's alternate proof that the odd-even merge is valid. The proof is based on a natural correspondence between 2-ordered files of $2N$ elements and paths connecting opposite corners of $N \times N$ lattice diagrams. An example of this correspondence is shown in Fig. 4. Starting at the upper left corner, we form a path whose kth segment goes down if the kth smallest element is an odd position in the file, and to the right if the kth smallest element is in an even position in the file. We shall denote the lattice point reached after i steps down and j steps to the right by (i, j). The path must end up at (N, N) since there are N even positions and N odd positions, and the correspondence is clearly unique. One can think of the final sorted file as a chain with $2N$ links, and the path as the unique arrangement of the chain adhered at the upper left corner with each link vertical if the corresponding element is in an odd position in the file and horizontal if the corresponding element is in an even position in the file.

The sorted array corresponds to the diagonal path through the lattice whose first segment is vertical (the dotted line in Fig. 4), and the merging process consists of transformations from an arbitrary path to that particular path. As mentioned above, Batcher's method can be divided into $\lceil \lg N \rceil + 1$ stages of independent compare-exchange operations. The proof that the odd-even merge is valid consists of showing that the stages correspond to "folding" (interchanging horizontal and vertical) the path about certain diagonals in the lattice diagram.

For example, the first stage, which compare-exchanges $A[2]$ with $A[1]$, then $A[4]$ with $A[3]$, then $A[6]$ with $A[5]$, etc., corresponds to folding the path about the main diagonal. To show this, we first note that any path can be divided into sections which are either "high" (totally above the diagonal) or "low" (on or below the diagonal). (The path in Fig. 4 consists of a low section followed by a high section.) Now, the jth comparison in the first stage results in an exchange if and only if the jth horizontal path segment (which corresponds to $A[2j]$) appears before the jth vertical path segment (which corresponds to $A[2j-1]$). But this can happen if and only if both segments are above the diagonal. Therefore, all elements represented by high path sections are involved in exchanges and no elements represented by low sections are involved in exchanges. After the exchanges, low sections are unchanged, and horizontal and vertical are interchanged in high sections, making them low. In other words, the whole path is reflected down about the diagonal.

The first stage folds down about the main diagonal, ensuring that the path falls below the main diagonal, and successive stages fold up about the diagonal δ units below the main diagonal, ensuring that the path falls in a "band" between that diagonal and the main diagonal. After the stage when $\delta = 1$ the path must coincide with the main diagonal, and the corresponding permutation is sorted. Figure 5 shows the sample 2-ordered permutation in Fig. 4 being sorted. Shaded areas are the areas within which the path is guaranteed to fall, and each stage "folds" the shaded area in the middle. The reader may wish to check the correspondence and the proof by seeing that successive paths in Fig. 5 correspond to successive permutations in Fig. 1. (In particular, note that there are no exchanges on the third stage, and the path is unchanged.)

This proof that Batcher's method is valid also gives us an easy way to count the number of exchanges used to sort any particular 2-ordered permutation. First, we notice that if any segment on the path is on the main diagonal, then the element corresponding to it will not be involved in any exchanges during the sort (since it is in a

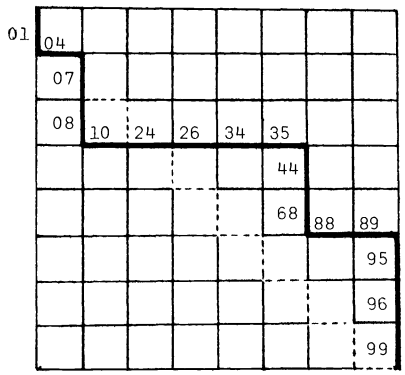


FIG. 4. *Lattice path for 2-ordered permutation of Fig. 1.*

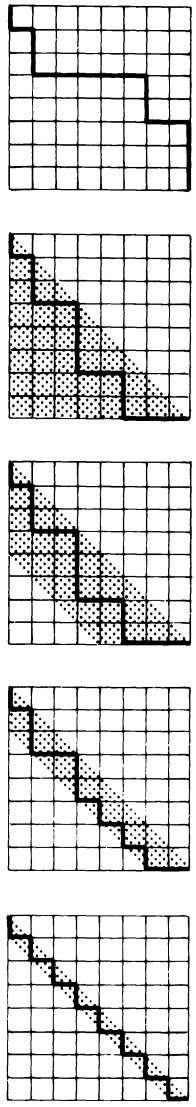


FIG. 5. *Sorting Fig.* 4.

"low" section for the first stage, and a "high" section for successive stages). If any segment on the path is on the diagonal one below the main diagonal, then the corresponding element must be involved in exactly one exchange (on the last stage). By following the "folding" process backwards in this way, we can assign a weight to each segment in the lattice which counts the number of exchanges the corresponding element will be involved in, if the path includes that segment. This process is illustrated, for $N = 4$, in Fig. 6.

Now, for any path through the lattice, if we sum the weights of its segments and divide by two (since each exchange involves two elements), we get the total number of exchanges used to sort the corresponding 2-ordered permutation. In fact, the sum of the weights of a path's vertical segments must equal the sum of the weights of its horizontal segments, and both sums count the number of exchanges. From Fig. 7, which has only vertical weights, we see that the example in Fig. 4 takes 12 exchanges, which agrees with our count in Fig. 1. From now on, we shall consider only vertical weights.

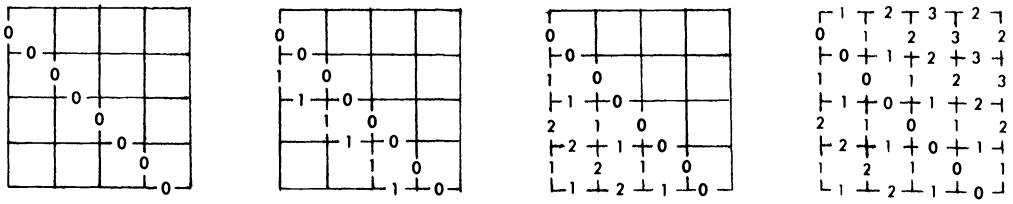


FIG. 6. *Assigning weights to the lattice.*

Continuing as in Fig. 6, the pattern of weights for general N is clear. First, we notice that all of the weights can be determined from the weights down the left edge. Since the folding is done along parallel diagonals, weights along diagonals are constant: if we denote the weight of the vertical segment from (i, j) to $(i+1, j)$ by $f(i, j)$, then we have

$$f(i,j)=\begin{cases} f(i-j,0) & \text{if } i \geqq j, \\ f(0,j-i) & \text{if } i \leqq j. \end{cases} \tag{2}$$

But from the first stage (the last "unfold") we know that

$$f(0,j+1)=f(j,0)+1 \tag{3}$$

and from the other stages we can write down an algorithmic definition of $f(i, 0)$:

(4)
$f(0, 0) := 0$; $i := 1$;
loop:
loop for $j := i-1$ **step** -1 **until** 0: $f(i, 0) := f(j, 0)+1$; $i := i+1$ **repeat**;
repeat;

In other words, in order to write down the values of $f(i, 0)$ for all i, first write down "0"; then repeatedly apply the following procedure: append to the string of numbers already written down the same string, but in reverse order, with each number incremented by 1. The value of $f(i, 0)$ is the ith number written.

This function, which is central to the study of data movement in Batcher's method, has a number of interesting properties. Since we shall be using it extensively, it will be convenient to drop the second argument and work with a more mathematical recursive definition:

(5)
$$f(0)=0,$$
$$f(2^n+j)=f(2^n-1-j)+1, \qquad n \geqq 0, \quad 0 \leqq j < 2^n.$$

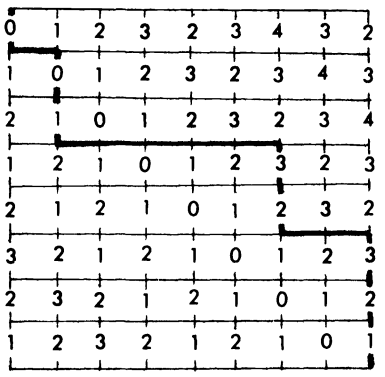
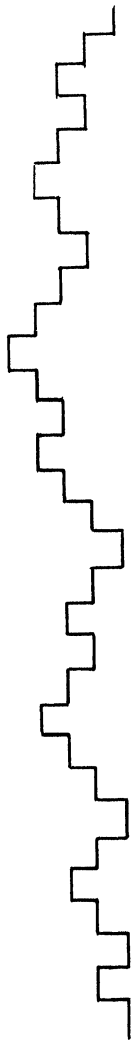


FIG. 7. *Vertical weights for* $N = 8$.



k	0	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	26	27	28	29	30	31
f(k)	0	1	2	1	2	3	2	1	2	3	4	3	2	3	2	1	2	3	4	3	4	5	4	3	2	3	4	3	2	3	2	1
F(k)	0	0	1	3	4	6	9	11	12	14	17	21	24	26	29	31	32	34	37	41	44	48	53	57	60	62	65	69	72	74	77	79
∇f(k)		1	1	-1	1	1	-1	-1	1	1	-1	-1	-1	1	-1	-1	1	1	1	-1	1	1	-1	-1	-1	1	1	-1	-1	1	-1	-1

FIG. 8. *The weight function.*

From this definition, we can explicitly evaluate the function for some arguments. For example, taking $j = 2^n - 1$, we find that $f(2^n - 1) = 1$ for $n \geqq 1$; then taking $j = 0$ gives $f(2^n) = 2$ for $n \geqq 0$ and taking $j = 2^{n-1}$ gives $f(3 \cdot 2^{n-1}) = 2$ for $n \geqq 2$. For other arguments, things are more complicated. However, there is a simple interpretation based on the binary representation of the argument. The binary representation of $2^n - 1 - j$ $(0 \leqq j < 2^n)$ is the "ones' complement" of the binary representation of $2^n + j$ (change 0's to 1's and 1's to 0's; then ignore leading zeros). Therefore, for example, $f(999) = f(1111100111_2) = f(11000_2) + 1 = f(111_2) + 2 = f(0) + 3 = 3$. The value of $f(k)$ for all k is exactly the number of times the binary representation of k changes parity. Figure 8 gives the value of $f(k)$ for $0 \leqq k < 32$ along with a graph of the function and values for $F(k) = \sum_{0 \leqq j < k} f(j)$ (the area under the curve) and $\nabla f(k) \equiv f(k) - f(k-1)$ (the slope), which we shall have use for later.

2. The worst case. To find the maximum number of exchanges that Batcher's algorithm will require, we can use the lattice diagram directly. The maximum number of exchanges is just the maximum possible weight of a path in the lattice diagram. Figure 9 shows the paths of highest weight for $N = 2, 4, 8, 16$.

A cursory inspection of Fig. 9 could lead to the conjecture that, at least for $N = 2^n$, the worst case might be the unique path through the lattice which contains the highest weights. Unfortunately, the situation is more complicated than this, as illustrated in Fig. 10 for $N = 32$. However, it does turn out that we need to examine only a few paths. Consider the paths through the lattice defined, for each integer k, as follows: proceed right along the top until encountering the first line with weight k. Then proceed down and to the right (along the diagonal of lines with weight k). After reaching the right edge of the lattice, proceed down to the corner. Figure 11 illustrates these paths, which we shall refer to as *major diagonals*, for $N = 32$. (Note that the last major diagonal is the unique path containing the highest weights.)

LEMMA. *The path of highest weight through the lattice must be one of the major diagonals.*

Proof. Clearly, for any path with segments below the main diagonal (the first major diagonal), there is a path of higher weight whose segments are all on or above

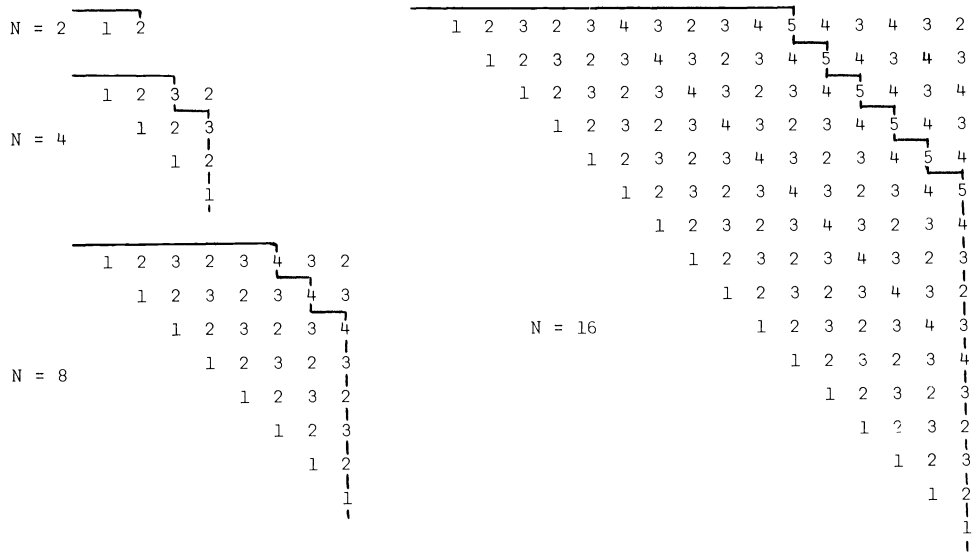


FIG. 9. *Worst-case lattice paths.*

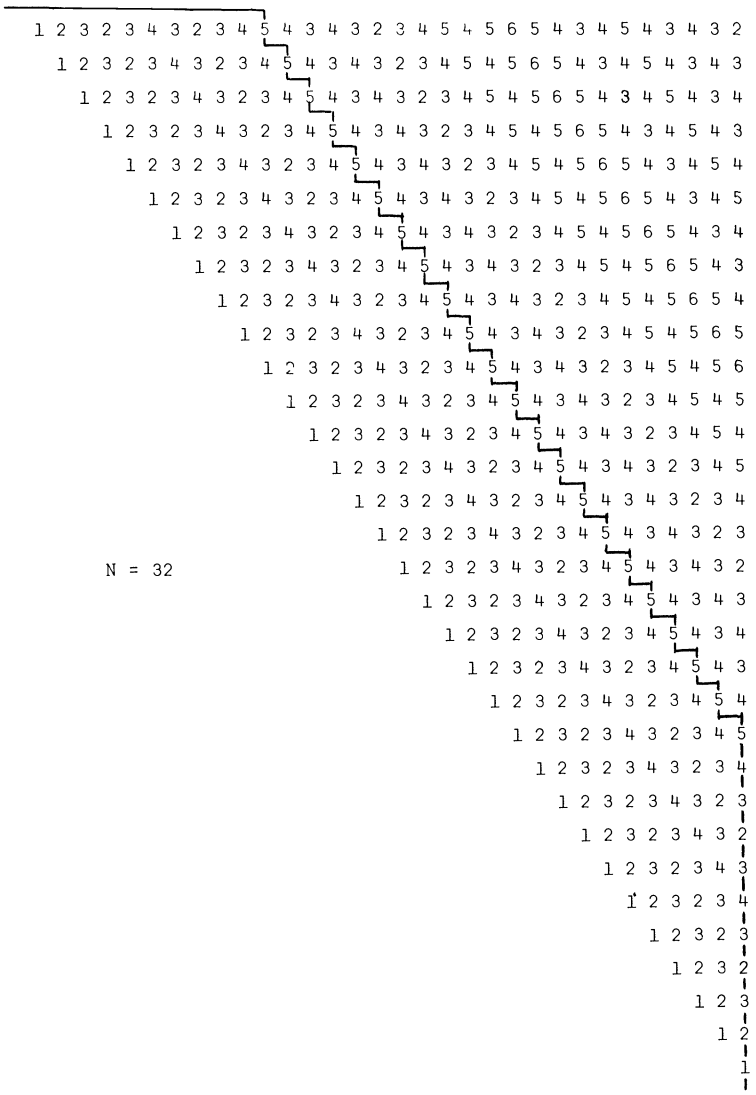


FIG. 10. *Worst-case path not containing highest weights.*

the main diagonal. Now, for any such path, consider the rightmost major diagonal which it crosses (has a segment in common with). The path must contain, sometime after the crossing, all of the weights which the major diagonal has on its vertical segment. None of the remaining weights can be higher than those on the major diagonal, because a higher weight would imply that the path crosses a major diagonal farther to the right. Therefore, for every path through the lattice, we can find a major diagonal whose weight is at least as high. □

Our problem is now reduced to finding the weights of all the major diagonal paths, and the maximum of these. To do so, we need to define

$$f^{-1}(k) = \{\text{smallest } j \text{ for which } f(j) = k\}$$

and

$$F(k) = \sum_{0 \le j < k} f(j).$$

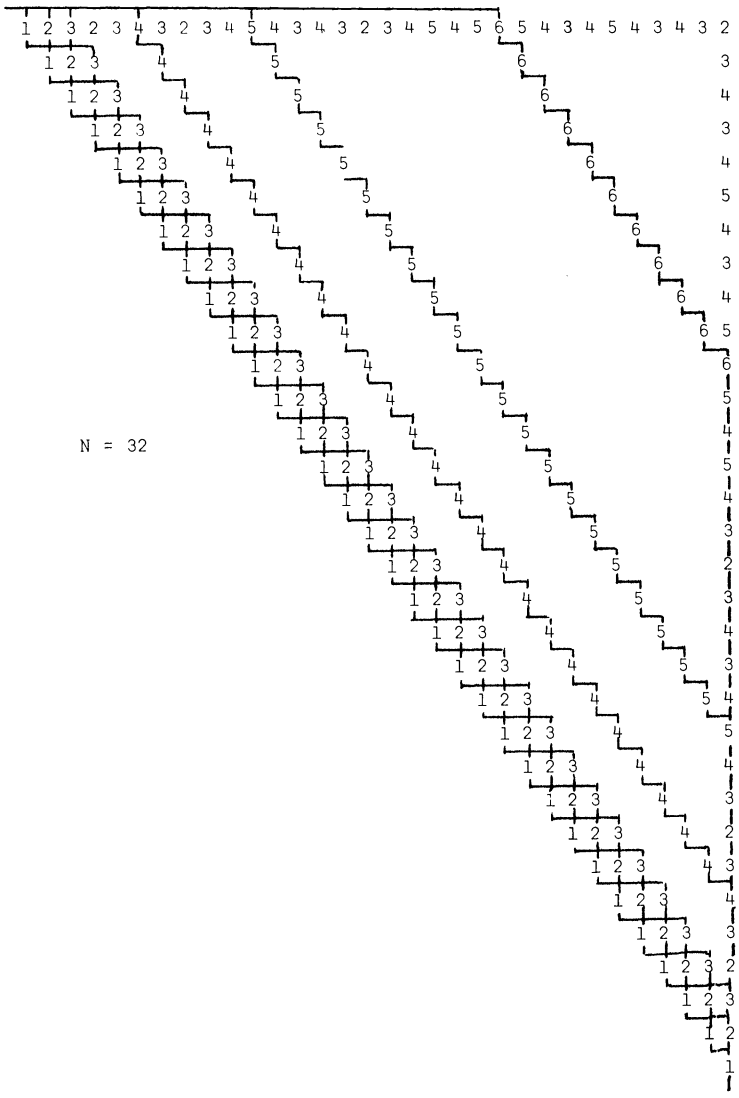


FIG. 11. *Major diagonals.*

The $(k+1)$st major diagonal path has $N-f^{-1}(k)$ segments along the diagonal, with weights $(k+1)$, and $f^{-1}(k)$ segments along the right edge, with total weight $F(f^{-1}(k))+f^{-1}(k)$. Therefore, if we let $w(k)$ denote the total weight of the $(k+1)$st major diagonal path, then

$$w(k)=F(f^{-1}(k))+f^{-1}(k)+(k+1)(N-f^{-1}(k))$$
$$=(k+1)N+F(f^{-1}(k))+kf^{-1}(k),$$

and we need to derive explicit expressions for $f^{-1}(k)$ and $F(f^{-1}(k))$.

A recurrence for $f^{-1}(k)$ follows immediately from the way that the function $f(k)$ "reflects" between powers of two. From (5) it is easy to prove by induction the difference between $f^{-1}(k-1)$ and $2^{k-1}-1$ must be the same as the difference between 2^{k-1} and $f^{-1}(k)$. (Also see Fig. 8.) In other words,

$$2^{k-1}-1-f^{-1}(k-1)=f^{-1}(k)-2^{k-1}.$$

Multiplying by $(-1)^k$ and telescoping, we find that

$$(-1)^k f^{-1}(k) = \sum_{0 \leq j \leq k} (-2)^j - \sum_{0 \leq j \leq k} (-1)^j$$

which leads, after the two geometric sums are evaluated, to the result

$$f^{-1}(k) = \frac{1}{6}(2^{k+2} - (-1)^k - 3). \tag{6}$$

As expected, these are all the numbers (0, 1, 2, 5, 10, 21, $\cdots$) whose binary representations alternate between 0 and 1. These numbers change parity most often, and so have the highest values of $f(k)$.

The calculation of $F(f^{-1}(k))$ is more complicated. First, we can set up a recurrence similar to the one which defines $f(k)$. Suppose that $2^{n-1} < k \leq 2^n$. We separate off the first 2^{n-1} terms of the sum and then apply the recurrence for $f(k)$ to the remaining terms:

$$\begin{aligned} F(k) &= \sum_{0 \leq j < 2^{n-1}} f(j) + \sum_{2^{n-1} \leq j < k} f(k) \\ &= F(2^{n-1}) + \sum_{0 \leq j < k-2^{n-1}} f(2^{n-1} + j) \\ &= F(2^{n-1}) + \sum_{0 \leq j < k-2^{n-1}} (f(2^{n-1} - 1 - j) + 1) \\ &= F(2^{n-1}) + \sum_{2^n - k \leq j < 2^{n-1}} f(j) + k - 2^{n-1} \\ &= 2F(2^{n-1}) - F(2^n - k) + k - 2^{n-1} \quad \text{for } 2^{n-1} < k \leq 2^n. \end{aligned}$$

In particular, if we take $k = 2^n$, then the formula becomes $F(2^n) = 2F(2^{n-1}) + 2^{n-1}$, which telescopes immediately to the solution

$$F(2^n) = n2^{n-1}.$$

Substituting this value, we find that

$$F(k) = -F(2^n - k) + (n-2)2^{n-1} + k \quad \text{for } 2^{n-1} < k \leq 2^n. \tag{7}$$

As before, the form of this recurrence clearly suggests that the value of $F(k)$ depends on the binary representation of k (and the dependence is much more complicated than for $f(k)$). Fortunately, the points $f^{-1}(k)$ at which we need to evaluate the function have a simple binary representation. We can get an explicit formula for $F(f^{-1}(k))$ by noticing from the ones' complement of the binary representation that $2^k - f^{-1}(k) - 1 = f^{-1}(k-1)$, so $F(2^k - f^{-1}(k)) = F(2^k - f^{-1}(k) - 1) + f(2^k - f^{-1}(k) - 1) = F(f^{-1}(k-1)) + f(f^{-1}(k-1)) = F(f^{-1}(k-1)) + k - 1$, and, since $2^{k-1} < f^{-1}(k) \leq 2^k$, the recurrence (7) becomes

$$F(f^{-1}(k)) = -F(f^{-1}(k-1)) - (k-1) + (k-2)2^{k-1} + f^{-1}(k).$$

This recurrence, after both sides are multiplied by $(-1)^k$, telescopes into a summation (note that $F(f^{-1}(0)) = F(0) = 0$):

$$(-1)^k F(f^{-1}(k)) = \sum_{1 \leq j \leq k} (j-1)(-1)^{j-1} - \sum_{1 \leq j \leq k} (j-2)(-2)^{j-1} + \sum_{1 \leq j \leq k} f^{-1}(j)(-1)^j.$$

After substituting for $f(j)$, we are left with a number of elementary sums: they can all be evaluated using the well-known identities for $\sum_{0 \leq j \leq k} x^j$ and $\sum_{0 \leq j \leq k} jx^j$ (see, for

example, Knuth [12, exercise 1.2.3-16]) with the result

$$F(f^{-1}(k))=\frac{1}{18}((3k-1)2^{k+1}-9k-(3k-2)(-1)^k). \tag{8}$$

Comparing this with the formula $F(2^n)=n2^{n-1}$, we find that both are of the form $F(N)=\frac{1}{2}N\lg N=O(N)$. In fact, it is possible to prove by induction that this does hold for all N, but the linear term is a complicated function of the binary representation of N.

Substituting these values for $f^{-1}(k)$ and $F(f^{-1}(k))$ into the formula given above for the total weight of the $(k+1)$st major diagonal path, we get the expression

$$\begin{aligned} w(k)&=(k+1)N+\frac{1}{18}((3k-1)2^{k+1}-9k-(3k-2)(-1)^k)-\frac{1}{6}k(2^{k+2}-(-1)^k-3) \\ &=(k+1)N-\frac{1}{9}((3k+1)2^k-(-1)^k). \end{aligned} \tag{9}$$

This function is clearly increasing for small k and decreasing for large k. The maximum number of exchanges required by Batcher's algorithm is the maximum value of the function. Note that the total weight of the last major diagonal is $\frac{2}{3}N\lg N+O(N)$. The following theorem shows that the proper choice of k leads to a path of much higher weight.

THEOREM 1. *Let $B_N^{\max}$ denote the maximum number of exchanges required when Batcher's odd-even merge is applied to a 2-ordered file of $2N$ elements. Then*

$$B_N^{\max}=(k'+1)N-\frac{1}{9}((3k'+1)2^{k'}-(-1)^{k'})$$

where k' is the largest integer satisfying $\frac{1}{9}((3k'+4)2^{k'-1}-\frac{2}{9}(-1)^{k'})\leqq N$. Asymptotically,

$$B_N^{\max}=N\lg N-N\lg\lg N+O(N).$$

Proof. Following the discussion above, the lemma says that we need only consider the major diagonals. We have:

$$B_N^{\max}=\max_{0\leqq k\leqq k''}(w(k)),$$

where k'' is the index of the last major diagonal (the largest integer satisfying $f(k'')\leqq N$). To calculate this maximum, consider the difference

$$w(k)-w(k-1)=N-\frac{1}{9}((3k+4)2^{k-1}-2(-1)^k).$$

The function $w(k)$ increases as long as this difference is positive, then decreases when the difference is negative. Clearly the maximum is $w(k')$, where k' is the largest integer for which the difference is positive. To complete the proof, it is necessary to show that this maximum is realizable, i.e. that $f^{-1}(k')\leqq N$. This is easily verified: we have $N\geqq\frac{1}{9}((3k'+4)2^{k'-1}-2(-1)^{k'})=\frac{1}{6}(\frac{2}{3}(3k'+4)2^{k'-1}-\frac{4}{3}(-1)^{k'})\geqq\frac{1}{6}(2^{k'+2}-(-1)^{k'}-3)=f^{-1}(k')$.

To find an asymptotic estimate of how the maximum grows with N, we start with the inequalities which define k':

$$\frac{1}{9}(3k'+4)2^{k'-1}-\frac{2}{9}(-1)^{k'}\leqq N<\frac{1}{9}(3k'+7)2^{k'}+\frac{2}{9}(-1)^{k'}.$$

After ignoring the $(-1)^{k'}$ factors (the inequalities still hold without them), if we multiply by 3, take logs (base 2) and solve for k', we get an inverted form of this formula:

$$\lg\left(3N-\frac{2}{3}\right)-\lg\left(k'+\frac{7}{3}\right)<k'<\lg\left(3N+\frac{2}{3}\right)-\lg\left(k'+\frac{4}{3}\right)+1.$$

These inequalities can now be iterated to give

$$\lg\left(3N-\frac{2}{3}\right)-\lg\left(\lg\left(3N+\frac{2}{3}\right)-\lg\left(k'+\frac{4}{3}\right)+1+\frac{7}{3}\right)$$

$$<k'<\lg\left(3N+\frac{2}{3}\right)-\lg\left(\lg\left(3N-\frac{2}{3}\right)-\lg\left(k'+\frac{7}{3}\right)+\frac{4}{3}\right)+1.$$

Now both sides reduce to the same asymptotic expression; we must have

$$k'=\lg 3N-\lg\lg 3N+O(1)$$
$$=\lg N-\lg\lg N+O(1)$$

and substituting this into the formula for $B_N^{\max}$ leads to the stated asymptotic estimate. □

The easiest way to actually compute $B_N^{\max}$ for any practical value of N is to use a table, since k' takes on relatively few values for realistic N. Table 1 gives the values of $B_{N_k}^{\max}$ for the inflection points N_k: numbers of the form $\frac{1}{9}((3k+4)2^{k-1}-2(-1)^k)$. Between the kth and $(k+1)$st inflection points the function is linear in N with slope $(k+1)$. Therefore, to compute $B_N^{\max}$ for arbitrary N, find the largest k for which $N_k \leqq N$, call it k', and set $B_N^{\max}=B_{N_{k'}}^{\max}+(k'+1)(N-N'_k)$. Table 2 shows the results of such a computation for $N=2^n$, $0\leqq n\leqq 20$.

TABLE 1
Inflection points for the worst case.

k	$N_k=\frac{1}{9}((3k+4)2^{k-1}-2(-1)^k)$	$B_{N_k}^{\max}$	C_{N_k}	$B_{N_k}^{\max}/C_{N_k}$
1	1	1	1	1.0000
2	2	3	3	1.0000
3	6	15	17	.8823
4	14	47	55	.8545
5	34	147	175	.8400
6	78	411	497	.8270
7	178	1,111	1,347	.8248
8	398	2,871	3,469	.8276
9	882	7,227	8,679	.8327
10	1,934	17,747	21,161	.8387
11	4,210	42,783	50,749	.8430
12	9,102	101,487	120,147	.8447
13	19,570	237,571	280,353	.8474
14	41,870	549,771	646,255	.8507
15	89,202	1,259,751	1,474.565	.8543
16	189,326	2,861,735	3,335,241	.8580
17	400,498	6,451,659	7,485,673	.8619
18	844,686	14,447,043	16,689,831	.8656
19	1,776,754	32,156,335	36,991,437	.8692

TABLE 2
The worst case at inflection points for the number of comparators.

N	k'	$B_N^{\max}$	C_N	$B_N^{\max}/C_N$
1	1	1	1	1.0000
2	2	3	3	1.0000
4	2	9	9	1.0000
8	3	23	25	.9200
16	4	57	65	.8769
32	4	137	161	.8509
64	5	327	385	.8493
128	6	761	897	.8484
256	7	1,735	2,049	.8468
512	8	3,897	4,609	.8455
1,024	9	8,647	10,241	.8444
2,048	10	19,001	22,529	.8434
4,096	10	41,529	49,153	.8449
8,192	11	90,567	106,497	.8504
16,384	12	196,153	229,377	.8552
32,768	13	422,343	491,521	.8593
65,536	14	904,761	1,049,477	.8621
131,072	15	1,929,671	2,228,225	.8660
262,144	16	4,099,641	4,718,593	.8668
524,288	17	8,679,879	9,961,473	.8713
1,048,576	18	18,320,953	20,971,521	.8736

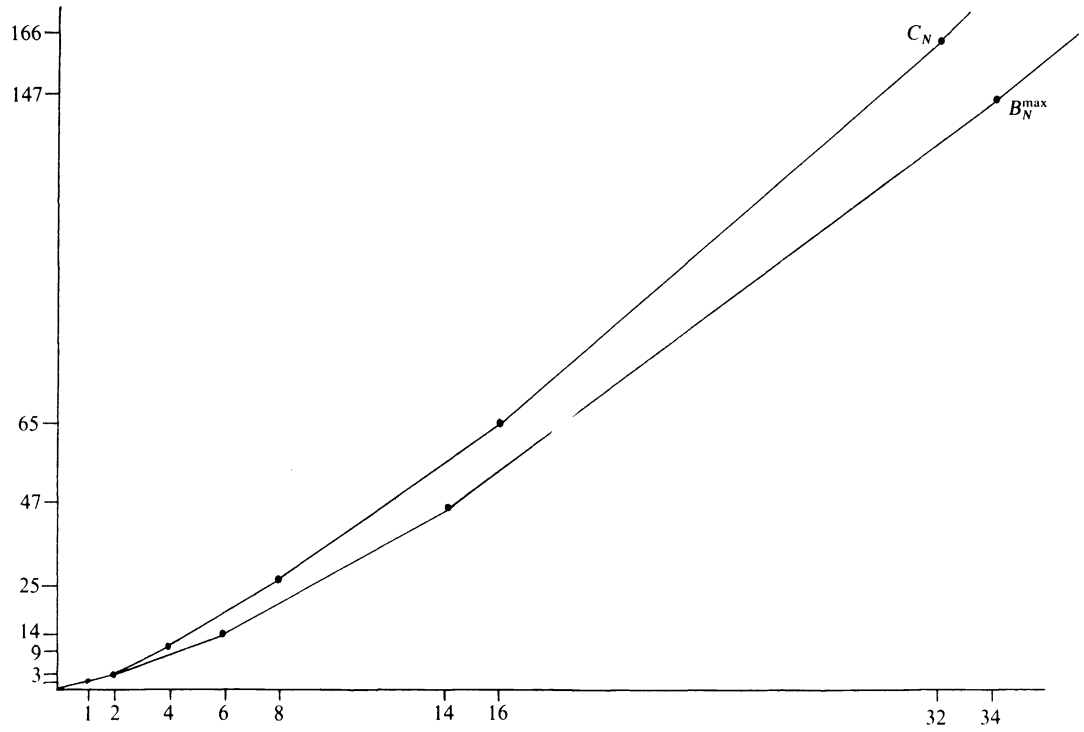


FIG. 12. *Number of comparators and maximum number of exchanges.*

Also given in Tables 1 and 2 is another quantity of interest: the percentage of comparators that perform exchanges in the worst case (the ratio $B_N^{\max}/C_N$). The graphs of C_N and $B_N^{\max}$ are both piecewise linear, with slope incrementing by 1 at each inflection point. This is illustrated in Fig. 12, for small N, where the effect is most pronounced. Both curves are therefore concave upwards, so they are closest together at the inflection points for C_N (numbers of the form 2^n) and farthest apart at the inflection points for $B_N^{\max}$. Tables 1 and 2 therefore show that the ratio is between 82% and 87% for all but very small and very large values of N. As $N \to \infty$ the ratio (slowly) approaches 1, which follows from a simple asymptotic calculation:

$$
\begin{aligned}
B_N^{\max}/C_N &= \frac{N \lg N - N \lg \lg N + O(N)}{N \lg N + O(N)} \\
&= \left(1 - \frac{\lg \lg N}{\lg N} + O\left(\frac{1}{\lg N}\right)\right)\left(1 + O\left(\frac{1}{\lg N}\right)\right) \qquad (10) \\
&= 1 - \frac{\lg \lg N}{\lg N} + O\left(\frac{1}{\lg N}\right).
\end{aligned}
$$

The value of N must be truly astronomical for the ratio to be close to 1.

3. The average case. The lattice diagram correspondence of § 1 leads to an expression for B_N, the average number of comparisons taken by Batcher's method to sort a random 2-ordered file of $2N$ elements. The derivation is long, and conveniently divides into two parts. First, we shall perform some manipulations which are somewhat independent of our particular weight function $f(j)$. and so lead to results applicable to the analysis of other properties of 2-ordered permutations (or other merging algorithms). The second part of the derivation uses complex analysis and some particular properties of $f(j)$, and leads to a method for computing B_N to any desired asymptotic accuracy.

One way to determine B_N, using the lattice diagram correspondence, would be to find the weight of each of the $\binom{2N}{N}$ paths through the lattice, sum them, and divide by $\binom{2N}{N}$. An alternate way is to find the number of paths which pass through each vertical line in the lattice, multiply by the weight, sum over all vertical lines, and then divide by $\binom{2N}{N}$. In § 1, we defined the weight of the vertical segment from (i, j) to $(i+1, j)$ to be $f(i, j)$ and derived some simple properties of this function. Now, the number of paths from $(0, 0)$ to (i, j) is clearly $\binom{i+j}{i}$, and the number of paths from $(i+1, j)$ to (N, N) is $\binom{2N-i-j-1}{N-j}$, so the total number of paths which pass through the vertical segment from (i, j) to $(i+1, j)$ is the product of these two binomial coefficients. Therefore,

$$
\binom{2N}{N} B_N = \sum_{0 \le i < N} \sum_{0 \le j \le N} f(i, j) \binom{i+j}{i} \binom{2N-i-j-1}{N-j}. \qquad (11)
$$

This can be transformed into an expression involving the single argument weight function defined in (5) because of the symmetries which are available. The first step is

to split the sum on j:

$$\binom{2N}{N} B_N = \sum_{0\leqq i<N} \sum_{0\leqq j\leqq i} f(i,j)\binom{i+j}{i}\binom{2N-i-j-1}{N-j} + \sum_{0\leqq i<N} \sum_{i<j\leqq N} f(i,j)\binom{i+j}{i}\binom{2N-i-j-1}{N-j}.$$

If we change j to $i-j$ in the first term and change j to $i+j+1$ and then i to $N-1-i$ in the second term, the terms can be recombined:

$$\begin{aligned}\binom{2N}{N} B_N &= \sum_{0\leqq i<N} \sum_{0\leqq j\leqq i} f(i,i-j)\binom{2i-j}{i}\binom{2N-2i+j-1}{N-i+j} \\ &\quad + \sum_{0\leqq i<N} \sum_{0\leqq j\leqq i} f(N-1-i,N-i+j)\binom{2N-2i+j-1}{N-i-1}\binom{2i-j}{i-j} \\ &= \sum_{0\leqq i<N} \sum_{0\leqq j\leqq i} (f(i,i-j)+f(N-1-i,N-i+j))\binom{2i-j}{i-j}\binom{2N-2i+j-1}{N-i-1}.\end{aligned}$$

Now, equations (2) and (3) in § 1 tell us that $f(i,i-j)=f(j,0)$ and $f(N-1-i,N-i+j)=f(0,j+1)=f(j,0)+1$. Adopting the shorthand $f(j,0)\equiv f(j)$, we have

$$\binom{2N}{N} B_N = \sum_{0\leqq i<N} \sum_{0\leqq j\leqq i} (2f(j)+1)\binom{2i-j}{i-j}\binom{2N-2i+j-1}{N-i-1}.$$

Interchanging the order of summation and changing i to $i+j$ gives

$$\binom{2N}{N} B_N = \sum_{0\leqq j<N} (2f(j)+1) \sum_i \binom{2i+j}{i}\binom{2N-2i-j-1}{N-i-j-1}.$$

The inner sum remaining in this expression was studied as far back as 1902 by Jensen [9], who gave an identity which implies that

$$\sum_i \binom{2i+j}{i}\binom{2N-2i-j-1}{N-j-i-1} = \sum_{i\geqq 0} \binom{2N-1-i}{N-i-j-1} 2^i$$

(see also Gould and Kaucký [8] or Knuth [12, exercise 1.2.6-28] for more general versions of this identity). This particular sum can be simplified even further, by applying the addition formula for binomial coefficients to set up a recurrence relation describing an alternate form of the sum. Denoting the sum by S_{Nj}, we have

$$\begin{aligned}S_{Nj} &\equiv \sum_{i\geqq 0} \binom{2N-1-i}{N-i-j-1} 2^i \\ &= \sum_{i\geqq 0} \left(\binom{2N-2-i}{N-i-j-1} + \binom{2N-2-i}{N-i-j-2}\right) 2^i \\ &= \sum_{i\geqq 1} \left(\binom{2N-1-i}{N-i-j} + \binom{2N-1-i}{N-i-j-1}\right) 2^{i-1} \\ &= \frac{1}{2} S_{N(j-1)} + \frac{1}{2} S_{Nj} - \frac{1}{2}\binom{2N}{N-j}.\end{aligned}$$

This implies that $S_{Nj} = S_{N(j+1)} + \binom{2N}{N-j-1}$, which telescopes to give the alternate

form

$$\sum_i \binom{2i+j}{i}\binom{2N-2i-j-1}{N-i-j-1} = \sum_{0\leq k<N-j}\binom{2N}{k}. \tag{12}$$

When substituted into our formula for B_N, this leads to the following result:

THEOREM 2. *For any assignment of weights to an $N\times N$ lattice satisfying $f(i,j)=f(i-j,0)$ for $i\geq j$, $f(i,j)=f(0,j-i)$ for $j\leq i$ and $f(0,j+1)=f(j,0)+1$, the average weight of a path through the lattice is*

$$B_N = \sum_{k\geq 1}\frac{\binom{2N}{N-k}}{\binom{2N}{N}}(2F(k)+k)$$

where $F(k)=\sum_{0\leq j<k} f(j)$ with $f(j)\equiv f(j,0)$.

Proof. From the discussion above, we have

$$\binom{2N}{N}B_N = \sum_{0\leq j<N}(2f(j)+1)\sum_{0\leq k<N-j}\binom{2N}{k}$$

which can be transformed into the stated result by changing k to $N-k$ and interchanging the order of summation. □

To proceed further, we need to examine the functions $f(k)$ and $F(k)$ in much more detail.

Digressing slightly, we can now easily compute the average number of inversions in a 2-ordered permutation as an example of the use of Theorem 2. (An inversion is an index pair (i,j) satisfying $i<j$ and $A[i]>A[j]$.) The lattice diagram correspondence and the initial expression (11) for B_N above are taken from Knuth's treatment of this problem [13, pp. 86–88 and exercises 5.2.1-12, 14, 15]. Knuth shows that the number of inversions in a 2-ordered permutation is equal to the area between its path in the lattice and the main diagonal. (Proof: changing ⌊ to ⌉ below the diagonal or ⌉ to ⌊ above the diagonal reduces the number of inversions by one and reduces the area by one unit.) The permutation in Fig. 4 has 12 inversions. The appropriate assignment of weights to the lattice is to take $f(i,j)=|i-j|$. This function satisfies (2) and (3), and we have $f(k)=k$ and $F(k)=\binom{k}{2}$. Then from the theorem we find that the average number of inversions must be

$$\sum_{k\geq 1}\frac{\binom{2N}{N-k}}{\binom{2N}{N}}k^2.$$

This sum can be easily evaluated by writing

$$\binom{2N}{N-k}k^2 = \binom{2N}{N-k}(N^2-(N-k)(N+k)) = N^2\binom{2N}{N-k} - 2N(2N-1)\binom{2N-2}{N-k-1}$$

and applying the identity $\sum_{k\geq 1}\binom{2N}{N-k}+\frac{1}{2}\left(2^{2N}-\binom{2N}{N}\right)$. These calculations lead to the result

$$\frac{N2^{2N-2}}{\binom{2N}{N}}$$

for the average number of inversions in a 2-ordered permutation of $2N$ elements. (This checks with Knuth's result, but his derivation depends on particular properties of $f(k)=k$.) Knuth suggests that such a simple answer deserves a simple derivation; perhaps a direct combinatorial derivation of Theorem 2 could be devised. In any case, the weight function $f(k)$ for Batcher's method is much more complicated than $f(k)=k$ (we don't even have a closed formula for it), and our problem will involve much more analysis.

Theorem 2 does lead to an easy way to *compute* B_N for all practical values of N. Expanding the binomial coefficients in their factorial representations, we find that

$$B_N=\sum_{1\leq k\leq N}(2F(k)+k)\prod_{0\leq j<k}\frac{N-j}{N+j+1}.$$

From this representation, we can see that the exact value of B_N can be computed in $O(N)$ steps, as follows:

```
product := 1; sum := 0;
loop for 1 ≦ k ≦ N:
    product := product*(N-k+1)/(N+k);
    sum := sum+(2*F(k)+k)*product;
repeat;
```
(13)

This program assumes that $F(k)$ has been precomputed and stored in an array $F(1:N)$, say by using (4) to compute $f(k)$ and then passing through the array once more to compute $F(k)$. This requirement for N memory cells can be removed by computing $F(k)$ incrementally within the loop. [We have $F(k)=F(k-1)+f(k-1)$, and $f(k)$ can be computed from $f(k-1)$ by looking at the binary representations of $(k-1)$ and k. The binary representation of k is obtained from the binary representation of $k-1$ by changing the rightmost 0 to 1 and all the 1's to its right to 0's. (All numbers are assumed to have 0 as the leftmost digit.) This will increment by 1 the number of times the binary representation changes parity (the value of f) if the bit to the left of the rightmost 0 in $(k-1)_2$ is 0; otherwise it will decrement f by 1. Therefore, we need only test this one bit: this can be done by performing an "exclusive or" of $(k-1)_2$ with $(k)_2$, adding 1, then "and"ing the result with $(k-1)_2$ (or $(k)_2$). If the result is 0, then $f(k)=f(k-1)+1$, otherwise $f(k)=f(k-1)-1$.] The program can be further improved because the terms become very, very small as k gets large. If we put in a test to leave the loop when the terms to be added become smaller than the smallest representable number in our computer, then it turns out that the loop is iterated only about $O(\sqrt{N})$ times for large N (we shall see why later). Thus exact values of B_N can be computed very quickly.

TABLE 3
Average number of exchanges (exact).

N	B_N	$\frac{B_N-(1/4)N\lg N}{N}$
1	.500	.50000000
2	1.333	.41666667
4	3.600	.40000000
8	9.131	.39141414
16	22.221	.38881721
32	52.370	.38657069
64	120.735	.38647725
128	273.339	.38546127
256	610.795	.38591836
512	1,349.217	.38519013
1,024	2,955.039	.38578023
2,048	6,420.731	.38512273
4,096	13,868.014	.38574580
8,192	29,778.788	.38510590
16,384	63,663.918	.38573720
32,768	135,499.012	.38510170
65,536	287,423.532	.38573505
131,072	607,531.912	.38510065
262,144	1,280,765.989	.38573451
524,288	2,692,271.510	.38510038
1,048,576	5,647,351.813	.38573438

Table 3 shows exact values of B_N for $N=2^n$, computed in this way. By taking differences in this table, it is quickly discovered that these numbers grow with $N \lg N$, and the coefficient is apparently $1/4$. Subtracting $\frac{1}{4}N \lg N$ from B_N and dividing by N gives the third column, which leads to the immediate conjecture that

$$B_N \approx \frac{1}{4}N \lg N + .385N$$

at least for $N=2^n$. In fact, a quick calculation with (13) proves that this formula is accurate to within 0.1% for $2^7 \leqq N \leqq 2^{20}$ (and to within 1% for $2<N<2^7$). From a practical standpoint, we are done, since we can accurately calculate B_N for any realistic value of N. From a theoretical standpoint, this answer is somewhat unsatisfactory, and the rest of the paper will be devoted to an analytic verification of this result. It turns out that precise formulas for B_N can be derived to any desired asymptotic accuracy; in particular, the coefficient of the linear term can be expressed in terms of classical mathematical functions. The derivation is an interesting example of a difficult type of asymptotic analysis, and it uncovers some interesting aspects of the structure of Batcher's method.

It will be convenient to begin by using the addition formula for binomial coefficients to transform the equation in Theorem 2 for B_N into a sum involving $\nabla f(k)$, which is simpler to work with than $F(k)$. First, just as in the derivation for the number of inversions, we can perform the summation $\sum_{k\geqq 1}\binom{2N}{N-k}k = \frac{1}{2}N\binom{2N}{N}$, which

leaves

$$\begin{aligned}\binom{2N}{N}\left(B_N-\frac{N}{2}\right) &= 2\sum_{k\geq 1}\binom{2N}{N-k}F(k)\\ &= 2\sum_{k\geq 1}\left(\binom{2N-2}{N-k}+2\binom{2N-2}{N-k-1}+\binom{2N-2}{N-k-2}\right)F(k)\\ &= 2\Bigg(\sum_{k\geq 0}\binom{2N-2}{N-k-1}F(k+1)+2\sum_{k\geq 1}\binom{2N-2}{N-k-1}F(k)\\ &\qquad +\sum_{k\geq 2}\binom{2N-2}{N-k-1}F(k-1)\Bigg)\\ &= 4\binom{2N-2}{N-1}\left(B_{N-1}-\frac{N-1}{2}\right)+2\sum_{k\geq 1}\binom{2N-2}{N-k-1}\nabla f(k).\end{aligned}$$

When both sides are divided by 4^N, this recurrence telescopes to a sum, and leads to the formulation

$$B_N=\frac{1}{2}\frac{4^N}{\binom{2N}{N}}\sum_{1\leq j<N}\frac{\binom{2j}{j}}{4^j}\sum_{k\geq 1}\frac{\binom{2j}{j-k}}{\binom{2j}{j}}\nabla f(k)+\frac{1}{2}N. \tag{14}$$

We shall now concentrate on evaluating the inner sum

$$b_j=\sum_{k\geq 1}\frac{\binom{2j}{j-k}}{\binom{2j}{j}}\nabla f(k). \tag{15}$$

After we have derived an asymptotic expression for b_j, we shall easily be able to deal with B_N.

Formulas of this type (involving a sum over the lower index of a binomial coefficient) appear relatively frequently in combinatorial analysis and the analysis of algorithms. We have already seen one example, counting inversions in a 2-ordered permutation. Knuth [13] gives several other specific examples which arise in the analysis of algorithms: bubble sort, digital searching, and radix exchange sort. Paths in a lattice may also be used to model other combinatorial problems, such as tree enumeration and the classical ballot problem, and formulas similar to Theorem 2 arise in the analysis. The method that we shall use is called the "gamma-function" method and is attributed by Knuth to N. G. de Bruijn. A derivation using the method is outlined in a paper on tree enumeration by de Bruijn, Knuth and S. O. Rice [5], and a similar description may be found in Knuth [13, pp. 132–134]. However, it will be useful to present the method in some detail here because our function $\nabla f(k)$ is more complicated than the corresponding functions for the prior derivations.

One goal in an asymptotic derivation is to use methods which could, at least in principle, yield an answer good to any given asymptotic accuracy. We shall be content to get a formula for B_N good to within $O(\sqrt{N}\log N)$; we are most interested in the coefficients of the N log N and linear terms. It turns out that it is sufficient to get b_j

with $O(j^{-1/2})$ to achieve this accuracy. In both cases, the methods can yield better asymptotic accuracy, if desired.

The first step in evaluating b_j is to use Stirling's approximation to replace the binomial coefficients with an exponential. Stirling's approximation says that

$$\ln n! = \left(n+\frac{1}{2}\right)\ln n - n + \ln\sqrt{2\pi} + O\left(\frac{1}{n}\right).$$

Applying this to the binomial coefficients in b_j, we have

$$\binom{2j}{j-k}\Big/\binom{2j}{j} = \exp\{2\ln j! - \ln(j+k)! - \ln(j-k)!\}$$

$$= \exp\left\{2\left(j+\frac{1}{2}\right)\ln j - \left(j+\frac{1}{2}\right)(\ln(j+k)+\ln(j-k))\right.$$

$$\left. -k(\ln(j+k)-\ln(j-k)) + O\left(\frac{1}{j}\right) + O\left(\frac{1}{j+k}\right) + O\left(\frac{1}{j-k}\right)\right\}.$$

Now, the $O(1/(j+k))$ and $O(1/(j-k))$ terms render this approximation useless unless the value of k is restricted in some way. In this case, the appropriate restriction is to take $|k| \leqq j^{1/2+\varepsilon}$ for some small positive constant $\varepsilon > 0$ (the reason for this will become apparent below). With this restriction, we can replace $O(1/(j+k))$ and $O(1/(j-k))$ by $O(1/j)$. Also, we get the asymptotic expansions

$$\ln(j+k) = \ln j + \frac{\kappa}{j} - \frac{k^2}{2j^2} + \frac{k^3}{3j^3} - O(j^{4\varepsilon-2})$$

and

$$\ln(j-k) = \ln j - \frac{k}{j} - \frac{k^2}{2j^2} - \frac{k^3}{3k^3} + O(j^{4\varepsilon-2}) \quad \text{for } |k| \leqq j^{1/2+\varepsilon}.$$

Substituting these and simplifying, we find that several terms cancel, leaving

$$\frac{\binom{2j}{j-k}}{\binom{2j}{j}} = e^{-k^2/j}(1+O(j^{4\varepsilon-1})) \quad \text{for } |k| \leqq j^{1/2+\varepsilon}. \tag{16}$$

This estimate can be used in our expression for b_j because the terms for $|k| \geqq j^{1/2+\varepsilon}$ are negligibly small. We have

$$\binom{2j}{j-k} < \binom{2j}{j-j^{1/2+\varepsilon}} \quad \text{for } |k| > j^{1/2+\varepsilon},$$

so (16) implies that

$$\frac{\binom{2j}{j-k}}{\binom{2j}{j}} = e^{-j^{2\varepsilon}}(1+O(j^{4\varepsilon-1})) \quad \text{for } |k| > j^{1/2+\varepsilon} \tag{17}$$

and this is $O(j^{-m})$ for all $m > 0$. Now, we can split the sum for b_j into two parts and

apply (16) and (17) to replace the binomial coefficients with an exponential (recall that $|\nabla f(k)| = 1$):

$$
\begin{aligned}
b_j &= \sum_{1 \le k \le j^{1/2+\varepsilon}} \frac{\binom{2j}{j-k}}{\binom{2j}{j}} \nabla f(k) + \sum_{j^{1/2+\varepsilon} < k \le j} \frac{\binom{2j}{j-k}}{\binom{2j}{j}} \nabla f(k) \\
&= \sum_{1 \le k \le j^{1/2+\varepsilon}} e^{-k^2/j}(1 + O(j^{4\varepsilon-1}))\, \nabla f(k) + O(je^{-j^{2\varepsilon}}) \qquad (18) \\
&= \sum_{k \ge 1} e^{-k^2/j}\, \nabla f(k)(1 + O(j^{4\varepsilon-1})).
\end{aligned}
$$

The terms for which the estimate (16) is not valid are exponentially small, as is $e^{-k^2/j}$; therefore it doesn't matter which we use in the "tail" of the sum.

If we had a simple expression for $\nabla f(k)$ we could proceed to get an asymptotic expression for b_j by applying the Euler–MacLaurin summation formula to approximate the sum with an integral, then do the integration. For example, we could apply the methods of the previous paragraph to the formula for B_N in Theorem 2 to get the asymptotic formula

$$B_N = \sum_{k \ge 1} e^{-k^2/N}(2F(k) + k)(1 + O(N^{4\varepsilon-1})),$$

and from equation (7) it is easy to prove that $F(k) = \frac{1}{2} k \lg k + O(k)$ so that the Euler–MacLaurin summation gives the approximation

$$B_N = \int_1^\infty e^{-x^2/N}(x \lg x + (O(x))(1 + O(N^{4\varepsilon-1}))\, dx$$

which, after the substitution $t = x^2/n$, leads to the well-known "exponential integral" function (see [1]), with the result

$$B_N = \frac{1}{4} N \lg N + O(N).$$

This method cannot be extended to find the coefficient of N, since the precise equation for $F(k)$ is quite complicated and depends on the binary value of k. Similarly, a simple equation for $\nabla f(k)$ is not available, and we need to resort to more advanced techniques to get an accurate estimate for b_j (and, eventually, B_N).

The "gamma-function" method that we shall use to evaluate b_j makes use of the residue theorem from the theory of functions of a complex variable. Knopp [10], [11] is the classical text on the theory of functions, and is an excellent introduction to the subject of complex analysis. Other aspects of complex analysis and properties of the two special functions that we use, the gamma (Γ) function and Riemann's zeta (ζ) function, may be found in Whittaker and Watson [15]. And we shall make use of a number of identities from Abramowitz and Stegun [1] and some other references noted below. The idea is to express $e^{-k^2/j}$ as an integral in the complex plane involving the Γ-function, then interchange the order of integration and summation. Although we don't have a simple closed formula for $\nabla f(k)$, we will be able to express the resulting complex series involving $\nabla f(k)$ in terms of classical analytic functions. This is the key to the analysis, for then the integral can be evaluated by finding residues within an appropriate contour of integration.

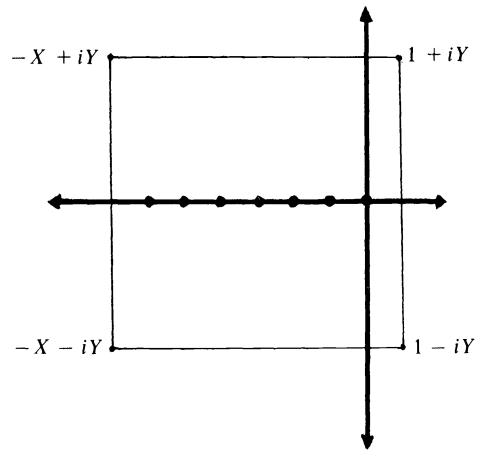


FIG. 13. *Contour of integration for* Γ*-function identity.*

We begin with the identity

$$e^{-r}=\frac{1}{2\pi i}\int_{1-i\infty}^{1+i\infty}\Gamma(z)r^{-z}\,dz. \tag{19}$$

This is the so-called Mellin transform of e^{-r} [7], a special case of Fourier inversion. We may prove this also directly from the residue theorem using the contour of integration R_{XY} shown in Fig. 13, and letting X and $Y\to\infty$. The function $\Gamma(z)r^{-z}$ has simple poles at $z=-k$, $k=0,1,2,\cdots$ with residue $r^k(-1)^k/k!$, so the value of the integral along R_{XY} is $\sum_{0\leq k<X}(-r)^k/k!$ which becomes e^{-r} as $X\to\infty$. The integral in (19) is the integral along the right boundary of R_{XY}; the integrals along the other boundaries vanish as X, $Y\to\infty$ because the Γ-function becomes exponentially small on them. (We shall skip the precise bounds here because they may be found in Knuth [13, p. 132] and we shall be doing similar calculations later.) Applying this identity to our formula (18) for b_j, we have

$$\begin{aligned}b_j&=\sum_{k\geq 1}\nabla f(k)\frac{1}{2\pi i}\int_{1-i\infty}^{1+i\infty}\Gamma(z)\left(\frac{k^2}{j}\right)^{-z}dz(1+O(j^{4\varepsilon-1}))\\&=\frac{1}{2\pi i}\int_{1-i\infty}^{1+i\infty}\Gamma(z)j^z\sum_{k\geq 1}\frac{\nabla f(k)}{k^{2z}}dz(1+O(j^{4\varepsilon-1})).\end{aligned}$$

(The reader may wish to check that the interchange of summation and integration is justified here because of absolute convergence.)

In order to proceed further we need to know the properties of the function $\sum_{k\geq 1}\nabla f(k)/k^z$. Remarkably, this function can be expressed in terms of the generalized Riemann (Hurwitz) ζ-function. Figure 14 shows the values of $\nabla f(k)$ broken up in a way that displays the pattern: the values for odd k go in the sequence $1,-1,1,-1,\cdots$; if those are removed, the odd values in the remaining sequence are $1,-1,1,-1,\cdots$; if those are removed, the odd values in the remaining sequence are $1,-1,1,-1,\cdots$; etc. (Proof: For $m>0$, the numbers $m\cdot 2^{n+2}+2^n$ and $m\cdot 2^{n+2}+2^n-1$ differ only in their last $(n+1)$ bits, so from the interpretation that $f(k)$ is the number of parity changes in the binary representation of k, we must have $\nabla f(m\cdot 2^{n+2}+2^n)=\nabla f(2^n)=1$ for all $m,n\geq 0$. (See discussion following (5).) A similar argument shows that $\nabla f(m\cdot 2^{n+2}+3\cdot 2^n)=\nabla f(3\cdot 2^n)=-1$ for all $m,n\geq 0$.) In terms of complex

k	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	26	27	28	29	30	31
∇f(k)	1	1	-1	1	1	-1	-1	1	1	1	-1	-1	1	-1	-1	1	-1	1	-1	1	1	-1	-1	-1	1	1	-1	-1	1	-1	-1
(k=2j+1)	1		-1		1		-1		1		-1		1		-1		1		-1		1		-1		1		-1		1		-1
(k=4j+2)		1				-1				1				-1				1				-1				1				-1	
(k=8j+4)				1								-1								1								-1			
(k=16j+8)								1																-1							
(k=32j+16)																1															

FIG. 14. *Decomposition of* $\nabla f(k)$.

functions this means that

$$\sum_{k\geq 1}\frac{\nabla f(k)}{k^z}=\left(\frac{1}{1^z}-\frac{1}{3^z}+\frac{1}{5^z}-\frac{1}{7^z}+\cdots\right)\left(\frac{1}{1^z}+\frac{1}{2^z}+\frac{1}{4^z}+\frac{1}{8^z}+\cdots\right).$$

Both of these series can be expressed in terms of classical functions of complex variables. The second is a simple geometric series:

$$\sum_{k\geq 0}\frac{1}{(2^k)^z}=\sum_{k\geq 0}\left(\frac{1}{2^z}\right)^k=\frac{1}{1-1/2^z}=\frac{2^z}{2^z-1}.$$

The first factor involves the generalized Riemann ζ-function, which is defined, for $\mathrm{Re}\,(z)>1$, by the equation

$$\zeta(z,a)=\sum_{n\geq 0}\frac{1}{(n+a)^z}.$$

Of course, we shall need to deal with the analytic continuation of this function, which is defined for all z except $z=1$, where there is a simple pole with residue 1. (The classical reference for properties of the ζ-function is Titchmarsh [14], though Whittaker and Watson [15] also have a full treatment, and Edwards [6] gives a nice historical perspective.) In terms of this function, we have

$$\begin{aligned}\sum_{k\geq 0}\frac{(-1)^k}{(2k+1)^z}&=2\sum_{k\geq 0}\frac{1}{(4k+1)^z}-\sum_{k\geq 1}\frac{1}{k^z}+\sum_{k\geq 1}\frac{1}{(2k)^z}\\&=\frac{2}{4^z}\zeta\left(z,\frac{1}{4}\right)-\frac{2^z-1}{2^z}\zeta(z,1).\end{aligned}$$

(It is customary to drop the second argument in $\zeta(z,1)$ and refer to it simply as $\zeta(z)$; this is the function originally studied by Riemann.) Therefore, we have found that

$$\sum_{k\geq 1}\frac{\nabla f(k)}{k^z}=\frac{2\zeta(z,1/4)}{2^z(2^z-1)}-\zeta(z). \tag{20}$$

It is the existence of this simple formula which makes the gamma-function method applicable to this problem. (Functions of this form are well-known in analytic number theory as Dirichlet series, and many techniques have been developed for dealing with them. See, for example, [2].)

Substituting, we have

$$b_j=\frac{1}{2\pi i}\int_{1-i\infty}^{1+i\infty}\Gamma(z)j^z\left(\frac{2\zeta(2z,1/4)}{4^z(4^z-1)}-\zeta(2z)\right)dz\,(1+O(j^{4\varepsilon-1})).$$

To evaluate this integral, we first approximate it by integrating around the contour R_Y shown in Fig. 15 and letting $Y\to\infty$. As before, as $Y\to\infty$ the integral along the right-hand side of R_Y approaches the given integral, and the integrals along the top, bottom and left can be bounded by using well-known bounds on the Γ and ζ functions. We have

$$|\Gamma(x+iy)|=O(|y|^{x-1/2}\,e^{-\pi|y|/2}), \tag{21}$$

which follows from Stirling's approximation (see, for example, [1, eq. 6.1.45]), and

$$|\zeta(x+iy,a)|=O(|y|^{1-x})\quad\text{for } x\geq -1 \tag{22}$$

(see, for example, Whittaker and Watson [15, p. 276]). Therefore, the integrals along

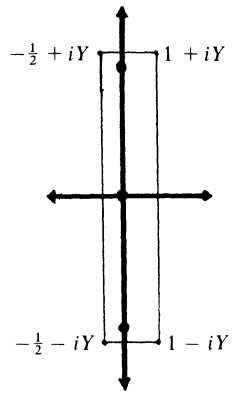


FIG. 15. *Contour of integral for final integral.*

the top and bottom lines of R_Y are

$$O\left(e^{-\pi|Y|/2}\int_{-1/2}^{1} j^x|Y|^{3/2-x}\,dx\right)$$

which is exponentially small and vanishes very quickly as $Y\to\infty$. The integral along the left line of R_Y is

$$O\left(j^{-1/2}\int_{-\infty}^{\infty} e^{-\pi|y|/2}|y|^2\,dy\right)=O(j^{-1/2})$$

so that we now have

$$b_j=\frac{1}{2\pi i}\int_{R_Y}\Gamma(z)j^z\left(\frac{2\zeta(2z,1/4)}{4^z(4^z-1)}-\zeta(2z)\right)dz(1+O(j^{4\varepsilon-1}))+O(j^{-1/2}). \tag{23}$$

The value of the integral is the sum of the residues within R_Y.

The only singularities within R_Y are contributed by $\Gamma(z)$ and $1/(4^z-1)$: the function $\Gamma(z)$ has a simple pole at $z=0$ with residue 1, and $1/(4^z-1)$ has simple poles at $z=2k\pi i/\ln 4$ for $k=0,\pm1,\pm2,\cdots$ with residue $1/\ln 4$. (Both $\zeta(2z,\frac{1}{4})$ and $\zeta(2z)$ have simple poles with residue 1 at $z=1/2$, but they cancel out.) There is therefore a double pole at $z=0$ and we need to use Laurent series expansions to find the residue there. We have

$$\Gamma(z)=\frac{\Gamma(z+1)}{z}=\frac{1}{z}\exp\{-\gamma z+O(z^2)\}=\frac{1}{z}-\gamma+O(z),$$

$$j^z=e^{z\ln j}=1+z\ln j+O(z^2),$$

$$\frac{1}{4^z}=e^{z\ln(1/4)}=1-z\ln 4+O(z^2),$$

$$\frac{1}{4^z-1}=\frac{1}{e^{z\ln 4}-1}=\frac{1}{z\ln 4+(z\ln 4)^2/2+O(z^3)}=\frac{1}{z\ln 4}-\frac{1}{2}+O(z),$$

$$\zeta\left(2z,\frac{1}{4}\right)=\frac{1}{4}+z\left(2\ln\Gamma\left(\frac{1}{4}\right)-\ln(2\pi)\right)+O(z^2),$$

and

$$\zeta(2z)=-\frac{1}{2}+O(z).$$

The expansion for $\Gamma(z)$ is well known (see Abramowitz and Stegun [1, eq. 6.1.33]), and the next three expansions are elementary. The expansions for the ζ-functions, which are crucial to the derivation, follow directly from Whittaker and Watson [15, p. 271] where it is shown that $\zeta(0, a)=\frac{1}{2}-a$ and $\zeta'(0, a)=\ln\Gamma(a)-\frac{1}{2}\ln(2\pi)$. Multiplying these series together, we find that

$$\Gamma(z)j^z\left(\frac{2\zeta(2z, 1/4)}{4^z(4^z-1)}-\zeta(2z)\right)=\left(\frac{1}{z}+\ln j-\gamma+O(z)\right)\left(\frac{1}{4z\ln 2}-\frac{1}{4}+\lg\frac{\Gamma(1/4)^2}{2\pi}+O(z)\right)$$

$$=\frac{1}{4z^2\ln 2}+\frac{1}{z}\left(\frac{1}{4}\lg j-\frac{\gamma}{4\ln 2}-\frac{1}{4}+\lg\frac{\Gamma(1/4)^2}{2\pi}\right)+O(1). \tag{24}$$

This gives the residue at $z=0$ (the coefficient of $1/z$).

To this we must add the residue at the other poles of $1/(4^z-1)$. The effect of these other terms is small (but not insignificant), and we shall encapsulate them in a single term,

$$\begin{aligned}\varepsilon(j)&\equiv\frac{2}{\ln 4}\sum_{k\neq 0}\Gamma\left(\frac{2k\pi i}{\ln 4}\right)j^{2k\pi i/\ln 4}\zeta\left(\frac{4k\pi i}{\ln 4}, \frac{1}{4}\right)\\&=\frac{1}{\ln 2}\sum_{k\geq 1}2\,\mathrm{Re}\left(\Gamma\left(\frac{k\pi i}{\ln 2}\right)j^{k\pi i/\ln 2}\zeta\left(\frac{2k\pi i}{\ln 2}, \frac{1}{4}\right)\right)\\&=\sum_{k\geq 1}(\xi_k\cos(k\pi\lg j)-\eta_k\sin(k\pi\lg j)),\end{aligned}$$

where

$$\frac{2}{\ln 2}\Gamma\left(\frac{k\pi i}{\ln 2}\right)\zeta\left(\frac{2k\pi i}{\ln 2}, \frac{1}{4}\right)\equiv\xi_k+i\eta_k.$$

To finish the evaluation of our b_j and B_N we need to estimate the Γ and ζ functions at these points along the imaginary axis. The Γ-function is easy to bound from Stirling's approximation (see Edwards [5, § 6.3]), and the ζ-function can be estimated by writing

$$\zeta(z, a)=\sum_{0\leq k<K}\frac{1}{(k+a)^z}+\sum_{k\geq K}\frac{1}{(k+a)^z},$$

and then applying Euler–MacLaurin summation to the second sum, for appropriate K. (These manipulations are valid for $\mathrm{Re}\,z>1$ only, but the resulting formulas are valid for all z, by analytic continuation—see Edwards [6, § 6.4] for details.) Table 4 shows the values of ξ_k and η_k for $k=1, 2, 3$ computed in this way. The values get exceedingly small for larger k, as can be verified from the bounds (21) and (22).

Adding all the residues, we have, from (23):

$$b_j=\frac{1}{4}\lg j+\lg\frac{\Gamma(1/4)^2}{2\pi}-\frac{1}{4}-\frac{\gamma}{4\ln 2}+\varepsilon(j)+O(j^{-1/2}). \tag{25}$$

This leads to our final result.

THEOREM 3. *The average number of exchanges used by Batcher's odd-even merge for a random 2-ordered file of $2N$ elements is*

$$B_N=\frac{1}{4}N\lg N+\left(\lg\frac{\Gamma(1/4)^2}{2\pi}+\frac{1}{4}-\frac{\gamma+2}{4\ln 2}+\delta(N)\right)N+O(\sqrt{N}\log N),$$

where $\delta(N)$ is a periodic function of $\log N$, with $\delta(4N)=\delta(N)$, $|\delta(N)|<.000490$, and

$\delta(2^n) \approx .000317(-1)^n$. (*The constant*

$$\lg \frac{\Gamma(1/4)^2}{2\pi} + \frac{1}{4} - \frac{\gamma+2}{4 \ln 2}$$

has the approximate value .385417224.)

Proof. From the discussion above, we need only substitute our result (25) for b_j into our equation (14) for B_N and perform the summation. We have

$$B_N = \frac{1}{2} \frac{4^N}{\binom{2N}{N}} \sum_{1 \le j < N} \frac{\binom{2j}{j}}{4^j} \left(\frac{1}{4} \lg j + \lg \frac{\Gamma(1/4)^2}{2\pi} - \frac{1}{4} - \frac{\gamma}{4 \ln 2} + \varepsilon(j) + O(j^{-1/2}) \right) + \frac{1}{2} N.$$

The terms not involving j are easily taken care of, since it is trivial to prove by induction that

$$\frac{1}{2} \frac{4^N}{\binom{2N}{N}} \sum_{0 \le j < N} \frac{\binom{2j}{j}}{4^j} = N.$$

(Direct proof:

$$\sum_{0 \le j < N} \frac{1}{4^j} \binom{2j}{j} = \sum_{0 \le j < N} (-1)^j \binom{-1/2}{j} = (-1)^{N-1} \binom{-3/2}{N-1} = \binom{N-1/2}{N-1} = \frac{N}{4^{N-1}} \binom{2N-1}{N-1};$$

for supporting identities, see Knuth [12].)

For the other terms, we can remove the binomial coefficients with Stirling's approximation, as in the derivation of (16). We have

$$\frac{\binom{2j}{j}}{4^j} = \frac{1}{\sqrt{\pi j}} + O(j^{-3/2}) \quad \text{and} \quad \frac{4^N}{\binom{2N}{N}} = \sqrt{\pi N} + O(N^{-1/2}).$$

TABLE 4

Values of constants in the asymptotic expansion for the average number of exchanges.

$$\xi_k + i\eta_k = \frac{2}{\ln 2} \Gamma\left(\frac{k\pi i}{\ln 2}\right) \zeta\left(\frac{2k\pi i}{\ln 2} \frac{1}{4}\right)$$

k	ξ_k	η_k
1	.003704670+	.002500177+
2	.000001560+	−.000000832−
3	.000000001−	.000000002+

$$\Gamma\left(\frac{1}{4}\right) = 3.6256099082+$$

$$\frac{1}{\ln 2} = 1.4426950408+$$

$$\gamma = 0.5772156649+$$

$$\pi = 3.1415926535+$$

Therefore the $O(j^{-1/2})$ term sums to $O(\sqrt{N}\log N)$, and

$$\frac{1}{2}\frac{4^N}{\binom{2N}{N}}\sum_{1\le j<N}\frac{\binom{2j}{j}}{4^j}\frac{1}{4}\lg j=\frac{\sqrt{N}}{8}\sum_{1\le j<N}\frac{\lg j}{\sqrt{j}}+O(\sqrt{N})$$

$$=\frac{\sqrt{N}}{8}\int_1^N\frac{\lg x}{\sqrt{x}}\,dz+O(\sqrt{N})$$

$$=\frac{\sqrt{N}}{2\ln 2}\int_1^{\sqrt{N}}\ln y\,dy+O(\sqrt{N})$$

$$=\frac{1}{4}N\lg N-\frac{1}{2\ln 2}N+O(\sqrt{N}).$$

Here the second step follows from Euler–MacLaurin summation (see, for example, Knuth [13, p. 110]) and the third step from the substitution $x=y^2$.

We have proved that

$$B_N=\frac{1}{4}N\lg N+\left(\lg\frac{\Gamma(1/4)^2}{2\pi}+\frac{1}{4}-\frac{\gamma+2}{4\ln 2}+\delta(N)\right)N+O(\sqrt{N}\log N);$$

it remains to evaluate the oscillatory term

$$N\delta(N)=\frac{1}{2}\frac{4^N}{\binom{2N}{N}}\sum_{1\le j<N}\frac{\binom{2j}{j}}{4^j}\varepsilon(j).$$

After substituting for $\varepsilon(j)$, we proceed in the same way as we did for the $\lg j$ term. The result of using Stirling's approximation on the binomial coefficients and Euler–MacLaurin summation on the resulting sums is

$$\delta(N)=\frac{1}{2\sqrt{N}}\sum_{k\ge 1}\left(\xi_k\int_1^N\frac{\cos(k\pi\lg x)}{\sqrt{x}}\,dx-\eta_k\int_1^N\frac{\sin(k\pi\lg x)}{\sqrt{x}}\,dx\right)+O\left(\frac{1}{\sqrt{N}}\right).$$

These integrals are elementary; the substitutions $x=y^2$, then $t=2\pi k\lg y$, transform them into standard integrals (for example, Abramowitz and Stegun [1, eqs. 4.3.136, 4.3.137]) with the eventual result

$$\delta(N)=\sum_{k\ge 1}\frac{\sigma_k}{\sigma_k^2+1}(\xi_k(\sigma_k\cos(\pi k\lg N)+\sin(\pi k\lg N))$$

$$-\eta_k(\sigma_k\sin(\pi k\lg N)-\cos(\pi k\lg N)))$$

where σ_k is $(\ln 2)/(2\pi k)$. From this formula, we see that $\delta(N)$ has the stated properties. With the aid of Table 4, we can easily compute the values

$$\delta(2^{2^n})=\sum_{k\ge 1}\frac{\sigma_k}{\sigma_k^2+1}(\sigma_k\xi_k+\eta_k)\approx .000317000\cdots \tag{26}$$

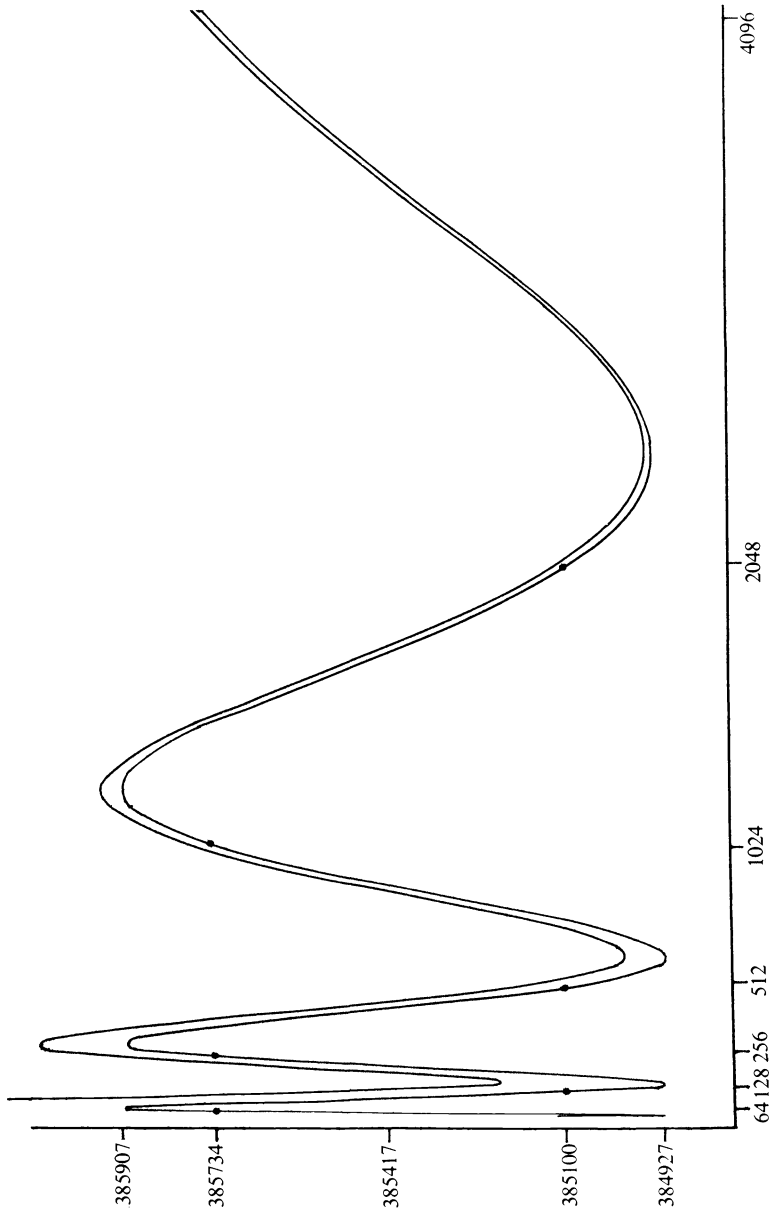


FIG. 16. *Coefficient of the linear term.*

and

$$\delta(2^{2n+1}) = \sum_{k\geq 1} (-1)^k \frac{\sigma_k}{\sigma_k^2+1}(\sigma_k\xi_k+\eta_k) \approx -.000317082\cdots. \tag{27}$$

Finally, since the function $a \sin x + b \cos x$ has the extreme values $\pm\sqrt{a^2+b^2}$, we have the extreme values

$$|\delta(N)| \leqq \sum_{k\geq 1} \frac{\sigma_k}{\sigma_k^2+1}\sqrt{(\xi_k\sigma_k+\eta_k)^2+(\xi_k-\eta_k\sigma_k)^2} \approx .000490177. \quad \square$$

From Theorem 3 we see that, asymptotically, only 1/4 of the comparators in Batcher's merge are involved in exchanges, on the average. The analytic result for the coefficient of the linear term given in Theorem 3 matches the exact computed value (Table 3) to six decimal places.

In principle we could extend the methods used to get any desired accuracy whatsoever. This would mainly involve carrying the asymptotic series expansions further, which gets very complicated in the applications of Euler–MacLaurin summation. Also, the left boundary has to be moved left for sharper asymptotic accuracy in (23). Each negative integer enclosed contributes another simple pole from the Γ function.

Figure 16 shows a graph of the coefficient of the linear term from Theorem 3,

$$\lg\frac{\Gamma(1/4)^2}{2\pi}+\frac{1}{4}-\frac{\gamma+2}{4\ln 2}+\delta(N),$$

together with the true values of $(B_N - \frac{1}{4}N \lg N)/N$, computed with (13). The upper curve is the actual values, and the lower curve is the asymptotic estimate. The difference between the curves is reflected in the $O(\sqrt{N}\log N)$ term in Theorem 3. The curves get very close for large N.

4. Sorting. Any merging method may be extended into a sorting method with the following recursive procedure: To sort a file of N elements, use the method to independently sort the odd elements and the even elements of the file, thus producing a 2-ordered file of N elements. Then apply the merging method. Figure 17 shows the sorting network resulting from applying this procedure to Batcher's odd-even merge. If merge stages are overlapped, the sort can be accomplished in $\frac{1}{2}\lceil \lg N\rceil$ ($\lceil \lg N\rceil + 1$) independent stages. Knuth gives a formula describing the number of comparators required [13, exercise 5.2.2-15]; it depends heavily on the binary representation of N. For simplicity, we shall assume throughout this section that $N = 2^n$. The number of comparators required is then described by the relation (see (1))

$$C^*_{2^n} = 2C^*_{2^{n-1}} + (n-1)2^{n-1} + 1$$

which telescopes, after division by 2^n, to the solution

$$C^*_N = \frac{1}{4}N(\lg N)^2 - \frac{1}{4}N\lg N + N - 1, \qquad N = 2^n. \tag{28}$$

Again, this method cannot compete with known $O(N \log N)$ sorting methods on serial computers, but it might do well if parallelism is available.

The average number of exchanges required can be calculated from a similar recurrence, using Theorem 3, since the odd and even elements are sorted independently. If

$$\alpha \equiv \lg \frac{\Gamma(1/4)^2}{2\pi} + \frac{1}{4} - \frac{\gamma+2}{4\ln 2}$$

we have the following expression for the average number of exchanges:

$$B^*_{2^n} = 2B^*_{2^{n-1}} + \left(\frac{1}{4}(n-1) + \alpha + \delta(2^{n-1})\right)2^{n-1} + O(\sqrt{N}\log N).$$

Iterating this recurrence (applying the same recurrence to $B^*_{N/2}$), we get

$$B^*_{2^n} = 4B^*_{2^{n-2}} + \left(\frac{1}{2}n + 2\alpha - \frac{3}{4} + \delta(2^{n-2}) + \delta(2^{n-1})\right)2^{n-1} + O(\sqrt{N}\log N).$$

If we define $\delta^*(2^n) \equiv \delta(2^{n-2}) + \delta(2^{n-1})$, then we know that $\delta^*(2^n) = \delta^*(2^{n-2})$ as in Theorem 3. Our recurrence then telescopes when divided by 2^n to the solution

$$\begin{aligned}\frac{B^*_{2^n}}{2^n} &= \sum_{0\le j\le n/2}\left(\frac{1}{4}(n-2j) + \alpha - \frac{3}{8} + \frac{1}{2}\delta^*(2^n)\right) + O(1)\\ &= \frac{n^2}{16} + \frac{1}{2}\left(\alpha - \frac{1}{8} + \delta^*(2^n)\right)n + O(1)\end{aligned}$$

or, in terms of N:

$$B^*_N = \frac{1}{16}N(\lg N)^2 + \frac{1}{2}\left(\alpha - \frac{1}{8} + \delta^*(N)\right)N\lg N + O(N), \qquad N = 2^n. \tag{29}$$

The value of $\frac{1}{2}(\alpha - \frac{1}{8})$ is about $.130208\cdots$ which is the value of the coefficient of the $N \lg N$ term to six places, since we know from (26) and (27) that $|\delta^*(N)| < 10^{-6}$.

In the same way, we could find from Theorem 1 that, asymptotically, all of the comparators could be involved in exchanges in the worst case. However, this asymptotic maximum is approached even more slowly than for the merging method, since the recursive nature of the sorting method guarantees that many small files will be merged.

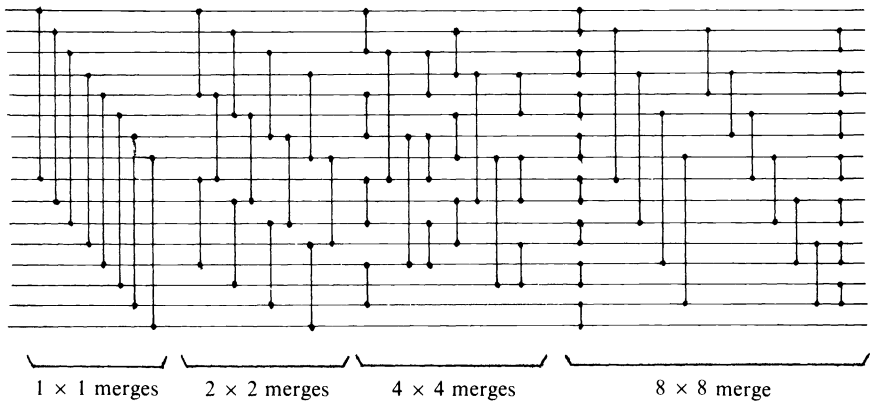


FIG. 17. *Odd-even sorting network.*

5. Conclusion. In this paper, we have derived formulas which accurately describe the number of exchanges involved in Batcher's odd-even merge, both on the average and in the worst case. This completes our understanding of Batcher's method, which is of some theoretical importance as a near-optimal nonadaptive method, and of some practical importance when parallelism is available.

The main results are the exact formulas for the worst case and the average given in Theorems 1 and 3. These lead to asymptotic statements that, as $N \to \infty$, about $\frac{1}{4}$ of the comparators do exchanges on the average and nearly all of them do exchanges in the worst case.

We have emphasized the methods of analysis, as well as the results, because they may have more general applicability. In particular, Theorem 2 could be of use in the analysis of other merging problems and other combinatorial problems which can be modeled with paths in a lattice. Also, the problem of determining the average number of exchanges has provided an excellent example of the application of de Bruijn's "gamma-function" method of asymptotic analysis.

Acknowledgments. I had thought this problem hopelessly difficult until Dave Notkin brought its details to my attention in a classroom project.

Note. The kind of asymptotic analysis that we used in determining the average number of exchanges has recently been used to solve yet another problem: determining the average number of registers needed to evaluate arithmetic expressions. See the recent reports by P. Flajolet, J. C. Raoult and J. Vuillemin, *On the average number of registers required for evaluating arithmetic expressions*, Proc. 18th Symp. on Foundations of Computer Science, Providence, RI; and by R. Kemp, *The average number of registers needed to evaluate a binary tree optimally*, Saarbrücken University Report A 77104, Saarbrücken, Germany.

REFERENCES

[1] M. ABRAMOWITZ AND I. A. STEGUN, *Handbook of Mathematical Functions*, Dover, New York, 1970.
[2] T. M. APOSTOL, *Introduction to Analytic Number Theory*, Springer-Verlag, New York, 1976.
[3] K. E. BATCHER, *A new internal sorting method*, Rep. GER 11759, Goodyear Aerospace Corp., Akron, OH, 1964.
[4] ———, *Sorting networks and their applications*, Proc. 1968 Spring Joint Comp. Conf., AFIPS Press, Montvale, NJ, 1968, pp. 307–314.
[5] N. G. DE BRUIJN, D. E. KNUTH AND S. O. RICE, *The average height of planted plane trees*, Graph Theory and Combinatorics, R. C. Read, ed., Academic Press, New York, 1972, pp. 15–22.
[6] H. M. EDWARDS, *Riemann's Zeta Function*, Academic Press, New York, 1974.
[7] A. ERDÉLYI, ed., *Tables of Integral Transforms*, McGraw-Hill, New York, 1954.
[8] H. W. GOULD AND J. KAUCKÝ, *Evaluation of a class of binomial coefficient summations*, J. Combinatorial Theory, 1 (1966), pp. 233–247.
[9] J. L. w. V. JENSEN, *Sur une identité d'Abel et sur d'autres formules analogues*, Acta Math., 26 (1902), pp. 307–318.
[10] K. KNOPP, *Theory of Functions Part I*, Dover, New York, 1945.
[11] ———, *Theory of Functions Part II*, Dover, New York, 1945.
[12] D. E. KNUTH, *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.
[13] ———, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1972.
[14] E. C. TITCHMARSH, *The Theory of the Riemann Zeta-Function*, Clarendon Press, Oxford, 1951.
[15] E. T. WHITTAKER AND G. N. WATSON, *A Course of Modern Analysis*, Cambridge University Press, London, 1973.

SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0002 $01.00/0

ISOMORPHISM TESTING FOR GRAPHS, SEMIGROUPS, AND FINITE AUTOMATA ARE POLYNOMIALLY EQUIVALENT PROBLEMS*

KELLOGG S. BOOTH†

Abstract. Two problems are polynomially equivalent if each is polynomially reducible to the other. The problems of testing either two graphs, two semigroups, or two finite automata for isomorphism are shown to be polynomially equivalent. For graphs the isomorphism problem may be restricted to regular graphs since we show that this is equivalent to the general case. Using the techniques of Hartmanis and Berman we then show that this equivalence is actually a polynomial isomorphism. It is conjectured that the isomorphism problem for groups is not in this equivalence class, but that it is an easier problem. If the conjecture is true then $P \neq NP$; if it is false then there exists a "subexponential" $O(n^{c_1 \log n + c_2})$ algorithm for graph isomorphism.

Key words. graph, regular graph, group, semigroup, finite automaton, isomorphism, polynomial reduction, *NP*-complete

1. Introduction. Determining the exact computational complexity of graph isomorphism is currently an open problem [11]. No algorithm has been proven to run in less than exponential time yet no nontrivial lower bound has been proven. Many problems of this type have been shown *NP*-complete but at present graph isomorphism is not among them [1], [6]. We investigate the class of problems which are "complete" over a graph isomorphism in the sense that any such problem will be polynomially equivalent to the problem of graph isomorphism. The notion of equivalence we use is mutual reducibility in the sense of either Karp or Cook [1], although we also show that the equivalence holds under the stronger notion of polynomial isomorphism discussed by Hartmanis and Berman [4].

After defining these basic concepts we introduce three isomorphism problems: isomorphism of graphs, isomorphism of semigroups, and isomorphism of finite automata. We show that testing graph isomorphism is no harder than testing regular graph isomorphism. We then extend the techniques to prove that the three isomorphism problems are in fact equivalent.

We conclude the discussion with a conjecture relating the problem of group isomorphism to the previous problems and to recent results of Tarjan [12] and Miller [10]. The implications of this conjecture to the $P = NP$ question are examined.

2. Basic terminology. The concept of polynomial reducibility has become familiar within the literature [1], [6]. A problem (language) L_1 is *polynomially reducible* to a problem (language) L_2 if there exists an encoding $\mathscr{E}$, computable in polynomial time, from strings over the alphabet of L_1 to strings over the alphabet of L_2 such that $w \in L_1$ iff $\mathscr{E}(w) \in L_2$. This is written as $L_1 \propto_p L_2$. If $L_1 \propto_p L_2$ and $L_2 \propto_p L_1$ we say that the problems are *polynomially equivalent* and write $L_1 \equiv_p L_2$; if the mapping $\mathscr{E}$ is a bijection and $\mathscr{E}^{-1}$ is a polynomial reduction of L_2 to L_1 we say that the problems are *polynomially isomorphic* and write $L_1 \cong_p L_2$ [4].

A graph $G = (V, E)$ is a set of *vertices* V and *edges* E such that each edge is a pair of vertices. A graph is *regular of degree r* if every vertex belongs to exactly r edges. K_n is the *complete graph* on n vertices, in which every vertex belongs to an edge with every other vertex. $K_{n,n}$ is the *complete bipartite graph* on $2n$ vertices. Two graphs

* Received by the editors March 14, 1977, and in revised form October 25, 1977.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.
This research was supported by the National Research Council of Canada under Grant A4307.

$G_1=(V_1, E_1)$ and $G_2=(V_2, E_2)$ are *isomorphic* if there exists a bijection $f: V_1 \to V_2$ such that $(u, v)\in E_1$ iff $(f(u), f(v))\in E_2$. This is written $G_1 \cong G_2$. The *graph isomorphism problem* is to decide, given two graphs G_1 and G_2, if such a bijection exists. Although many algorithms have been proposed [11] there is no known polynomial or even subexponential algorithm for deciding this question.

A semigroup $S=(X, \circ)$ is a set X having a single binary operation such that $x \circ (y \circ z)=(x \circ y)\circ z$ for all $x, y, z \in X$ ($\circ$ is an associative operation). The semigroup is commutative if $x \circ y = y \circ x$ for all $x, y \in X$. Two semigroups $S_1=(X_1, \circ_1)$ and $S_2=(X_2, \circ_2)$ are isomorphic if there exists a bijection $f: X_1 \to X_2$ such that $x \circ_1 y = z$ iff $f(x)\circ_2 f(y)=f(z)$. This is written $S_1 \cong S_2$. The *semigroup isomorphism problem* is to determine if such a bijection exists. More information on semigroups is given in [3].

A *finite automaton* $A=(Q, \Sigma, \delta, s, F)$ is a set of *states* Q, an *alphabet* Σ, a *transition function* $\delta: Q\times\Sigma \to Q$, *initial state* $s \in Q$, and a set of *final states* $F \subseteq Q$. Two finite automata $A_1=(Q_1, \Sigma_1, \delta_1, s_1, F_1)$ and $A_2=(Q_2, \Sigma_2, \delta_2, s_2, F_2)$ are *isomorphic* if there exist bijections $f: Q_1 \to Q_2$ and $g: \Sigma_1 \to \Sigma_2$ such that

1) $f(\delta_1(q, a))=\delta_2(f(q), g(a))$ for all $q \in Q_1$ and $a \in \Sigma_1$,

2) $f(s_1)=s_2$,

and

3) $q \in F_1$ iff $f(q)\in F_2$ for all $q \in Q_1$.

The *finite automaton isomorphism problem* is to determine if two such bijections exist [8]. More information on finite automata is given in [3].

3. Graphs. Although some classes of graphs such as trees [1] and planar graphs [5] are easy to test for isomorphism there are many well-known classes of graphs for which the isomorphism problem is no easier than for the most general case [2]. One such class having this property is the class of regular graphs.

THEOREM 1. *Regular graph isomorphism* $\equiv_p$ *graph isomorphism.*

Proof. Any isomorphism test for arbitrary graphs will also work for regular graphs. We need only show that graph isomorphism $\propto_p$ regular graph isomorphism.

Let $G=(V, E)$ be any graph having $V=\{v_i | 1 \leqq i \leqq n\}$ and $E=\{e_i | 1 \leqq j \leqq m\}$ where every vertex belongs to at least one edge and $m-n>2$. Define the following sets.

$$V_1=\{f_j | 1 \leqq j \leqq m\},$$

$$V_2=\{g_k | 1 \leqq k \leqq m-2\},$$

$$V_3=\{h_l | 1 \leqq l \leqq m-n+2\}$$

and

$$E_1=\{\{v_i, e_j\} | v_i \in e_j,\ 1 \leqq i \leqq n \text{ and } 1 \leqq j \leqq m\},$$

$$E_2=\{\{v_i, f_j\} | v_i \notin e_j,\ 1 \leqq i \leqq n \text{ and } 1 \leqq j \leqq m\},$$

$$E_3=\{\{e_j, g_k\} | 1 \leqq j \leqq m \text{ and } 1 \leqq k \leqq m-2\},$$

$$E_4=\{\{f_j, h_l\} | 1 \leqq j \leqq m \text{ and } 1 \leqq l \leqq m-n+2\}.$$

Let REGULAR(G) be the graph $(V \cup E \cup V_1 \cup V_2 \cup V_3, E_1 \cup E_2 \cup E_3 \cup E_4)$. We can establish two facts about REGULAR(G): it is a regular graph of degree m and given REGULAR(G) we can recover G uniquely.

The first fact is easily verified. Each $v_i \in V$ has degree m in REGULAR(G) because it is adjacent to either e_j or f_j for all $1 \leqq j \leqq m$; each $e_j \in E$ has degree m because it is adjacent to exactly 2 of the $v_i \in V$ and to all $m-2$ of the $g_k \in V_2$; each $f_j \in V_1$ is adjacent to exactly $n-2$ of the $v_i \in V$ and also to all $m-n+2$ of the $h_l \in V_3$;

each $g_k \in V_2$ is adjacent to all m of the $e_j \in E$; finally each $h_l \in V_3$ is adjacent to all m of the $f_j \in V_1$.

The second fact follows from the observation that in REGULAR(G) every $g_k \in V_2$ has exactly the same set of neighbors and every $h_l \in V_3$ has exactly the same set of neighbors. We can tell these two sets apart because $|V_2| > |V_3|$ since $n > 4$ if $m - n > 2$ in a graph. Having thus located V_2, we know that

$$E = \{\text{vertices at distance 1 from } V_2\},$$

$$V = \{\text{vertices at distance 2 from } V_2\}$$

and also that $\{u, v\} \in E$ iff there is an edge in REGULAR(G) from both u and v to some $e_j \in E$. The encoding $(G_1, G_2) \to (\text{REGULAR}(G_1), \text{REGULAR}(G_2))$ thus has the property that $G_1 \cong G_2$ iff $\text{REGULAR}(G_1) \cong \text{REGULAR}(G_2)$. Moreover, it is clearly computable in polynomial time and hence is a polynomial reduction of graph isomorphism to regular graph isomorphism if we realize that isolated vertices can be handled with a simple pre-test and that adding an equal number of copies of K_4 to both G_1 and G_2 will not affect their isomorphism but will ensure that $m - n > 2$, without increasing the size of the input by more than a polynomial. Q.E.D.

Similar constructions exist which show that graph isomorphism and directed graph isomorphism are polynomially equivalent [2], [10]. We will thus make no distinction between the two problems (indeed, our definition of graph conveniently glossed over any distinction).

Having seen that for at least one subclass of graphs the isomorphism problem is still equivalent to the general case, we should not be surprised to find many others. A survey of classes having this property is contained within [2]. What is of more interest is to find substantially "different" problems which are still equivalent. The next two sections provide examples of such problems, although both are admittedly still isomorphism problems and thus maybe not too "different."

4. Semigroups. Miller and Monk [9] have shown that the isomorphism problem for any algebraic structure is polynomially reducible to graph isomorphism because the operation tables can be encoded as directed graphs. Conversely, if we have an algorithm for handling isomorphism of arbitrary algebraic structures it will also work for graphs if we consider the adjacency relation to be a binary operation [2]. Thus the general algebraic structure isomorphism problem is polynomially equivalent to the graph isomorphism problem. When axioms are added to the algebras, it is no longer obvious that an arbitrary graph can be encoded as an algebra. We can ask for which axioms the algebraic isomorphism problem is polynomially equivalent to the graph isomorphism problem.

Tarjan has given an $O(n^{c_1 \log n + c_2})$ algorithm for testing isomorphism of groups [12]. Miller then extended the algorithm to work for quasigroups and Latin squares while only affecting the constants c_1 and c_2 [10]. Algorithms of this time complexity are "subexponential" in the sense that c^n is not $O(n^{c_1 \log n + c_2})$ for any constants c_1 and c_2 when $c > 1$. Thus a bound of this type would be an improvement over any known bound for existing graph isomorphism algorithms.

Although we believe that graph isomorphism is not an *NP*-complete problem, we also believe that it is not as easy as group isomorphism. As a motivation for this remark we prove the following.

THEOREM 2. *Semigroup isomorphism* $\equiv_p$ *graph isomorphism.*

Proof. In light of Miller and Monk's result it is sufficient to show that graph isomorphism $\propto_p$ to semigroup isomorphism.

Let $G=(V, E)$ be any graph. Define SEMIGROUP(G) to be $(V \cup E \cup \{0\}, \circ)$ where $\circ$ is the binary operation defined by

$$x \circ y = y \circ x = \begin{cases} x & \text{if } x = y, \\ y & \text{if } x \in y \in E, \\ \{x, y\} & \text{if } \{x, y\} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

We can observe the following two facts about SEMIGROUP(G): it is a semigroup and given SEMIGROUP(G) we can recover G uniquely.

Associativity is easily verified. The second fact follows from the observation that 0 is the unique element with the property that $x \circ 0 = 0 \circ x = 0$ for all x, and that

$$E = \{x \mid x \circ y = x \text{ for exactly two } y \neq x\}.$$

All other elements must be in V. The rest of the proof is similar to that of Theorem 1. We conclude that the encoding $(G_1, G_2) \to$ (SEMIGROUP(G_1), SEMIGROUP(G_2)) is a polynomial reduction of graph isomorphism to semigroup isomorphism. Q.E.D.

The construction used in Theorem 2 actually proves a stronger result. The binary operation defined there is obviously commutative.

THEOREM 3. *Commutative semigroup isomorphism* $\equiv_p$ *graph isomorphism.*

The primary difference between semigroups and either groups or quasigroups is the ability to solve equations. This property plays a central role in the algorithms of both Tarjan and Miller. The construction in Theorem 2 relies heavily upon the fact that we could allow $x \circ y = 0$ for *any* values of x and y. The constraint of associativity (or commutativity) was not critical, but the ability to solve equations seems to be a luxury we can't have if we want to encode arbitrary graphs. The conjecture that group isomorphism is intrinsically easier than semigroup isomorphism (and thus, by Theorem 2, graph isomorphism) is based upon the belief that the axioms for groups (or even quasigroups) drastically reduce the number of possible isomorphisms and thus the inherent complexity of the isomorphism problem.

5. Finite automata. The relationship between semigroups and automata is well known [3], so a polynomial reduction appears trivial, but the familiar construction from a semigroup yields a *semiautomaton* not an automaton, because no start or final states are defined [3]. If we are interested in automata with start states we must do a little more work to prove equivalence.

THEOREM 4. *Finite automaton isomorphism* $\equiv_p$ *graph isomorphism.*

Proof. The Miller–Monk result extends also to finite automata so we need only show that finite automaton isomorphism $\propto_p$ graph isomorphism.

Let $G=(V, E)$ be any graph. Define AUTOMATON(G) to be $(V \cup \{\text{"}start\text{"}, \text{"}stop\text{"}\}, V, \delta, \text{"}start\text{"}, V)$ where "*start*" and "*stop*" are special states. The transition function δ is given by

$$\delta(\text{"}start\text{"}, v) = v,$$

$$\delta(\text{"}stop\text{"}, v) = \text{"}stop\text{"},$$

$$\delta(u, v) = \begin{cases} u & \text{if } u = v, \\ v & \text{if } \{u, v\} \in E, \\ \text{"}stop\text{"} & \text{otherwise.} \end{cases}$$

(Strictly speaking the alphabet should be a copy of V, but we ignore this technicality.)

Again we notice two things about the construction: AUTOMATON(G) is a finite automaton and given AUTOMATON(G) we can recover G uniquely. Thus the encoding $(G_1, G_2) \to$ (AUTOMATON(G_1), AUTOMATON(G_2)) is a polynomial reduction of graph isomorphism to finite automaton isomorphism. Q.E.D.

The construction encodes the graph structure using the state set and the alphabet. We might ask if this is necessary. Leiss has pointed out that if we fix the size of the alphabet, automaton isomorphism becomes polynomial if we assume all states are reachable, as they are here [7]. A similar result holds if we fix the size of the state set. We can say more, however, in the case of semiautomata.

Ignoring the initial and final state designations of an automaton we have a semiautomaton. The usual construction of a semiautomaton from a semigroup uses the semigroup for both the state set and the alphabet. This is not necessary for our purposes, and we can restrict ourselves to a binary alphabet.

THEOREM 5. *Semiautomaton isomorphism* $\equiv_p$ *graph isomorphism, even if we restrict the alphabet to be of size two.*

Proof. As before we prove only that graph isomorphism $\propto_p$ semiautomaton isomorphism. This time, however, we choose the directed version of graph isomorphism.

Given a graph $G = (V, E)$ define SEMIAUTOMATON(G) to be $(V \cup E \cup \{\text{“}dead\text{”}\}, \{\text{“}in\text{”}, \text{“}out\text{”}\}, \delta)$, where “*dead*”, “*in*”, and “*out*” are new objects and the transition function is given by

$$\delta(q, \text{“}in\text{”}) = \begin{cases} q & \text{if } q \in V, \\ u & \text{if } q \in E \text{ and } q = (u, v), \\ \text{“}dead\text{”} & \text{otherwise,} \end{cases}$$

$$\delta(q, \text{“}out\text{”}) = \begin{cases} v & \text{if } q \in E \text{ and } q = (u, v), \\ \text{“}dead\text{”} & \text{otherwise.} \end{cases}$$

The usual arguments show that the mapping $(G_1, G_2) \to$ (SEMIAUTOMATON(G_1), SEMIAUTOMATON(G_2)) is a polynomial reduction of graph isomorphism to semiautomaton isomorphism. Q.E.D.

6. Polynomial isomorphism. Hartmanis and Berman have examined the equivalence relation defined by polynomial isomorphism [4]. It is a refinement of the polynomial equivalence we have been discussing. They have observed that all known *NP*-complete problems are actually polynomially isomorphic [4]. We prove a similar result here by showing that all of the problems discussed above are polynomially isomorphic. We use a technical result from [4] to establish our claim.

LEMMA 6. *If* L_1 *and* L_2 *are languages over the alphabets* Σ_1 *and* Σ_2, *respectively, and there exist polynomial time computable functions* $\mathscr{E}_1: \Sigma_1^* \times \Sigma_1^* \to \Sigma_1^*$, $\mathscr{D}_1: \Sigma_1^* \to \Sigma_1^*$, $\mathscr{E}_2: \Sigma_2^* \times \Sigma_2^* \to \Sigma_2^*$, *and* $\mathscr{D}_2: \Sigma_2^* \to \Sigma_2^*$ *such that for* $i = 1, 2$

1) $\forall x, y \in \Sigma_i^*$, $\mathscr{E}_i(x, y) \in L_i$ *iff* $x \in L_i$,
2) $\forall x, y \in \Sigma_i^*$, $\mathscr{D}_i(\mathscr{E}_i(x, y)) = y$,
3) $\forall x, y \in \Sigma_i^*$, $|\mathscr{E}_i(x, y)| > |x| + |y|$,

then $L_1 \cong_p L_2$ *iff* $L_1 \equiv_p L_2$.

The function $\mathscr{E}_i$ is used to encode the language L_i into itself by “padding” the strings to an appropriate length. $\mathscr{D}_i$ is a decoding function which retrieves the second argument of $\mathscr{E}_i$ from the padded string. In practice, as observed in [4], it is easy to find the appropriate encoding and decoding functions.

THEOREM 7. *All of the following problems are polynomially isomorphic: graph isomorphism, regular graph isomorphism, semigroup isomorphism, commutative semigroup isomorphism, finite automaton isomorphism, and semiautomaton isomorphism.*

Proof. By use of Lemma 6 it is sufficient to prove the existence of the encoding and decoding functions; the previous theorems have already shown that all of the problems are polynomially equivalent. For simplicity we assume, without loss of generality, that all problems are represented over a $\{0, 1\}$ alphabet.

Graph isomorphism. The encoding function $\mathscr{E}_G$ for graph isomorphism is computed in polynomial time as follows. Given strings x and y determine if x is the description of two graphs, (G_1, G_2). If not, let $n = 0$. If yes, then let n be the maximum number of a vertex in either graph. Construct a description of two graphs (G_1', G_2') where

$$G_i' = G_i \cup H_0 \cup \bigcup_{j=1}^{|y|} H_j.$$

The graph H_0 is K_3 with its vertices numbered $n+1, n+2, n+3$, and each H_j is either K_1 (if $y_j = 0$) or K_2 (if $y_j = 1$) with distinct, successively higher numbered vertices.

The decoding function is easily computed by finding some copy of K_3 having the highest numbered vertices and then "reading off" y in its binary representation. The three properties of Lemma 6 are easily verified.

Regular graph isomorphism. The same functions work except that we first find r, the degree of regularity ($r \geqq 2$ can be assumed without loss of generality) and then let H_0 be the complement of an $(r+3)$-cycle and let H_j be K_{r+1} (if $y_j = 0$) or $K_{r,r}$ (if $y_j = 1$).

Semigroup isomorphism. The encoding function $\mathscr{E}_S$ is computed by finding n, the highest numbered element in the pair of semigroups (S_1, S_2). Elements a_{n+1}, $a_{n+2}, \cdots, a_{n+|y|+1}$ are then added to both semigroups such that

$$a_i \circ a_{n+1} = a_{n+1} \circ a_i = a_{n+1} \qquad \text{for } 1 \leqq i \leqq n+|y|+1,$$

$$a_i \circ a_{n+j+1} = a_{n+j+1} \circ a_i = \begin{cases} a_{n+j} & \text{if } y_j = 0 \\ a_{n+j+1} & \text{if } y_j = 1 \end{cases} \qquad \text{for } 1 \leqq i \leqq n+j+1 \text{ and } i \neq n+1.$$

The decoding function locates a_{n+1} (the unique annihilator or reset element) and then "reads off" the binary representation of y.

Commutative semigroup isomorphism. The semigroup encoding and decoding functions work here also since $\mathscr{E}_S(G_1, G_2)$ is a pair of commutative semigroups iff G_1 and G_2 are commutative.

Finite automaton isomorphism. Assume the initial state is q_1. The encoding function computes the number of states, m, and the number of letters, n, and then adds one new letter a_{n+1} and $|y|+1$ new states. The transition function is augmented by the extra transitions for $1 \leqq i \leqq n$, $1 \leqq j \leqq |y|$, $2 \leqq k \leqq m$

$$\delta(q_1, a_{n+1}) = q_{m+1},$$

$$\delta(q_{m+j}, a_i) = q_{m+j+1},$$

$$\delta(q_{m+j}, a_{n+1}) = q_{m+j+y_j},$$

$$\delta(q_{m+|y|+1}, a_i) = q_{m+|y|+1},$$

$$\delta(q_k, a_{n+1}) = q_k,$$

Again the decoding is easily polynomial and the conditions of the lemma are satisfied.

Semiautomaton isomorphism. This is similar to the previous construction for automata, although the alphabet size will be three instead of two as was the case in Theorem 5. Q.E.D.

7. Concluding remarks. We have shown that three isomorphism problems are of the same complexity: graph isomorphism, semigroup isomorphism, and finite automaton isomorphism. We have conjectured that group isomorphism is not of the same complexity but that it is easier. If the conjecture is true $P \neq NP$. If it is false there is an $O(n^{c_1 \log n + c_2})$ algorithm for graph isomorphism. A proof of either result would be most interesting.

An area for further research is to find a problem which is not obviously an isomorphism problem but which is polynomially equivalent to those examined here.

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. S. BOOTH, *Problems polynomially equivalent to graph isomorphism*, Tech. Rep. CS-77-04, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario.

[3] A. GINZBURG, *Algebraic Theory of Automata*, Academic Press, New York, 1968.

[4] J. HARTMANIS AND L. BERMAN, *On isomorphisms and density of NP and other complete sets*, Proceedings of the Eighth Annual ACM Symposium on Theory of Computing (May 1976), Association for Computing Machinery, New York, 1976, pp. 30–40.

[5] J. E. HOPCROFT AND J. K. WONG, *A linear time algorithm for isomorphism of planar graphs*, Proceedings of the Sixth Annual ACM Symposium on Theory of Computing (May 1974), Association for Computing Machinery, New York, 1974, pp. 172–184.

[6] R. M. KARP, *On the computational complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.

[7] E. LEISS, private communication.

[8] A. N. MELIKOV, L. S. BERSHTEYN AND V. P. KARELIN, *Isomorphism of graphs and finite automata*, Engrg. Cybernetics, 1 (1968), pp. 124–129.

[9] G. L. MILLER AND L. MONK, private communication.

[10] G. L. MILLER, *Graph isomorphism, general remarks*, Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (May 1977), pp. 143–150; J. Comput. System Sci., (Dec. 1978), to appear.

[11] R. C. READ AND D. G. CORNEIL, *The graph isomorphism disease*, J. Graph Theory, 1 (1977), pp. 239–363.

[12] R. E. TARJAN, Private communication.

SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0003 $01.00/0

FINDING ALL SPANNING TREES OF DIRECTED AND UNDIRECTED GRAPHS*

HAROLD N. GABOW† AND EUGENE W. MYERS†

Abstract. An algorithm for finding all spanning trees (arborescences) of a directed graph is presented. It uses backtracking and a method for detecting bridges based on depth-first search. The time required is $O(V+E+EN)$ and the space is $O(V+E)$, where V, E, and N represent the number of vertices, edges, and spanning trees, respectively. If the graph is undirected, the time decreases to $O(V+E+VN)$, which is optimal to within a constant factor. The previously best-known algorithm for undirected graphs requires time $O(V+E+EN)$.

Key words. spanning tree, arborescence, bridge, depth-first search

1. Introduction. The problem of finding all spanning trees of directed and undirected graphs arises in the solution of electrical networks [7, pp. 252–364]. Algorithms of varying efficiency have been proposed [4], [5], [6], [8], [9], [10], [11], [13]. For undirected graphs, the best algorithm seems to be that of Minty, Read and Tarjan. It uses $O(V+E+EN)$ time and $O(V+E)$ space, where the graph has V vertices, E edges, and N spanning trees. We refine their approach and present an algorithm that uses $O(V+E+VN)$ time and $O(V+E)$ space. In terms of worst-case asymptotic bounds, this algorithm is optimal. The algorithm also applies to directed graphs. Here it uses $O(V+E+EN)$ time and $O(V+E)$ space. A previous algorithm [11] uses exponential time per tree (in the worst case).

We first review some terms for undirected graphs, and generalize them to directed graphs. In a connected undirected graph G, a *spanning tree* is a subgraph having a unique simple path between any two vertices of G. A *bridge* is an edge e where $G-e$ is not connected. Equivalently, e is in every spanning tree of G.

In a directed graph G, a *spanning tree (rooted at r)* is a subgraph having a unique (directed) path from r to any vertex of G. If such a tree exists, G is *rooted at r*. A *bridge (for r)* is an edge e where $G-e$ is not rooted at r. Equivalently, e is in every spanning tree rooted at r. ("Spanning arborescence" is often used for "spanning tree" of a directed graph. There appears to be no standard term for what we call a "bridge" of a directed graph.)

The problem we consider is to find all spanning trees of a graph. This means that for a given graph, a list is to be printed that contains each spanning tree exactly once. Section 2 presents our results. Section 3 discusses some open problems.

2. Algorithm for all spanning trees. This section begins with an algorithm for all spanning trees rooted at a given vertex r in a directed graph. This algorithm is used to find all spanning trees, first in a directed graph and then in an undirected graph.

For all spanning trees rooted at r, the approach is to find all spanning trees containing a given subtree T rooted at r. To do this, first choose an edge e_1 directed from T to a vertex not in T; find all spanning trees containing $T \cup e_1$; then delete e_1 from the graph. Next choose an edge e_2 from T to a vertex not in T; find all spanning trees (in the modified graph) containing $T \cup e_2$; then delete e_2. To continue, repeatedly choose an edge e_i from T to a vertex not in T; find all spanning trees (in the modified graph) containing $T \cup e_i$; then delete e_i. Stop when the edge e_k that has just been processed is a bridge of the modified graph. At this point each spanning tree

* Received by the editors January 21, 1977. This work was partially supported by the National Science Foundation under Grant GJ36461.

† Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309.

containing T has been found (exactly once). For if a spanning tree does not contain any e_j, $j<k$, it must contain the bridge e_k.

This basic approach needs an efficient method for discovering when an edge e is a bridge. This can be done in a variety of ways; set merging techniques and edge exchanges are two possibilities [3]. Below we describe a method based on depth-first search.

We choose edges e so the tree T grows depth-first. More precisely, we always add the edge e to T that originates at the greatest depth possible. Now suppose all spanning trees containing $T \cup e$ have been found, and we want to check if e is a bridge. Let L be the last spanning tree found that contains $T \cup e$, and let $e=(u, v)$. Intuitively, in L, vertex v has the fewest descendents possible (among all spanning trees containing $T \cup e$). Equivalently, no edge goes from a nondescendent of v to a proper descendent of v. (This is proved below in Lemma 3.) So e is a bridge when no edge (besides e) goes from a nondescendent of v to v. This observation gives an efficient bridge test.

To grow T depth-first requires some care. The algorithm below uses F, a list of all edges directed from vertices in T to vertices not in T. F uses stack operations: to enlarge T, an edge e is popped from the front of F and added to T; new edges for $T \cup e$ are pushed onto the front of F. In addition when e is added to T, some edges are removed from F; when e is removed from T, these edges are restored in F. It is crucial that the remove and restore operations leave the order of edges unchanged in F. Otherwise, T will not grow depth-first.

Besides F, the algorithm uses lists FF. Each recursive invocation has a local FF list. It is used to reconstruct the original F list. It is managed as a stack.

The algorithm also uses data structures for T, the current tree, and L, the last spanning tree output thus far. The algorithm is given below in ALGOL-like notation.

```
                    procedure S; comment S finds all spanning trees rooted
                      at r in a directed graph G rooted at r; begin
                      procedure GROW; comment GROW finds all spanning
                        trees rooted at r containing T; begin
 1.                     if T has V vertices then begin L ← T; output (L) end
 2.                     else begin make FF an empty list, local to GROW;
 3.                       repeat
 4. new tree edge:          pop an edge e from F; let e go from T to vertex v,
                              v ∉ T;
 5.                         add e to T;
 6. update F:               push each edge (v, w), w ∉ T, onto F;
 7.                         remove each edge (w, v), w ∈ T, from F;
 8. recurse:                GROW;
 9. restore F:              pop each edge (v, w), w ∉ T, from F;
10.                         restore each edge (w, v), w ∈ T, in F;
11. delete e:               remove e from T and from G; add e to FF;
12. bridge test:            if there is an edge (w, v), where w is not a
                              descendent of v in L then b ← false else b ← true;
13.
14. reconstruct G:        until b;
                          pop each edge e from FF, push e onto F, and add e to G;
                        end end GROW;
```

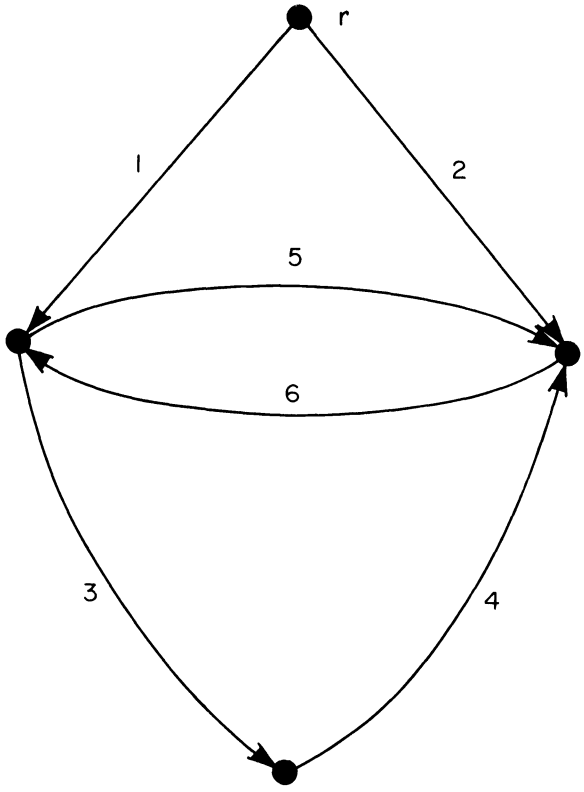


FIG. 1. *Example graph.*

15. *start*: initialize T to contain the vertex r; initialize F to contain all edges (r, v) from r;
16. GROW;
end S;

Figure 1 shows a graph with four spanning trees rooted at r. Figure 2 shows a computation tree indicating how S finds these trees T_i, $1 \leqq i \leqq 4$. In the computation tree, a node represents a call to GROW; the arcs directed down from the node correspond to the edges e added to T in line 5. For example, the root node first adds edge 1, then deletes 1 and adds 2. Since 2 is a bridge in the modified graph, no other edges are added.

Note the importance of restoring edges in correct order in line 10. When edge 4 is added to get T_1, edges 5 and 2 are removed from F. If they are restored in opposite order (i.e., 2, 5), T_3 is found, and then T_2; then edge 1 is mistakenly declared a bridge, and T_4 is not found.

Now we prove procedure S is correct. Note the original graph G is modified by GROW, by deleting and replacing edges (lines 11,14). In the discussion below the *current graph* refers to the edges in the graph at a specified point in the computation.

We first show the tree T grows depth-first. This amounts to showing F simulates the stack of active vertices in a normal depth-first search.

LEMMA 1. *Let* GROW *be called with* F *containing the sequence of edges* (v_i, w_i), $i = 1, 2, \cdots, |F|$. *Then*

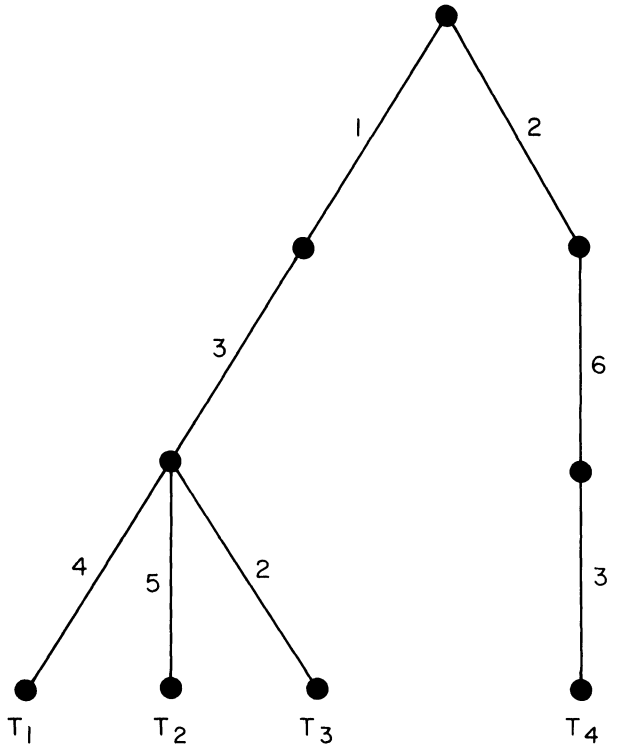


FIG. 2. *Computation tree.*

(i) *F contains all edges joining T to the rest of the graph, i.e.,* $\{(v, w)|v \in T, w \notin T, (v, w)$ *is in the current graph*$\} = \{(v_i, w_i)|1 \leqq i \leqq |F|\}$.

(ii) *F contains edges in a "depth-first order", i.e., if $j \geqq i$ then v_j is a descendent*[1] *of v_i in T.*

Proof. First note that on exit from GROW, F is identical to what it was on entry. This follows by observing that the changes to F in lines 4, 6 and 7 are undone by lines 14, 9 and 10, respectively.

Clauses (i)–(ii) of the Lemma hold for the first call to GROW (line 16), by the initialization step (line 15). In general, if clauses (i)–(ii) hold when GROW is called, they hold for the calls made in line 8, by inspection of lines 3–13 and by the preliminary remark. So by induction, clauses (i)–(ii) hold for all calls to GROW. □

COROLLARY 1. *Let e_j, $1 \leqq j \leqq |T|$, be the edges in T, indexed in the order they are added to T. Let e_j be directed to vertex v_j. Then the descendents of any v_j in T are vertices v_k, $j \leqq k \leqq J$, for some J.*

Proof. It suffices to show the descendents of a given vertex v are added to T consecutively. We do this as follows. Imagine modifying G, by adding an edge $(v, 0)$ leading to a dummy vertex 0; in GROW, when v is added to T and edges (v, w) are pushed onto F (line 6), push $(v, 0)$ first. Now Lemma 1(ii) shows as long as $(v, 0)$ is in F, the edges added to T (in line 4) join descendents of v. When $(v, 0)$ is removed from F, Lemma 1(i) shows there are no edges from descendents of v to vertices not in T. Thus no other vertices become descendents of v. The corollary follows. □

Now we show the bridge test of line 12 is correct.

[1] A vertex is considered a descendent, but not a proper descendent, of itself.

LEMMA 2. *The bridge test sets b to* **true** *exactly when edge e is a bridge of the current graph.*

Proof. Let $e=(u, v)$, and let D denote the descendents of vertex v in the latest spanning tree L. Below we show that when e's bridge test is executed, the current graph has no edge (w, x), where $w \notin D$, $x \in D-v$. This suffices to prove the lemma. For e is *not* a bridge if and only if some path P not containing e goes from r to v. When there are no edges (w, x) as above. P must end in an edge (w, v), $w \notin D$; further, P exists if and only if such an edge (w, v) exists. The bridge test checks if (w, v) exists. Hence it is correct.

So we must show the current graph has no edge (w, x), $w \notin D$, $x \in D-v$. Let the edges in L be e_j, $j=1, \cdots, V-1$, indexed in the order they are added; let $e=e_i$. The bridge test is executed on edges e_j, $j=V-1, \cdots, i+1$, and then on $e_i=e$. For $j>i$, b is set **true**. (Otherwise, another spanning tree would be output after L.)

Now consider any vertex $x \in D-v$. By Corollary 1, the edge in L directed to x is some e_h, $h>i$. So b is **true** for e_h. Thus no edge (w, x), $w \notin D$, exists when e_h's bridge test is executed.

So if (w, x) is in the current graph, it is added in an execution of line 14 following some e_k's bridge test, where $i<k \leqq h$. Corollary 1 shows e_k joins descendents of v. The edges added following e_k's bridge test precede e_k in list F. Lemma 1 (ii) shows these edges originate from D. Thus no edge (w, x), $w \notin D$, is added. We conclude no edge (w, x) is in the current graph. □

Now we can show S is correct.

LEMMA 3. *Procedure S finds all spanning trees rooted at r of a directed graph G rooted at r.*

Proof. Suppose GROW is called, with T a tree rooted at r. Let C be the current graph (when GROW is called). It suffices to show GROW finds all spanning trees (rooted at r) of C containing T. For in the initial call (line 16), T contains only the vertex r, and $C=G$.

The proof is by induction, with the calls to GROW ordered so the size of T is nonincreasing. For the base case, T contains V vertices; this is handled correctly by line 1. For the inductive step, suppose when GROW is called T contains less than V vertices. Let F contain edges e_i, $i=1, \cdots, |F|$. Define

$$\mathcal{T}_i = \{R \mid R \text{ is a spanning tree rooted at } r \text{ and } T \cup e_i \subseteq R \subseteq C-\{e_j \mid 1 \leqq j<i\}\}.$$

By induction, it is easy to see GROW finds the trees $\cup_{i=1}^{k} \mathcal{T}_i$, where e_k is the first edge for which b (in line 12) is **true**. Sets $\mathcal{T}_i$ are disjoint, by definition. Lemma 2 shows e_k is a bridge in the graph $C-\{e_j \mid 1 \leqq j<k\}$. Thus any spanning tree R of C that contains T contains some e_j, $1 \leqq j \leqq k$, i.e., $R \in \mathcal{T}_j$. So GROW finds the desired spanning trees. □

To estimate the efficiency of S, we must give some implementation details. First we discuss how F is managed, and in particular, how it is restored to its original state in line 10. F is a doubly linked list of edges. Line 7 traverses the list of edges directed to v, from beginning to end. Each edge directed from T is removed from F; however, the values of its links are *not* destroyed. Line 10 traverses the list of edges directed to v in the reverse direction, from end to beginning. Each edge directed from T is inserted in F, at the position given by its link values. This way, each edge is restored in its original position.

Next we discuss the implementation of the bridge test. To detect descendents efficiently, the vertices of L are numbered in preorder [1, pp. 54–55]: For a vertex v, $P(v)$ is v's preorder number, and $H(v)$ is the highest preorder number of a descendent of v. So w is a descendent of v if and only if $P(v) \leqq P(w) \leqq H(v)$. This test is used in

line 12. In line 1, when L is formed, the values $P(v)$ and $H(v)$ are computed and stored in the data structure for L.

Now we derive the resource bounds for S.

LEMMA 4. *Procedure S uses $O(EN)$ time and $O(E)$ space on a directed graph rooted at r.*

Proof. First consider time. One execution of the body of the **repeat** loop (lines 4–12), excluding the recursive call (line 8), takes time proportional to the number of edges directed to and from vertex v. Here v is the vertex added to T. In the process of generating a spanning tree, v ranges over all vertices (except r). So the total time in the loop body for one tree is $O(E)$. This dominates the run time of S, which thus is $O(EN)$.

Next consider the space. The graph $\dot{G}$ is stored as a collection of doubly linked lists of edges directed to and from each vertex. This uses $O(E)$ space. At any point in the computation, an edge e may be on the F list, or on at most one FF list. So F and FF use $O(E)$ space. In addition, $O(V)$ space is needed for T, P, and H. Thus the space is $O(E)$. □

Now consider the problem of finding all spanning trees of a directed graph. The possible root vertices r form a strongly connected component that precedes all others. A strong connectivity algorithm can be used to find these roots in time $O(V+E)$[12]. Then procedure S can be applied to each root. So we have the following result.

THEOREM 1. *All spanning trees of a directed graph can be found in time $O(V+E+EN)$ and space $O(V+E)$.*

Next consider the problem of finding all spanning trees of an undirected graph. If the graph is made directed (by giving each edge both directions), and root r is chosen arbitrarily, then procedure S finds all spanning trees of the undirected graph. The time can be estimated more precisely, as follows:

THEOREM 2. *All spanning trees of an undirected graph can be found in time $O(V+E+VN)$ and space $O(V+E)$.*

Proof. We need only show the time bound. Line 1 does a preorder traversal and outputs each spanning tree; the time is clearly $O(V)$ per tree, or $O(VN)$ total. Now we analyze the time spent in lines 4–12, when edge e is added. Ignoring the recursive call (line 8), the time is proportional to the number of edges incident to v. Now we consider two cases, and show in each case the total time in lines 4–12 is $O(VN)$.

First suppose e is a bridge. Each edge f incident to v is in some spanning tree R containing $T \cup e$. Charge the time spent on f, $O(1)$, to R. Then each spanning tree gets charged $O(V)$. So a total time $O(VN)$ is spent on bridges in lines 4–12.

Now suppose e is a nonbridge. The time spent on e in lines 4–12 is $O(V)$. Now we show there are exactly $N-1$ nonbridges, so the total time spent on nonbridges is $O(VN)$: Let the nonbridge e correspond to the tree L used in e's bridge test. Since e fails the test, it gets deleted, and another tree is grown before the next bridge test. So a given tree L corresponds to at most one nonbridge. If L is any spanning tree but the last one found, it is used in the bridge test for some nonbridge. So it corresponds to precisely one nonbridge. Thus there are exactly $N-1$ nonbridges. □

Procedure S can be sped up in a number of ways. The preorder labeling of trees can be done as trees are grown. Several trees can be grown at once (e.g., each edge (w, v) in line 7 gives a spanning tree. Algorithms using this "factoring" approach are [2, pp. 20–25], [8]). However, if each tree is output as a list of edges, $O(VN)$ time is required for the output step. So on undirected graphs, the algorithm is optimal, to within a constant factor.

We have programmed S and other spanning tree algorithms in FORTRAN on

TABLE 1.
Time for graphs with $V = 10$.

N	50	105	310	680	839	1415
t(msec)	100	180	456	831	1048	1634

the CDC6400. Compared to the Minty, Read and Tarjan algorithm, S is over 3 times faster for $6 \leqq V \leqq 10$, $8 \leqq E \leqq 14$; the difference increases with denser graphs. Table 1 shows that the time for S, with V fixed at 10, is approximately proportional to N, as predicted by the $O(VN)$ time bound.

3. Open problems. This section briefly discusses two problems related to this work. The first is to improve the $O(V+E+EN)$ time bound for spanning trees of a directed graph. To illustrate the difficulty here, consider the family of graphs illustrated in Fig. 3. The top part consists of N paths of length 2 from r to s; the bottom part is a directed path of length N from s to t, plus all possible back edges. The

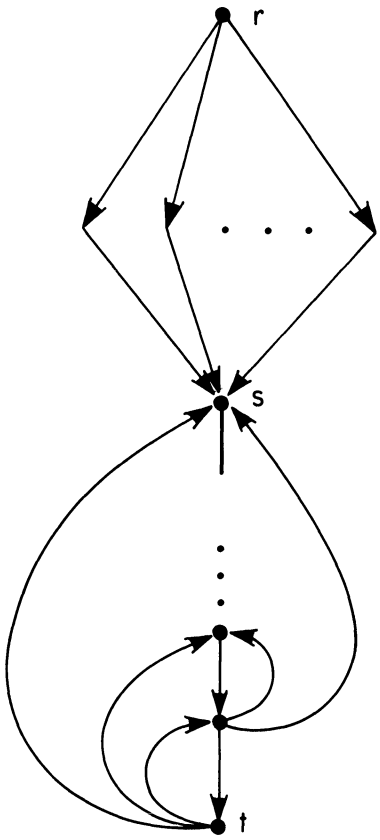


FIG. 3. *Difficult graph.*

algorithm of Theorem 1 uses $O(EN)$ time on these graphs. The time spent repeatedly scanning back edges is "wasted." (As a point of interest, note the algorithm of [11] can use exponential time per tree on these graphs.)

The second problem is, can the computation tree of S (Fig. 2) be represented in less than $O(VN)$ space? Note some computation trees have $O(VN)$ nodes. For

instance, the tree for an undirected cycle has $V(V-1)/2 = O(VN)$ nodes. There are two reasons why a more compact form is desirable.

First, the claim S is optimal for undirected graphs is based on a lower bound for outputting the spanning trees. If a computation tree is acceptable output, it may be possible to lower this bound and speed up the algorithm.

Second, consider the problem of listing all spanning trees in order of increasing weight in a weighted undirected graph. (In a *weighted* graph, each edge has a numerical weight; a tree's weight is the sum of all its edge weights.) One approach is to find all spanning trees, and then sort them. The sort takes time $O(N \log N)$, which is $O(\min(V \log V, E)N)$, since $N \leqq \min(2^E, V^{V-2})$. This dominates the run time of the algorithm. The space is $O(VN)$, since the spanning trees must be saved until the sort is done. A previous algorithm [3] uses $O(EN)$ time and $O(E+N)$ space. So our approach is no slower, sometimes faster, but uses more space. Thus a "reduced" computation tree is desirable.

Acknowledgments. The authors thank the referees for their suggestions.

REFERENCES

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

[2] S. M. CHASE, *Analysis of algorithms for finding all spanning trees of a graph*, RC3190, IBM T. J. Watson Research Center, Yorktown Heights, NY, Dec. 1970.

[3] H. N. GABOW, *Two algorithms for generating weighted spanning trees in order*, this Journal, 6 (1977), pp. 139–150.

[4] S. L. HAKIMI AND D. G. GREEN, *Generation and realization of trees and k-trees*, IEEE Trans. on Circuit Theory, CT-11 (1964), pp. 247–255.

[5] F. J. MACWILLIAMS, *Topological network analysis as a computer program*, IRE Trans., CT-5 (1958), pp. 228–229.

[6] W. MAYEDA AND S. SEHU, *Generation of trees without duplications*, IEEE Trans. on Circuit Theory, CT-12 (1965), pp. 181–185.

[7] W. MAYEDA, *Graph Theory*, John Wiley, New York, 1972.

[8] M. D. MCILROY, *Generation of spanning trees (Algorithm 354)*, Comm. ACM, 12 (1969), p. 511.

[9] G. J. MINTY, *A simply algorithm for listing all the trees of a graph*, IEEE Trans. on Circuit Theory, CT-12 (1965), p. 120.

[10] R. C. READ AND R. E. TARJAN, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, Networks, 5 (1975), pp. 237–252.

[11] S. SHINODA, *Finding all possible directed trees of a directed graph*, Electron. Commun. Japan, 51-A (1968), pp. 45–46.

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

[13] H. WATANABE, *A computational method for network topology*, IRE Trans., CT-7 (1960), pp. 296–392.

SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0004 $01.00/0

EFFICIENT CALCULATION OF EXPECTED MISS RATIOS IN THE INDEPENDENT REFERENCE MODEL*

RONALD FAGIN† AND THOMAS G. PRICE†

Abstract. In the independent reference model of program behavior, King's formulas for the expected FIFO ("first-in-first-out") and expected LRU ("least-recently-used") miss ratios each contain an exponential number of terms (very roughly n^{CAP}, where n is the number of pages and CAP is the capacity of main memory). Hence, under the straightforward algorithms, these formulas are computationally intractable. We present an algorithm which is both efficient (there are $O(n \cdot \text{CAP})$ additions, multiplications, and divisions) and provably numerically stable, for calculating the expected FIFO miss ratio. In the case of LRU, we present an efficient method, based on an urn model, for obtaining an unbiased estimate of the expected LRU miss ratio (the method requires $O(n \cdot \text{CAP})$ additions and comparisons, and $O(\text{CAP})$ divisions and random number generations).

Key words. independent reference model, miss ratio, FIFO, first-in-first-out, LRU, least-recently-used, storage management, page replacement, numerically stable

1. Introduction. The *independent reference model* (IRM) is a simple, widely studied model of page reference behavior in a paged computer system (Aho, Denning and Ullman [1]; Aven, Boguslavskii and Kogan [2]; Fagin [7], [8]; Fagin and Easton [9]; Franaszek and Wagner [10]; Gelenbe [11]; King [13]; Yue and Wong [24]). In this model, at each point in discrete time, exactly one page is referenced, where page i is referenced with probability p_i, independent of past history. We present an efficient, numerically stable algorithm for obtaining the expected FIFO ("first-in-first-out") miss ratio, and an efficient algorithm, based on an urn model, for obtaining an unbiased estimate of the expected LRU ("least-recently-used") miss ratio.

It is known that actual program page reference and data base segment or page reference patterns in a paging environment are quite intricate (Lewis and Shedler [15]; Lewis and Yue [16]; Madison and Batson [17]; Spirn and Denning [23]; Rodriguez-Rosell [21]). In particular, sequences of page references may be nonstationary. The assumption of independent references is not only intuitively suspect, but inconsistent with observed reference patterns. Why, therefore, should the IRM be investigated?

In the study of computer system performance, it is sometimes helpful to experiment with overly simple models, in order to gain insight into system behavior. In particular, since paging is a complex phenomenon, it is useful to study the effects of paging in conjunction with simple models of page reference patterns. From a mathematical point of view, the IRM is the simplest model in which pages retain their identity (as opposed, for example, to the independent LRU stack model of Oden and Shedler [19], and related models, in which all pages are treated identically). We remark that the formulation of other simple models that capture salient aspects of the referencing behavior of programs remains an important problem. The IRM is simple enough so as to be tractable, yet complex enough in the context of paging that there are nontrivial, surprising results. Sometimes these results generalize to realistic situations. Thus, in Fagin and Easton [9], it is shown that from the approximate independence of miss ratio on page size in the IRM, it follows that the miss ratio is approximately independent of page size in certain more realistic models, in which there is a

* Received by the editors November 1, 1976, and in final revised form October 25, 1977.
† IBM Research Laboratory, San Jose, California 95193.

"random" component and a "sequential" component. In fact, this independence of miss ratio on page size has been observed in an actual database system (see [9] for details).

Beyond easing the task of experimenting with the IRM, there is a further justification for obtaining algorithms that efficiently calculate miss ratios for the IRM. The IRM has been used as a component of more complex models that have accurately predicted miss ratio behavior. Thus, the miss ratios for Easton's model of data base references [6] and for Baskett and Rafii's model of program references [3] can be obtained directly from miss ratios in an IRM. The Easton and the Baskett–Rafii miss ratios, in turn, are supposedly fairly accurate predictors of genuine systems miss ratios.

Throughout, we assume that there are n pages, and that the probability that page i is referenced at time t is p_i $(i = 1, \cdots, n)$, independent of past history. Of course, $\sum p_i = 1$. Denote the *capacity*, or size of first-level memory, by CAP $(1 \leqq \text{CAP} \leqq n)$.

We deal with two page replacement algorithms, both of which are *demand* policies (Aho et al. [1]); that is, a page is brought into main (first-level) memory if and only if it is referenced but not present in main memory. The choice of which page is removed from main memory to make room for the newly-referenced but nonpresent page is determined by the page replacement algorithm. The first page replacement algorithm which we study in this paper is FIFO (Belady [4]), which replaces the page that has spent the longest time in memory. The second page replacement algorithm which we study is LRU (Mattson, Gecsei, Slutz, and Traiger [18]), which replaces the page that has been least recently referenced.

Define the *expected miss ratio* (in the independent reference model) to be the limit (as $t \to \infty$) of the probability that the page referenced at time t was not present in main memory at time t. King [13] showed that the expected FIFO and expected LRU miss ratios exist and are independent of the initial configuration of main memory. He showed that the expected FIFO miss ratio is

$$\frac{\sum p_{i_1} p_{i_2} \cdots p_{i_{\text{CAP}}} (1 - p_{i_1} - \cdots - p_{i_{\text{CAP}}})}{\sum p_{i_1} \cdots p_{i_{\text{CAP}}}}, \tag{1.1}$$

where the sums are each taken over all CAP-tuples $(i_1, \cdots, i_{\text{CAP}})$ such that $i_j \neq i_k$ if $j \neq k$. Further, he showed that the expected LRU miss ratio is

$$\sum \frac{p_{i_1} p_{i_2} \cdots p_{i_{\text{CAP}}} (1 - p_{i_1} - \cdots - p_{i_{\text{CAP}}})}{(1 - p_{i_1})(1 - p_{i_1} - p_{i_2}) \cdots (1 - p_{i_1} - p_{i_2} - \cdots - p_{i_{\text{CAP}-1}})}, \tag{1.2}$$

where again, the sum is taken over all CAP-tuples $(i_1, \cdots, i_{\text{CAP}})$ such that $i_j \neq i_k$ if $j \neq k$. We note that Gelenbe [11] showed that under the RAND ("random") page replacement algorithm (Belady [4]), in which the page to be removed from main memory in the event of a page fault is selected randomly, the expected miss ratio is the same as that of FIFO, that is, formula (1.1).

Each of the sums appearing in (1.1) and (1.2) contain very roughly n^{CAP} terms (actually $n(n-1)\cdots(n-\text{CAP}+1)$ terms). Hence, for moderate values of n and CAP, formulas (1.1) and (1.2) cannot be evaluated numerically under the straightforward algorithm. For example, if $n = 100$ and $\text{CAP} = 30$, then each of the sums contain over 10^{57} terms. The purpose of this paper is to provide fast, stable methods for evaluating the expected FIFO miss ratio (1.1) and for approximating the expected LRU miss ratio (1.2).

2. An efficient algorithm for the expected FIFO miss ratio. In this section we present an efficient, provably stable algorithm for evaluating King's formula

$$\frac{\sum p_{i_1} p_{i_2} \cdots p_{i_{\mathrm{CAP}}}(1 - p_{i_1} - \cdots - p_{i_{\mathrm{CAP}}})}{\sum p_{i_1} p_{i_2} \cdots p_{i_{\mathrm{CAP}}}} \tag{2.1}$$

for the expected FIFO miss ratio in the independent reference model. (The sums are taken over all CAP-tuples $(i_1, \cdots, i_{\mathrm{CAP}})$ such that $i_j \neq i_k$ if $j \neq k$.)

Let $p_1, \cdots, p_n$ be a fixed but arbitrary ordering of the n page probabilities. For each positive integer $m \leqq n$ and each positive integer r, define

$$E(r, m) = \sum p_{i_1} \cdots p_{i_r},$$

where the sum is taken over all r-element *subsets* $\{i_1, \cdots, i_r\}$ of $\{1, \cdots, m\}$.

In other words, for each r-element subset of the first m probabilities $p_1, \cdots, p_m$, there is a term of $E(r, m)$ which is the product of these r probabilities. We make the usual convention that an empty sum is 0; hence $E(r, m) = 0$ if $r > m$.

We now express the expected FIFO miss ratio (2.1) as a function of the terms $E(r, m)$. Note that the numerator of formula (2.1) can be rewritten as

$$\sum p_{i_1} p_{i_2} \cdots p_{i_{\mathrm{CAP}}} \sum_{i \neq i_1, \cdots, i_{\mathrm{CAP}}} p_i,$$

which equals

$$\sum p_{i_1} p_{i_2} \cdots p_{i_{\mathrm{CAP}+1}}, \tag{2.2}$$

where the sum in (2.2) is taken over all (CAP+1)-tuples $(i_1, \cdots, i_{\mathrm{CAP}+1})$ such that $i_j \neq i_k$ if $j \neq k$. But (2.2) is simply $(\mathrm{CAP}+1)!E(\mathrm{CAP}+1, n)$, since $E(\mathrm{CAP}+1, n)$ is a sum over sets while (2.2) is the corresponding sum over tuples. Likewise, the denominator of (2.1) equals $\mathrm{CAP}!E(\mathrm{CAP}, n)$. Since we just showed that the numerator of (2.1) equals $(\mathrm{CAP}+1)!E(\mathrm{CAP}+1, n)$ and the denominator is $\mathrm{CAP}!E(\mathrm{CAP}, n)$, it follows that the expected FIFO miss ratio (2.1) equals

$$\frac{(\mathrm{CAP}+1)E(\mathrm{CAP}+1, n)}{E(\mathrm{CAP}, n)}. \tag{2.3}$$

We now show how to obtain an efficient, numerically stable algorithm for computing (2.3). Along the way, we also derive an efficient but *unstable* algorithm for computing (2.3).

We first verify the following recurrence equation for $E(r, m)$ when $r > 1$ and $m > 1$:

$$E(r, m) = E(r, m-1) + p_m E(r-1, m-1). \tag{2.4}$$

The first term $E(r, m-1)$ of (2.4) is the sum of those terms in $E(r, m)$ which do not have p_m as a factor, and the second term $p_m E(r-1, m-1)$ is the sum of the terms which do have p_m as a factor.

Recurrence equation (2.4) can be used recursively to compute the matrix of values $E(r, m)$ with $1 \leqq r \leqq \mathrm{CAP}+1$ and $1 \leqq m \leqq n$. Using this approach, we can calculate (2.3), the expected FIFO miss ratio, with approximately $2n \cdot \mathrm{CAP}$ additions and multiplications. Unfortunately, this method suffers from numerical instability, because for interesting values of n and CAP, the entries of the E matrix vary by enough orders of magnitude that they exceed the range of typical floating-point hardware (and so underflow occurs). For example, assume that all page reference

probabilities are equal. Then $E(r, m) = \binom{m}{r}(1/n)^r$. So if $n = 1000$, then $E(1,1000) = 1$ and $E(1000, 1000) = 10^{-3000}$. The reason why underflow causes large errors for us is that if we add two terms x and y which are nearly equal (such as $x = E(r, m-1)$ and $y = p_m E(r-1, m-1)$ on the right-hand side of (2.4)), and if the y term has underflowed to zero but the x term has not, then a large relative error is introduced and then propagated. In the example given, the actual value of $x+y$ would be almost twice the calculated value.

We now show how to calculate (2.3) both efficiently and stably. In order to calculate (2.3), we need only calculate the ratio $E(\text{CAP}+1, n)/E(\text{CAP}, n)$ (but not $E(\text{CAP}+1, n)$ and $E(\text{CAP}, n)$ separately). Let

$$F(r, m) = E(r, m)/E(r-1, m)$$

for $1 \leqq r \leqq \text{CAP}+1$ and $1 \leqq m \leqq n$. Under the usual convention that empty products are 1, we define $E(0, m)$ to be 1 for each m; then $F(1, m) = E(1, m) = p_1 + \cdots + p_m$. Formula (2.3), and hence the expected FIFO miss ratio, equals $(\text{CAP}+1)F(\text{CAP}+1, n)$. We can derive a recurrence equation for F directly and avoid our earlier numerical difficulties. We first note that if $r > 1$ and $m > 1$, then

$$\begin{aligned} F(r, m) &= \frac{E(r, m)}{E(r-1, m)} \\ &= \frac{E(r, m)/E(r-1, m-1)}{E(r-1, m)/E(r-2, m-1)} E(r-1, m-1)/E(r-2, m-1). \end{aligned} \tag{2.5}$$

If we divide both sides of (2.4) by $E(r-1, m-1)$, then we obtain

$$\begin{aligned} E(r, m)/E(r-1, m-1) &= (E(r, m-1)/E(r-1, m-1)) + p_m \\ &= F(r, m-1) + p_m. \end{aligned} \tag{2.6}$$

If we use (2.6) to replace $E(r, m)/E(r-1, m-1)$ in (2.5) by $F(r, m-1)+p_m$, and if similarly, we replace $E(r-1, m)/E(r-2, m-1)$ in (2.5) by $F(r-1, m-1)+p_m$, then we obtain the recurrence equation

$$F(r, m) = \frac{F(r, m-1)+p_m}{F(r-1, m-1)+p_m} F(r-1, m-1), \tag{2.7}$$

which holds for $1 \leqq r \leqq \text{CAP}+1$ and $1 \leqq m \leqq n$. This recurrence equation can be used recursively to compute the matrix of values $F(r, m)$ starting from the boundary conditions

$$\begin{aligned} F(r, 1) &= \begin{cases} p_1, & r = 1, \\ 0, & 2 \leqq r \leqq \text{CAP}+1, \end{cases} \\ F(1, m) &= p_1 + \cdots + p_m, \quad 1 \leqq m \leqq n. \end{aligned} \tag{2.8}$$

One way to calculate F is to initialize the first column and row using the equations for $F(r, 1)$ and $F(1, m)$ in (2.8) and then to calculate the entries for $2 \leqq r \leqq \text{CAP}+1$ and $2 \leqq m \leqq n$, column by column, by using equation (2.7). Then the expected FIFO miss ratio with capacity CAP is $(\text{CAP}+1)F(\text{CAP}+1, n)$.

This algorithm requires approximately $4n \cdot \mathrm{CAP}$ additions, multiplications, and divisions. We note that this algorithm has the interesting property that in calculating the expected FIFO miss ratio with capacity CAP, we automatically calculate the expected FIFO miss ratios with capacities $1, \cdots, \mathrm{CAP}-1$.

We sketch a proof of the numerical stability of this FIFO algorithm in the Appendix.

3. An unbiased estimate of the expected LRU miss ratio. In this section we present an efficient method for obtaining an unbiased estimate of the expected LRU miss ratio, which, we recall, is given by

$$\sum \frac{p_{i_1} p_{i_2} \cdots p_{i_{\mathrm{CAP}}} (1 - p_{i_1} - \cdots - p_{i_{\mathrm{CAP}}})}{(1-p_{i_1})(1-p_{i_1}-p_{i_2}) \cdots (1-p_{i_1}-p_{i_2}-\cdots-p_{i_{\mathrm{CAP}-1}})}, \tag{3.1}$$

in the independent reference model.

Consider the following experiment, which involves drawing balls from an urn without replacement. Assume that an urn contains n balls numbered $1, \cdots, n$ (which correspond to our n pages). We say that ball i has *weight* p_i $(i = 1, \cdots, n)$, where $\{p_1, \cdots, p_n\}$ is the page probability distribution. Select one ball from the urn, in such a way that a given ball is selected with probability equal to its weight. Thus, ball i is selected with probability p_i $(i = 1, \cdots, n)$. Assume that ball i_1 was selected. Now renormalize the weights of the remaining $(n-1)$ balls so that the sum of their weights is 1. Thus, the weight of ball j is now $p_j/(1-p_{i_1})$, for $j \neq i_1$. Select a second ball from the urn, where once again a given ball is selected with probability equal to its new weight. Assume that ball i_2 was selected. Now renormalize the weights of the remaining $(n-2)$ balls so that the sum of their weights is 1: thus, the weight of ball j is now $p_j/(1-p_{i_1}-p_{i_2})$, for $j \neq i_1, i_2$. Continue the process until CAP balls have been selected. Let A (an estimate of (3.1)) be the value $1 - p_{i_1} - \cdots - p_{i_{\mathrm{CAP}}}$. Note that with probability

$$p_{i_1} \frac{p_{i_2}}{1-p_{i_1}} \frac{p_{i_3}}{1-p_{i_1}-p_{i_2}} \cdots \frac{p_{i_{\mathrm{CAP}}}}{1-p_{i_1}-\cdots-p_{i_{\mathrm{CAP}-1}}}, \tag{3.2}$$

ball i_1 was selected first, ball i_2 was selected second, $\cdots$, and ball i_{CAP} was selected last; in this case A took on the value $1 - p_{i_1} - \cdots - p_{i_{\mathrm{CAP}}}$. Therefore, the expected value of the random variable A is given by (3.1); that is, A is an unbiased estimate of (3.1).

The experiment we just described is faithfully mimicked by the algorithm in Figure 1 for obtaining a value for A (in the first line, P is our probability vector of page probabilities).

```
LET P1 = P;
LET A = 1;
DO I = 1 TO CAP;
    SELECT A RANDOM NUMBER R BETWEEN 0 AND 1;
    FIND THE FIRST J BETWEEN 1 AND N SUCH THAT
      P1(1)+···+P1(J)≧R;
    LET A = A-P(J);
    LET S = 1-P1(J);
    DO K = 1 TO N;
     LET P1(K) = P1(K)/S;
    END;
    LET P1(J) = 0;
END;
```

FIG. 1

If there are, say, 100 independent replications of the experiment (that is, if the program is run 100 times, with different seeds to the pseudo-random number generator), then the average of the 100 values of A which are obtained also, of course, give an unbiased estimate of (3.1), and we can use the central limit theorem to obtain approximate confidence intervals for our estimate.

We now give a numerical example, using "Zipf's Law" (Zipf [25]; Knuth [14, vol. 3, p. 397]), in which the probability p_i of referencing the ith most frequently referenced page is

$$p_i = \frac{k}{i^{\theta}}, \qquad 1 \leqq i \leqq n,$$

where θ is a positive constant (the "skewness"), and k is a normalizing constant chosen so that $\sum p_i = 1$.

In our example, the skewness θ is 0.5, the number n of pages is 100, and the capacity CAP is 30. When we ran a version of the program in Fig. 1 100 times, we obtained 100 results $A_1, \cdots, A_{100}$. The average value

$$\bar{A} = (A_1 + \cdots + A_{100})/100$$

turned out to be 0.6119 (rounded to 4 decimal places). This value is our unbiased estimate of the expected LRU miss ratio (3.1). How much confidence should be placed in this estimate? To answer this question, we calculated several other statistical quantities of interest. Let

$$D = \left(\left(\sum_{i=1}^{100} (A_i - \bar{A})^2 \right) \Big/ 99 \right)^{1/2}.$$

In general, instead of 99, we would use $(L-1)$, where L is the number of independent runs of the program. Then D is an unbiased estimate of the standard deviation, and $X = D/\sqrt{L}$ of the standard deviation of the mean. In this case, D turned out to be 0.0301, and so X was 0.0030. Under the normal approximation, which is valid in large samples by the central limit theorem (here the sample size is 100), we know that an approximate 95% confidence interval for the sample mean is given by $\bar{A} \pm 2X$. (The normal approximation was justified in this case by the Kolmogorov–Smirnov test [14, vol. 2, p. 41]). So with approximately 95% confidence, we can say that the expected LRU miss ratio for this probability distribution (Zipf's Law, skewness 0.5, number of pages 100) with capacity CAP = 30 is 0.6119 ± 0.0060.

4. Examples. As a demonstration of the power of current techniques (including those developed in this paper) for obtaining expected miss ratios in the independent reference model, we present a family of examples (Table 1). In each case, we consider a Zipf's law probability distribution with skewness $\theta = 0.5$. We vary the number n of pages, and we also vary the capacity CAP in such a way that the "normalized capacity" CAP/n is 0.3. All values are rounded to four decimal places. We include not only the expected FIFO and LRU miss ratios, but also the expected WS, or working-set miss ratios (Denning and Schwartz [5]), the expected A_0 miss ratios (Aho et al. [1]), and the expected VMIN miss ratios (Prieve and Fabry [20]; Slutz [22]). Here A_0 is the optimal page replacement algorithm with no knowledge of the future in the independent reference model, and VMIN is the optimal variable-space page replacement algorithm under demand paging (with lookahead).

In the case of WS and VMIN, which are variable-space page-replacement algorithms, the quantity CAP is the *expected* number of pages in main memory. For

example, in the WS case, the window size T is chosen in such a way that CAP is the expected working-set size.

The fact that the expected LRU, WS, A_0, and VMIN miss ratios have limiting values (as in Table 1) is proven in Fagin [8], where closed-form formulas are exhibited for these limits. Further, it is shown there that the limits in the LRU and WS cases are the same. (In the case of Table 1, the common limit is 0.5701.) It is an open problem as to whether there is a limiting value for the expected FIFO miss ratio, and how to find the limit.

We close this section with a minor technical comment on the LRU calculations in Table 1. Except for the $n = 10$ case, for which we used King's LRU formula, the given interval in the LRU column is approximately a 95% confidence interval. For the $n = 100$ and $n = 1000$ cases, the experiment described in § 3 was performed 100 times (that is, the L of § 3 is 100). For the $n = 10000$ case, the experiment was performed only 30 times, because of the great amount of paging which takes place when dealing with very large vectors.

TABLE 1
Expected miss ratios. (Zipf's Law, skewness $\theta = 0.5$, normalized capacity 0.3)

	FIFO	LRU	WS	A_0	VMIN
$n = 10$, CAP = 3	0.6660	0.6607	0.6599	0.5741	0.3601
$n = 100$, CAP = 30	0.6304	0.6119 ± 0.0060	0.6096	0.4870	0.2858
$n = 1{,}000$, CAP = 300	0.6091	0.5827 ± 0.0017	0.5831	0.4629	0.2706
$n = 10{,}000$, CAP = 3,000	0.6007	0.5748 ± 0.0010	0.5742	0.4556	0.2663
Limiting value	?	0.5701	0.5701	0.4523	0.2643

Appendix. The numerical stability of the FIFO algorithm. We sketch a proof that the FIFO algorithm described at the end of § 2 is numerically stable. We first show that there is not a large range in the matrix of values $F(r, m)$, where $1 \leqq r \leqq \text{CAP}+1$ and $1 \leqq m \leqq n$; that is, we show that there are not many orders of magnitude between the smallest positive entry and the largest positive entry (note that there are no negative entries, although there are zero entries). In fact, we show that the largest entry is 1, and the smallest positive entry is at least $(\min p_i)/(\text{CAP}+1)$. Therefore, there is no underflow in cases of interest. We then sketch a relative error analysis which shows that the maximum relative error in the entries $F(m, r)$ grows linearly with n (the number of pages).

Pick r_0 and m_0 so that $1 \leqq r_0 \leqq \text{CAP}+1$ and $1 \leqq m_0 \leqq n$. If $r_0 > m_0$ then $F(r_0, m_0) = 0$. So assume that $r_0 \leqq m_0$. We now show that

$$(\text{A.1}) \qquad (\min\{p_i : i = 1, \cdots, n\})/(\text{CAP}+1) \leqq F(r_0, m_0) \leqq 1.$$

Let $S = p_1 + \cdots + p_{m_0}$, and define $q_i = p_i/S$ for $1 \leqq i \leqq m_0$; that is, we take the first m_0 probabilities $p_1, \cdots, p_{m_0}$, we normalize them so that their sum is 1, and we call the normalized probabilities $q_1, \cdots, q_{m_0}$. Define $E_1(r_0, m_0)$ and $E_1(r_0 - 1, m_0)$ in the same way as we defined $E(r_0, m_0)$ and $E(r_0 - 1, m_0)$ in § 2, except that we use the probabilities $q_1, \cdots, q_{m_0}$ instead of the probabilities $p_1, \cdots, p_n$. Hence, by analogy with (2.3) we know that the expected FIFO miss ratio, using the probability distribution $q_1, \cdots, q_{m_0}$, and with capacity $r_0 - 1$ is

$$(\text{A.2}) \qquad \frac{r_0 E_1(r_0, m_0)}{E_1(r_0 - 1, m_0)}.$$

We now obtain upper and lower bounds for (A.2). Since (A.2) is the expected value of a miss ratio, it is bounded above by 1. Furthermore, (A.2) is at least as big as the expected value of the A_0 miss ratio (Aho et al. [1]) with the same capacity, where A_0 is the optimal page replacement algorithm with no knowledge of the future in the independent reference model. This expected A_0 miss ratio (with capacity r_0-1) is in turn at least as big as the expected A_0 miss ratio with capacity m_0-1 (since $(r_0-1)\leqq(m_0-1)$ by assumption). And, it is easy to check that the expected A_0 miss ratio with capacity m_0-1 is at least as big as $\min\{q_i: i=1,\cdots,m_0\}$. (Recall that these miss ratios are over the probability distribution $\{q_1,\cdots,q_{m_0}\}$.) To summarize the bounds we have obtained for (A.2), we have shown that

$$\min\{q_i: i=1,\cdots,m_0\}\leqq\frac{r_0E_1(r_0,m_0)}{E_1(r_0-1,m_0)}\leqq 1. \tag{A.3}$$

It is easy to see that $E(r_0,m_0)=S^{r_0}E_1(r_0,m_0)$, since $E(r_0,m_0)$ is a sum of products of r_0-element subsets of $\{p_1,\cdots,p_{m_0}\}$, while $E_1(r_0,m_0)$ is the corresponding sum of products of r_0-element subsets of $\{q_1,\cdots,q_{m_0}\}=\{p_1/S,\cdots,p_{m_0}/S\}$. Similarly, $E(r_0-1,m_0)=S^{r_0-1}E_1(r_0-1,m_0)$. Hence

$$\begin{aligned}F(r_0,m_0)&=\frac{E(r_0,m_0)}{E(r_0-1,m_0)}\\&=\frac{S^{r_0}E_1(r_0,m_0)}{S^{r_0-1}E_1(r_0-1,m_0)}\\&=\frac{SE_1(r_0,m_0)}{E_1(r_0-1,m_0)}.\end{aligned} \tag{A.4}$$

If, using (A.4), we substitute $F(r_0,m_0)/S$ for $E_1(r_0,m_0)/E_1(r_0-1,m_0)$ in (A.3), then after multiplying all parts of the resulting inequality by S/r_0, we obtain

$$(S/r_0)\min\{q_i: i=1,\cdots,m_0\}\leqq F(r_0,m_0)\leqq S/r_0. \tag{A.5}$$

Now $(S/r_0)\leqq 1$ since $S\leqq 1$ and $r_0\geqq 1$, and so it follows from (A.5) that $F(r_0,m_0)\leqq 1$, which establishes our upper bound on F. (In fact, this upper bound is attained, since $F(1,n)=1$.) As for the lower bound: we know that $\min\{q_i: i=1,\cdots,m_0\}=\min\{p_i: i=1,\cdots,m_0\}/S\geqq\min\{p_i: i=1,\cdots,n\}/S$. Furthermore, $r_0\leqq\mathrm{CAP}+1$, and so from (A.5) we obtain

$$\min\{p_i: i=1,\cdots,n\}/(\mathrm{CAP}+1)\leqq F(r_0,m_0), \tag{A.6}$$

which gives our lower bound on F. (This lower bound is actually attained when $p_i=1/n$ for each i, when $r_0=\mathrm{CAP}+1$, and when $m_0=n$.)

Having shown that there is no underflow (in cases of interest), we can now analyze the propagation of relative error. (If A is a quantity and A' is the calculated value of A, then the *relative error* is $(A'-A)/A$; note that the relative error can be positive, negative, or zero.) The key to stability for our algorithm is the fact that if the relative error in $F(r,m-1)$ is ε_1, and if the relative error in $F(r-1,m-1)$ is ε_2, then the relative error in

$$F(r,m)=\frac{F(r,m-1)+p_m}{F(r-1,m-1)+p_m}F(r-1,m-1)$$

is smaller in magnitude than the maximum of the magnitudes of ε_1 and ε_2. Why is this? For notational convenience, write A for $F(r,m-1)$, B for $F(r-1,m-1)$, and C

for p_m. We assume for now that C has no relative error. It is an important combinatorial fact that $F(r-1, m-1) \geqq F(r, m-1)$; this is Theorem 53 in Hardy, Littlewood and Polya [12, p. 52]. Therefore, $B \geqq A$. Let A' and B' be the calculated values of A and B respectively; thus, $A' = A(1+\varepsilon_1)$, and $B' = B(1+\varepsilon_2)$. Then the relative error in $F(r, m)$, that is, the relative error in $(A+C)B/(B+C)$, is given by

$$\left(\frac{(A'+C)B'}{(B'+C)} \Big/ \frac{(A+C)B}{(B+C)}\right) - 1. \tag{A.7}$$

Since (A.7) is the relative error in $F(r, m)$, our goal is to show that the absolute value of (A.7) is bounded above by $\max(|\varepsilon_1|, |\varepsilon_2|)$.

If we replace A' by $A(1+\varepsilon_1)$ and B' by $B(1+\varepsilon_2)$ in (A.7), it is easily verified that the resulting expression equals

$$\left(\frac{\varepsilon_1(1+\varepsilon_2)(B+C)A}{(A+C)} + \varepsilon_2 C\right) \Big/ ((1+\varepsilon_2)B + C). \tag{A.8}$$

The absolute value of (A.8) is bounded above by

$$\left(\frac{|\varepsilon_1|(1+\varepsilon_2)(B+C)A}{(A+C)} + |\varepsilon_2|C\right) \Big/ ((1+\varepsilon_2)B + C), \tag{A.9}$$

where we have assumed that $|\varepsilon_2| < 1$ (so that $1+\varepsilon_2$ is positive). It follows immediately from $B \geqq A$ that $(B+C)A/(A+C) \leqq B$. Therefore, (A.9) is bounded above by the expression obtained by replacing $(B+C)A/(A+C)$ in (A.9) by B. That is, (A.9) is bounded above by

$$(c_1|\varepsilon_1| + c_2|\varepsilon_2|)/(c_1 + c_2), \tag{A.10}$$

where $c_1 = (1+\varepsilon_2)B$ and $c_2 = C$. But (A.10) is a weighted average of $|\varepsilon_1|$ and $|\varepsilon_2|$, and so is bounded above by their maximum. This is what we wanted to show.

If we now take into consideration the effect of roundoff and the fact that the constants p_m may also be in error due to the fact that they may not be represented exactly, then we find that the maximum relative error in the entries $F(r, m)$ is a small constant times the number n of columns of F, times ε, where ε is the inherent roundoff error in a floating-point number. (Thus, ε is 2^{-t}, where t is the number of bits in the representation of the mantissa of a floating-point number.)

Acknowledgment. The authors are grateful to Gerry Shedler for useful comments that led to improved exposition.

REFERENCES

[1] A. V. AHO, P. J. DENNING AND J. D. ULLMAN, *Principles of optimal page replacement*, J. Assoc. Comput. Mach., 18 (1971), pp. 80–93.

[2] O. I. AVEN, L. B. BOGUSLAVSKII AND YA. A. KOGAN, *Some results on distribution-free analysis of paging algorithms*, IEEE Trans. Computers, C-25 (1976), pp. 737–745.

[3] F. BASKETT AND A. RAFII, *The A_0 inversion model of program paging behavior*, Computer Science Rep. CS-76-579, Stanford University, Nov. 1976, Comm. ACM, to appear.

[4] L. A. BELADY, *A study of replacement algorithms for a virtual-storage computer*, IBM Systems J., 5 (1966), pp. 78–101.

[5] P. J. DENNING AND S. C. SCHWARTZ, *Properties of the working-set model*, Comm. ACM, 15 (1972), pp. 191–198.

[6] M. C. EASTON, *Model for interactive data base reference string*, IBM J. Res. and Devel., 19 (Nov. 1975), pp. 550–556.

[7] R. FAGIN, *A counterintuitive example of computer paging*, Comm. ACM, 19 (1976), pp. 96–97. (*Corrigendum*: Comm. ACM, 19 (1976), p. 187.)

[8] ———, *Asymptotic miss ratios over independent references*, J. Comput. System Sci., 14 (1977), pp. 222–250.

[9] R. FAGIN AND M. C. EASTON, *The independence of miss ratio on page size*, J. Assoc. Comput. Mach., 23 (1976), pp. 128–146.

[10] P. A. FRANASZEK AND T. J. WAGNER, *Some distribution-free aspects of paging algorithm performance*, Ibid., 21 (1974), pp. 31–39.

[11] E. GELENBE, *A unified approach to the evaluation of a class of replacement algorithms*, IEEE Trans. Computers, C-22 (1973), pp. 611–618.

[12] G. H. HARDY, J. E. LITTLEWOOD AND G. PÓLYA, *Inequalities*, Cambridge University Press, London, 1964.

[13] W. F. KING, III, *Analysis of paging algorithms*, IFIP Conf. Proc. Ljubljana, Yugoslavia (Aug. 1971).

[14] D. E. KNUTH, *The Art of Computer Programming*, vol. 2, 1969, and vol. 3, 1973, Addison-Wesley, Reading, MA.

[15] P. A. W. LEWIS AND G. S. SHEDLER, *Empirically derived micromodels for sequences of page exceptions*, IBM J. Res. Devel., 17 (1973), pp. 86–100.

[16] P. A. LEWIS AND P. C. YUE, *Statistical analysis of program reference patterns in a paging environment*, Proc. IEEE Conf. Computers, Boston, 1971.

[17] A. W. MADISON AND A. P. BATSON, *Characteristics of program localities*, Comm. ACM, 19 (1976), pp. 285–294.

[18] R. MATTSON, J. GECSEI, D. SLUTZ AND I. TRAIGER, *Evaluation techniques for storage hierarchies*, IBM Systems J., 9 (1970), pp. 78–117.

[19] P. H. ODEN AND G. S. SHEDLER, *A model of memory contention in a paging machine*, Comm. ACM, 15 (1972), pp. 761–771.

[20] B. G. PRIEVE AND R. S. FABRY, *VMIN—an optimal variable-space page replacement algorithm*, Ibid., 19 (1976), pp. 295–297.

[21] J. RODRIGUEZ-ROSELL, *Empirical data reference behavior in data base systems*, IEEE Computer, 9 (1976), no. 11, pp. 9–13.

[22] D. R. SLUTZ, *A relation between working set and optimal algorithms for segment reference strings*, IBM Research Rep. RJ 1623, July 1975, San Jose, California.

[23] J. R. SPIRN AND P. J. DENNING, *Experiments with program locality*, AFIPS Conf. Proc. 41, (1972), pp. 611–621.

[24] P. C. YUE AND C. K. WONG, *On the optimality of the probability ranking scheme in storage applications*, J. Assoc. Comput. Mach., 20 (1973), pp. 624–633.

[25] G. K. ZIPF, *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Cambridge, MA, 1949.

SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0005 $01.00/0

IMPLEMENTATION AND ANALYSIS OF BINOMIAL QUEUE ALGORITHMS*

MARK R. BROWN†

Abstract. The binomial queue, a new data structure for implementing priority queues that can be efficiently merged, was recently discovered by Jean Vuillemin; we explore the properties of this structure in detail. New methods of representing binomial queues are given which reduce the storage overhead of the structure and increase the efficiency of operations on it. One of these representations allows any element of an unknown priority queue to be deleted in log time, using only two pointers per element of the queue. A complete analysis of the average time for insertion into and deletion from a binomial queue is performed. This analysis is based on the result that the distribution of keys in a random binomial queue is also the stationary distribution obtained after repeated insertions and deletions.

Key words. analysis of algorithms, binomial queue, priority queue, heap

Introduction. A *priority queue* is a structure for maintaining a collection of items, each having an associated key, such that the item with the smallest key is easily accessible. More precisely, if Q is a priority queue and x is an item containing a key from a linearly-ordered set, then the following operations are defined:

Insert (x, Q)	Add item x to the collection of items in Q.
DeleteSmallest (Q)	Remove the item containing the smallest key among all items in Q from Q; return the removed item.

These actions are referred to informally as *insertion* and *deletion.*

A *mergeable priority queue* is a priority queue with the additional property that two disjoint queues can be combined quickly into a single queue. That is, the operation

Union (T, Q)	Remove all items from T and add these items to Q.

is defined when T and Q are mergeable priority queues; this operation is informally referred to as *merging* T into Q. Any pair of priority queues can be merged by using repeated applications of Insert and DeleteSmallest, but the qualification "mergeable" is generally reserved for those priority queues which can be merged quickly: merging should not require examining a positive fraction of the items in the queues.

The priority queue is recognized as a useful abstraction due to the large number of applications in which it arises [1], [2], [13]. Priority queues are also interesting simply because a number of subtle data structures have been devised for representing them, including heaps [13], leftist trees [13], and 2-3 trees with heap ordering [1]; the last two of these structures are mergeable. Recently the *binomial queue*, a new data structure for implementing mergeable priority queues, was described by Vuillemin [21]. This structure does not improve upon the asymptotic time bounds already known for the operations it performs, but is interesting because of its intrinsic beauty and simplicity, and because it uses less storage than other methods.

One goal of this paper is to show that the binomial queue is not just another pretty data structure, but is the most practical structure for priority queues in many situations. Section 1 defines binomial queues, and § 2 gives algorithms for operating on them. Section 3 discusses the underlying structures which seem most suitable for

* Received by the editors October 29, 1976, and in revised form May 27, 1977.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520. This research was supported by a National Science Foundation graduate fellowship at Stanford University.

implementing binomial queues under various assumptions. One of the structures allows an arbitrary element of an unknown priority queue containing m elements to be deleted in $O(\log m)$ time, using only two pointers per element of the queue. (Implementation of the priority queue primitives, using two of the structures introduced in § 3, are given in [2].)

A second goal of the paper is to give a complete average-case analysis of the time required for insertion and deletion using binomial queues. In § 4 we show that our intuitive sense of the binomial queue's simplicity is reflected by the fact that binomial queues do not degenerate from their "random" state (the state brought about by consecutive random insertions) when deletions occur. This leads to the analysis of binomial queue algorithms given in § 5. The results of this analysis also help to establish the practicality of binomial queues by aiding in a comparison between binomial queues and other structures; the conclusion is that binomial queues are the fastest known mergeable priority queue organization, and may have some advantages even when fast merging is not required.

1. Binomial trees, forests and queues. For each $k \geqq 0$ we define a class B_k of ordered trees as follows:

(1.1) Any tree consisting of a single node is a B_0 tree.

(1.2) Suppose that Y and Z are disjoint B_{k-1} trees for $k \geqq 1$. Then the tree obtained by adding an edge to make the root of Y become the leftmost offspring of the root of Z is a B_k tree.

A *binomial tree* is a tree which is in class B_k for some k; the integer k is called the *index* of such a binomial tree. Binomial trees have appeared several times in the computer literature: they arise implicitly in backtrack algorithms for generating combinations [15]; B_0 through B_4 trees are shown explicitly in an algorithm for prime implicant determination [17]; a B_5 tree is given as the frontispiece for [11]; and oriented binomial trees, called S_n trees, were used by Fischer in an analysis of set union algorithms [6].

It should be clear from the definition above that all binomial trees having a given index are isomorphic in the sense that they have the same shape. Figure 1 illustrates rule (1.2) for building binomial trees, and Figure 2 displays the first few cases.

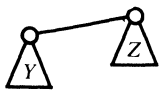


FIG. 1. *Construction of a binomial tree.*

An alternative construction rule, equivalent to (1.2), is often useful:

(1.3) Suppose that $Z_{k-1}, \cdots, Z_0$ are disjoint trees such that Z_l is a B_l tree for $0 \leqq l \leqq k-1$. Let R be a node which is disjoint from each Z_l. Then the tree obtained by taking R as the root and making the roots of $Z_{k-1}, \cdots, Z_0$ the offspring of R, left to right in this order, is a B_k tree.

Figure 3 illustrates rule (1.3) for building binomial trees. The equivalence of (1.2) and (1.3) follows by induction on k.

For future reference we record some properties of binomial trees, including the property which originally motivated their name:

LEMMA 1. *Let Z be a B_k tree. Then*

(i) *Z has 2^k nodes;*

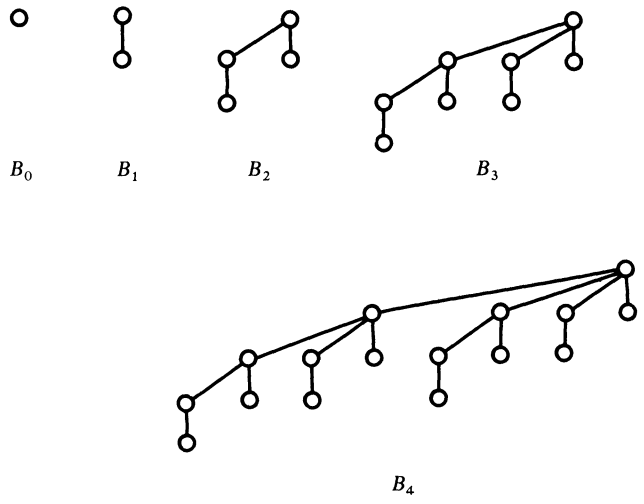


FIG. 2. *Small binomial trees.*

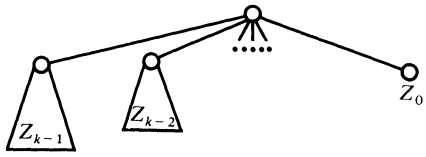


FIG. 3. *Alternative construction of a binomial tree.*

(ii) *Z has $\binom{k}{l}$ nodes on level l.*

Proof. The proof is trivial by induction on k. □

For each $m \geqq 0$ we define a *binomial forest* of size m to be an ordered forest of binomial trees with the properties:

(1.4) The forest contains m nodes.

(1.5) If a B_k tree Y is to the left of a B_l tree Z in the forest, then $k > l$. (That is, the indices of trees in the forest are strictly decreasing from left to right.)

Since by (1.5) the indices of all trees in the forest are distinct, the structure of a binomial forest of size m can be encoded in a bit string $b_l b_{l-1} \cdots b_0$ such that b_j is the number (zero or one) of B_j trees in the forest. By Lemma 1, the number of nodes in the forest is $\sum_{j \geqq 0} b_j \cdot 2^j$; hence $b_l b_{l-1} \cdots b_0$ is just the binary representation of m. This shows that a binomial forest of size m exists for each $m \geqq 0$, and that all binomial forests of a given size are isomorphic. Figure 4 shows some small binomial forests.

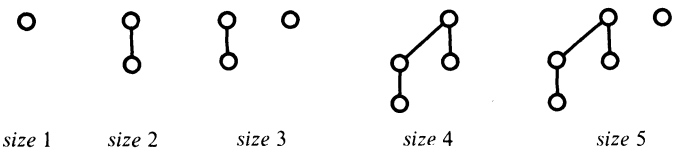


FIG. 4. *Small binomial forests.*

LEMMA 2. *Let F be a binomial forest of size $m > 0$. Then*

(i) *the largest tree in F is a $B_{\lfloor \lg m \rfloor}$ tree;*

(ii) *there are $\nu(m) = (\#$ of 1's in binary representation of $m)$ trees in F; this is at most $\lfloor \lg (m+1) \rfloor$ trees;*

(iii) *There are $m - \nu(m)$ edges in F.*

Proof. The proof is obvious. □

Consider a binomial forest of size m such that each node has an associated *key*, where a linear order $\leqq$ is defined on the set of possible key values. This forest is a *binomial queue* of size m if each binomial tree of the forest is *heap-ordered*: no offspring has a smaller key than its parent. This implies that no node in a tree has a smaller key than the root. Figure 5 gives an example of a binomial queue.

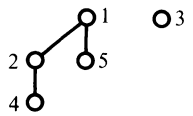


FIG. 5. *A binomial queue of size 5 (with integer keys).*

To avoid dwelling on details at this point, we shall defer discussion of representations for binomial queues until later sections. The timing bounds we give here and in the next section can only be fully justified by reference to a specific representation, but the bounds should be plausible as they stand.

The following propositions relating to binomial queues are essential:

LEMMA 3. *Two heap-ordered B_k trees can be merged into a single heap-ordered B_{k+1} tree in constant time.*

Proof. We use construction rule (1.2). The merge is accomplished by first comparing the keys of the two roots, then adding an edge to make the larger root become the leftmost son of the smaller. (Ties can be broken in an arbitrary way.) This process requires making a single comparison and adding a single edge to a tree; for an appropriate tree representation this requires constant time. □

LEMMA 4. *Let T be a heap-ordered B_k tree. Then the forest consisting of subtrees of T whose roots are the offspring of the root of T is a binomial queue of size $2^k - 1$.*

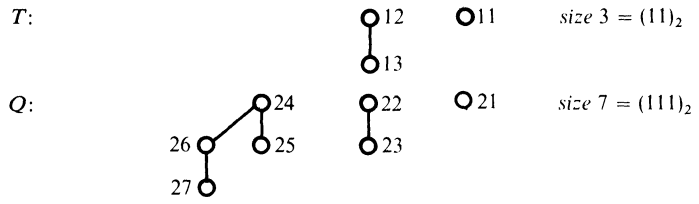
Proof. This follows immediately from construction rule (1.3) and the fact that subtrees of a heap-ordered tree are heap-ordered. □

2. Binomial queue algorithms. In order to implement a mergeable priority queue using binomial queues, we must give binomial queue algorithms for the operations Insert, DeleteSmallest and Union which were defined in the Introduction. In the following informal description of the algorithms we let $\|Q\|$ denote the number of elements in a queue Q.

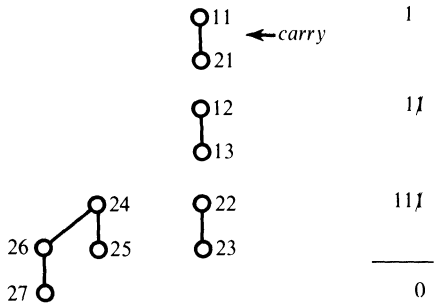
Consider first the operation Union (T, Q), which merges the elements of T into Q. If $\|T\| = t$ and $\|Q\| = q$, then the process of merging the binomial queues for T and Q is analogous to the process of adding t and q in binary. We successively "add" pairs of heap-ordered B_k trees, as described in Lemma 3, for increasing values of k. In the initial step there are at most two B_0 trees present, one from each queue. If two are present, merge (add) them to produce a single B_1 tree, the carry. In the general step, there are at most three B_k trees present: one from each queue and a carry. If two or more are present we add two of them and carry the result, a B_{k+1} tree. Each step of this procedure requires constant time, and by Lemma 2 there are at most $\max(\lfloor \lg (t+1) \rfloor, \lfloor \lg (q+1) \rfloor)$ steps. Hence the entire operation requires

$O(\max(\log \|T\|, \log \|Q\|))$ time. Figure 6 gives an example of Union with binomial queues.

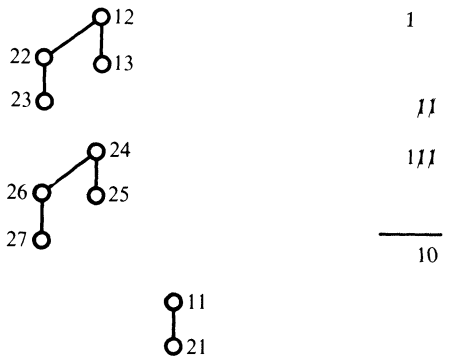
Given the ability to perform Union, the operation Insert (x, Q), which adds item x to queue Q, is trivial to specify: just let X be the binomial queue containing only the item x, and perform Union (X, Q). By this method, the time required for an insertion into Q is $O(\log \|Q\|)$.



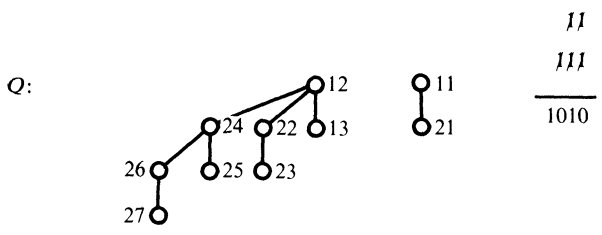
(a) *Binomial queues of size* 3 *and* 7 *to be merged for Union operation.*



(b) *After merge of* B_0*'s; result is carry.*



(c) *After merge of* B_1*'s.*



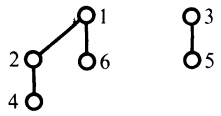
(d) *Merge completed.*

FIG. 6. *Binomial queue Union operation.*

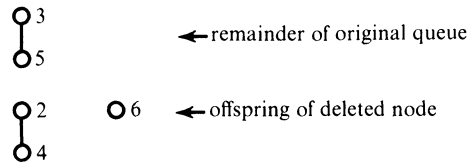
The operation DeleteSmallest (Q) is a bit more complicated. The first step is to locate the node x containing the smallest key. Since x is the root of one of the queue's B_k trees, it can be found by examining each of these roots once. By Lemma 2 this requires $O(\log \|Q\|)$ time.

The second step of a deletion begins by removing the heap-ordered B_k tree T containing x from the binomial queue representing Q. Then T is partially dismantled by deleting all edges leaving the root x; this results in a binomial queue T' of size $2^k - 1$, as suggested by Lemma 4, plus the node x which will be returned as the value of DeleteSmallest.

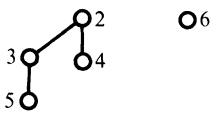
The final step consists of merging the two queues formed in the second step: the queue T' formed from T, and the queue Q' formed by removing T from Q. Since each queue is smaller than $\|Q\|$, the operation Union (T', Q') requires $O(\log \|Q\|)$ time; therefore the entire deletion requires $O(\log \|Q\|)$ time. Figure 7 gives an example of DeleteSmallest with binomial queues.



(a) *Binomial queue of size* 6. *Node* 1 *is to be deleted.*



(b) *Two queues which result from removing node* 1.



(c) *Resulting queue of size* 5 *after merging.*

FIG. 7. *DeleteSmallest on a binomial queue.*

In some situations it is useful to be able to delete an arbitrary element of a priority queue, not just the smallest. It is possible to accomplish this with binomial queues by generalizing the tree-dismantling step of DeleteSmallest. Suppose x is the node to be deleted, where x is contained in the B_k tree T. Referring back to Fig. 1, we can decompose T into two B_{k-1} trees Y and Z. Now x lies in either Y or Z, and it lies in Y if and only if the root of Y is on the path from x to the root of T. So we remove the edge joining Y and Z, save the tree which does not contain x, and repeat the process on the tree containing x until x stands alone as a B_0 tree. When the process terminates, k subtrees have been saved, and they consistitue a binomial queue of size $2^k - 1$. (Note that when x is the root of T, this procedure just deletes all edges leaving x.) The deletion is completed with a final Union, as before; the same time estimates also apply as long as we can delete each edge in constant time during the tree-dismantling step.

It should be mentioned that there is an alternative deletion algorithm for

binomial queues which is analogous to deletion from a heap [13]. The first step of this DeleteSmallest procedure is to locate the node x containing the smallest key in the queue Q. Then x is removed from the tree T containing it, and if T was the smallest binomial tree in Q then the procedure terminates. Otherwise we remove the root node y from the smallest tree in Q and make y the root of T. At this point T is a binomial tree, but may no longer be heap-ordered; thus we "sift down" the node y by repeatedly exchanging y with the smallest of its offspring until y is smaller than all of its offspring. This establishes heap-order in T and completes the deletion process. Unfortunately, the siftdown step is relatively expensive: in a B_k tree, it requires $k-1$ comparisons to find the smallest offspring of the root, and may require $k-2$ comparisons to find the smallest offspring of this node, etc. Hence a worst-case siftdown may use $O(\log^2 \|Q\|)$ steps. (It should be clear how to generalize this DeleteSmallest procedure to handle arbitrary deletions in $O(\log^2 \|Q\|)$ steps.)

It is interesting to note that the time bound given for the Insert operation can be substantially improved if we study the effect of several consecutive insertions. Consider the sequence of instructions

$$\text{Insert } (x_1, Q); \text{ Insert } (x_2, Q); \cdots; \text{ Insert } (x_k, Q).$$

The time for each insertion is just $O(1)+O(\text{number of edges created by the insertion})$. If $\|Q\| = m$ initially, the number of edges created by this sequence of instructions is $(m+k-\nu(m+k))-(m-\nu(m)) = k+\nu(m)-\nu(m+k)$ by Lemma 2. Hence the time for k insertions into a queue is $O(k)+O(k+\nu(m)-\nu(m+k)) = O(k+\log m)$ if the queue has size m initially.

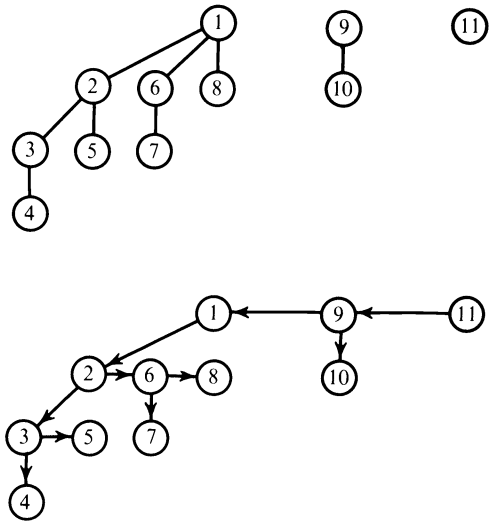
As mentioned in the Introduction, leftist trees and 2-3 trees can be used to implement mergeable priority queues. The time bounds for Insert, DeleteSmallest and Union using these structures have the same order of magnitude as those given above for binomial queues. But for both of these structures, insertions must be handled in a special way in order to achieve the $O(k+\log m)$ time bound for a sequence of Insert instructions. The naive approach, that of inserting elements individually into the leftist or 2-3 tree, can cost about $\log (k+m)$ per insertion for a total cost of $O(k \log (k+m))$. The faster approach is to buffer the insertions by maintaining the newly inserted elements as a forest of trees with graduated sizes, such as powers of two. Then insertions can be handled by balanced merges, just as with binomial queues. Individual merges require more than constant time, but the time for k insertions comes to $O(k+\log m)$.

3. Structures for binomial queues. In implementing binomial queues our objectives are to make the operations described in the previous section as efficient as possible while requiring a minimum of storage for each node. As usual, the most appropriate structure may depend on which operations are to be performed most frequently.

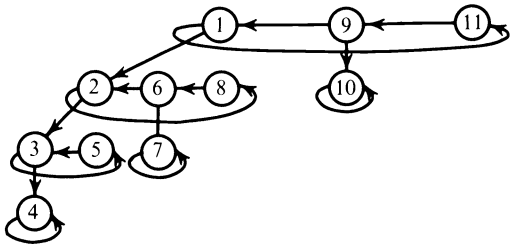
Since a binomial queue is a forest, it is natural to represent it as a binary tree [11]. But not all orientations of the binary tree links allow binomial queue operations to be performed efficiently. Evidently the individual trees of the binomial forest must be linked together from smaller to larger, in order to allow "carries" to propagate during the Union operation. But in order to allow two heap-ordered binomial trees to be merged in constant time, it seems necessary that the root of a binomial tree contain a pointer to its leftmost child; hence the subtrees must be linked from larger to smaller. This structure for binomial queues was suggested by Vuillemin [21]; we shall call it structure V. An example of a binomial queue and its representation using structure V is given in Fig. 8(a).

The time bounds given in the preceding section for Insert, DeleteSmallest, and Union can be met using structure V, provided that the queue size is available during these operations. The queue size is necessary in order to determine efficiently the sizes of the trees in the queue as they are being processed. (The alternative is to store in each node the size of the tree of which it is the root; this will generally be less useful than keeping the queue size available, and it will use more storage.) In what follows we shall assume that the queue size is available as part of the *queue header*; the other component of the queue header will be a pointer to the structure representing the queue.

One drawback of structure V for binomial queues is that the direction of the top-level links is special. This means that in this representation, the subforest consisting of trees whose roots are offspring of the root of a binomial tree is not represented as a binomial queue (as would be suggested by Lemma 4); the top level links are backwards. Structure R, the ring structure shown in Fig. 8(b), eliminates this problem. In this structure the horizontal links point to the left, except that the leftmost tree among a group of siblings points to the rightmost. Downward links point to the

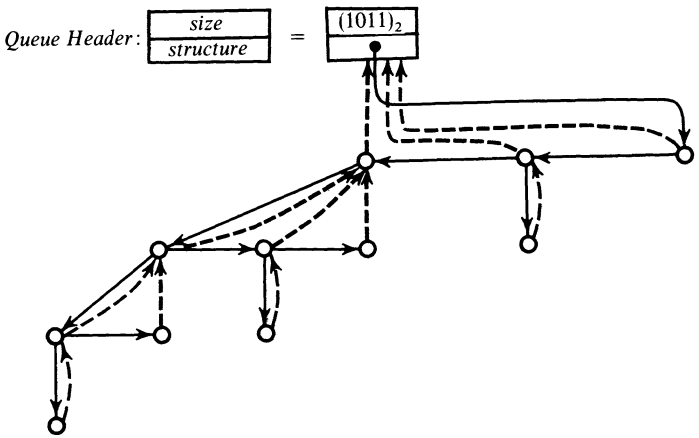


(a) *A binomial queue and its representation using structure V.*

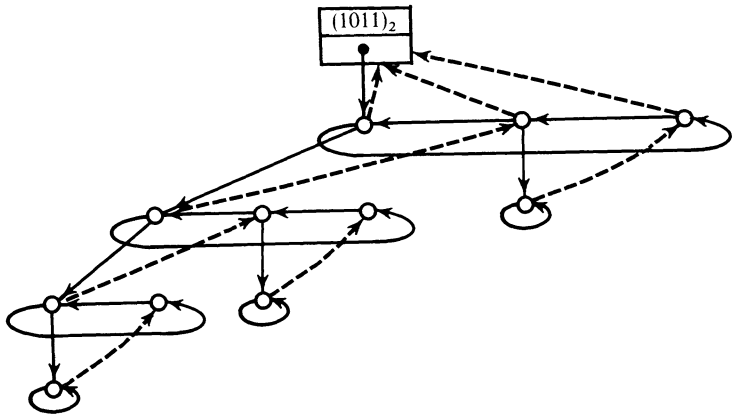


(b) *A representation for the same queue using structure R.*

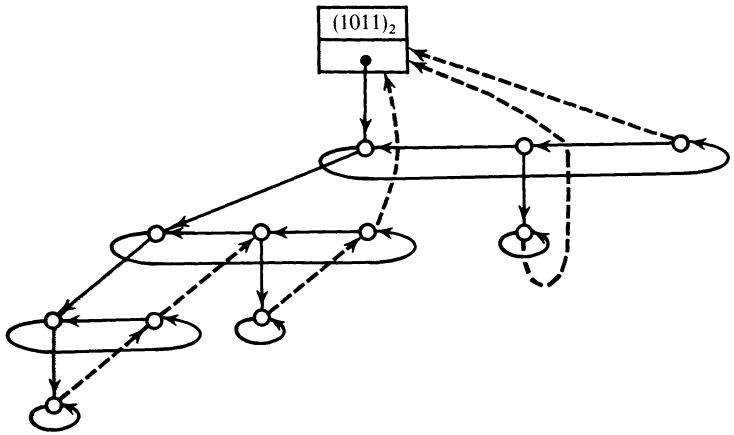
FIG. 8. *Structures for binomial queues.*



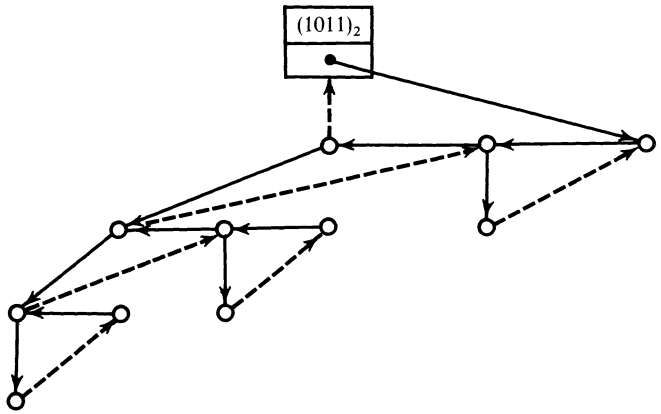
(a) *Structure V with upward pointers.*



(b) *Structure R with upward pointers.*



(c) *A structure with only two pointers per node.*



(d) *Structure K.*

FIG. 9. *Structures for binomial queues allowing arbitrary deletions.*

leftmost (largest) subtrees, as in structure V. It appears that structure R is slightly inferior to structure V for insertions, but is enough better for deletions to make it preferable for most priority queue applications. Structure V has some advantages for implementing the fast minimum spanning tree algorithm [3], since the ordering of subtrees helps to limit stack growth in that algorithm. (The stack can be stored in a linked fashion using the deleted nodes, thereby removing this objection to structure R.)

Neither of the structures described so far allows an arbitrary node to be deleted from a binomial queue, given only a pointer to the node. It is evident that in order for this to be possible, the structure must contain upward pointers of some sort which allow the path from any node to the root of the tree containing it to be found quickly. It must also be possible to find the queue header, since it will change during a deletion.

Simply adding a pointer from each node to its parent node (to the queue header in case of a root) in structure V results in a structure which allows arbitrary deletions to be performed. An example is given in Fig. 9(a). Starting from any node in this structure, it is possible to follow the upward links and trace the path to the root of the binomial tree containing the node. The upward link from the root leads to the queue header, which we assume is distinguishable in some way from a queue node. Once the path to the root is known, the top-down deletion procedure described in the preceding section can be applied.

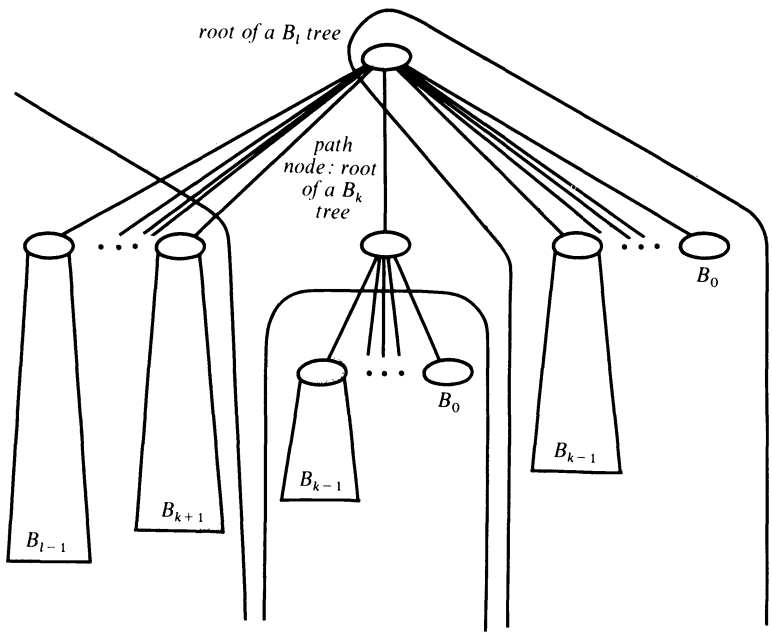


FIG. 10. *One step of the bottom-up deletion process.*

While the top-down deletion process is easy to describe, a more efficient bottom-up procedure would be used in practice. It is also essential to understand the bottom-up procedure in order to comprehend how alternative structures can be used. In the initial step of the bottom-up procedure we save all of the trees whose roots are offspring of the node to be deleted, and call this node the *path node.* (See Fig. 10.) In the general step the path node was originally the root of a B_k tree within the binomial tree being dismantled; its parent was the root of a B_l tree, and we have saved $B_{k-1}, \cdots, B_0$ trees so far. We first save the B_k tree formed by the right siblings of the path node, taking the path node's parent as a root. Then we save the $B_{k+1}, \cdots, B_{l-1}$ trees which are left siblings of the path node, and make the parent of the path node the new path node. When the path node becomes the root, the process terminates. The forest of trees saved by this process is the same as that created by the top-down process, and the remaining steps of the two algorithms are identical.

Figure 9(b) shows a modification of structure R which allows arbitrary deletions to be performed. This structure keeps an upward pointer only in the leftmost node among a group of siblings, and this pointer indicates the right sibling of the parent of nodes on this level. Note that the rightmost sibling in any family has no offspring, so the parent's right sibling always exists when needed. It is not too hard to convince oneself that the bottom-up deletion procedure just described can be performed on this structure. During the deletion process, nodes are visited in a zig-zag path moving upward to the queue header. We first move left among a group of siblings until the leftmost is reached, and then move up to the next level using the upward pointer. These steps are repeated until the queue header is reached.

Figure 9(c) shows a method of encoding the previous structure which uses only two pointers per node. The regularity of the binomial tree structure allows us to recover the information about which "child" pointers actually point upward, as follows: the rightmost node in any of the horizontal rings has no offspring (except perhaps on the top level of the forest), so its "child" pointer goes upward. If a node is an only child, or is the right sibling of a node having an only child, then it is one of these rightmost nodes. A node is an only child if and only if it is its own left sibling, so it is possible to test efficiently whether or not a "child" pointer goes upward. The upward pointer convention in Fig. 9(c) is slightly irregular at the top levels; here the decoding depends on our ability to distinguish the queue header from other nodes.

Structure K, another structure which allows arbitrary deletions using only two pointers per node, is shown in Fig. 9(d). This structure contains some null links, and seems to require less pointer updating per operation than the structure in Fig. 9(c). Note that a path from an arbitrary node to the queue header can be found by always following "left" links, some of which go upwards. To move to the right on a given level we just follow the child pointer and then the "left" pointer.

4. Random binomial queues. We define a *random binomial queue* of size m to be the queue formed by choosing a random permutation of $\{1, 2, \cdots, m\}$ and inserting the permutation's elements successively into an initially empty binomial queue. (By a *random permutation* we mean a permutation drawn from the space in which all $m!$ permutations are equally likely.) Equivalently, a random binomial queue of size m is formed from a random binomial queue of size $m-1$ by choosing a random element x from $\{1-\frac{1}{2}, 2-\frac{1}{2}, \cdots, m-\frac{1}{2}\}$, inserting x into the queue, and *renumbering* the queue such that the keys come from $\{1, 2, \cdots, m\}$ and the ordering among nodes is preserved.

This definition of a random queue is simple, yet is not artificial. The second statement of the definition, which says that the mth random insertion falls with equal probability into each of the m intervals defined by keys in the queue, is equivalent to another definition of random insertion which arises from event list applications. In these situations, a random insertion is obtained as follows: generate an independent random number X from the negative exponential distribution, in which the probability that $X \leqq x$ is $1-e^{-x}$. Then insert the number $X+t$, where t is the key most recently removed from the queue (0 if no deletions have taken place). Here t is interpreted as the current instant of simulated time, and X is a random "waiting time" to the occurrence of some event. The fact that this definition of a random insertion is equivalent to the one we have adopted was proved by Jonassen and Dahl [8]; it follows without difficulty from the well-known "memoryless" property of the exponential distribution.

Our goal in this section is to study the structure of random binomial queues. The gross structure of such a queue is already evident; we observed earlier that all binomial forests of a given size are isomorphic. But more information about the distribution of keys in the forest is necessary to fully analyze the performance of binomial queue algorithms. For example, in order to analyze the behavior of DeleteSmallest it is necessary to determine the probability of finding the smallest element in the various trees of the binomial queue. It is also important to determine whether or not a random queue stays random after a DeleteSmallest has been performed, since if this is true then the analysis of random queues may apply even in situations where both insertions and deletions are used to build the queue.

Our first observation is that the insertion algorithm shows a certain indifference to the sizes of the elements inserted.

LEMMA 5. *Let $p = p_1 p_2 \cdots p_m$ be a permutation of $\{1, 2, \cdots, m\}$. Then in the binomial queue obtained by inserting $p_1, p_2, \cdots, p_m$ successively into an initially empty queue, the tree containing p_j is determined by j for $j = 1, 2, \cdots, m$.*

Proof. We proceed by induction on m. The result is obvious for $m = 1$. For $m > 1$, let $l = 2^{\lfloor \lg m \rfloor}$ be the largest power of two less than or equal to m. After the first l elements of p have been inserted, the queue consists of a single $B_{\lfloor \lg m \rfloor}$ tree. Later insertions have no effect on this tree, since it can only be merged with another tree of equal size. Hence the first l elements of p must fall into the leftmost tree of the queue. Furthermore, since the leftmost tree is not touched, the remaining $m-l$ insertions distribute the last elements of p into smaller trees as if the insertions were into an empty queue. This proves the result by induction. □

A quicker but less suggestive proof of Lemma 5 simply notes that comparisons between keys in the insertion algorithm only affect the relative placement of subtrees in the tree being constructed. Such comparisons never determine which tree is to receive a given node.

What the given proof of Lemma 5 says is that the input permutation p can be partitioned into blocks whose sizes are distinct powers of two, such that the 2^k elements of block b_k form a B_k tree when all m insertions are complete. The sizes of these blocks decrease from left to right, just as the sizes of trees in the forest decrease. (Another priority queue structure with this sort of indifferent behavior is an unsorted linear list; with the linear list, the blocks are all of size one.)

The deletion algorithm exhibits a similar dependence on when the deleted item was inserted, and a similar indifference to key sizes. What the following lemma states is that if we delete an element from a binomial queue, then the resulting queue is the same as we obtain by never inserting the element at all, but permuting the elements

that we *do* insert in a manner which depends only on when the deleted element was inserted.

LEMMA 6. *Let $p = p_1 p_2 \cdots p_m$ be a permutation of $\{1, 2, \cdots, m\}$. Then there is a mapping $r = r_{j,m}$ from $\{1, 2, \cdots, m-1\}$ onto $\{1, 2, \cdots, j-1, j+1, \cdots, m\}$ such that the result of inserting $p_1 p_2 \cdots p_m$ into an initially empty binomial queue and then deleting p_j is identical to the result of inserting $p_{r(1)} p_{r(2)} \cdots p_{r(m-1)}$ into an initially empty binomial queue.*

Proof. We basically mimic the procedure for deleting p_j and then read the mapping from the result. The exact mapping depends on arbitrary choices made during the merging process and would be tedious to exhibit for general j and m, so we will give an example of the construction for $m = 10$, $j = 3$. First the input is divided into blocks as described above.

$$[0\ 0\ \bullet\ 0\ 0\ 0\ 0\ 0][\ ][0\ 0][\ ].$$

Then the block containing j, which holds all elements of the binomial tree T containing j in the queue, is further divided to exhibit the subtrees produced when T is dismantled.

$$[(0\ 0)\ \bullet\ (0)(0\ 0\ 0\ 0)][\ ][0\ 0][\ ].$$

This division clearly depends only on m and j.

Following the dismantling step is a merging step. Since the results of the deletion depend on details of the merging strategy, we must choose one; a possible strategy for this merge is as follows. If the dismantled binomial tree T was the smallest tree in the original queue, then no merging is required. Otherwise combine the smallest tree in the original queue with the forest just obtained by dismantling T. This produces a new tree which has the same size as T had, plus a forest of small trees; the merge is then complete. The same effect would be created (in the case we are considering) by reinserting all nodes in the order

$$(0\ 0\ 0\ 0)(0\ 0)[0\ 0](0).$$

To see this, just simulate the insertion process on this input. The intermediate trees created during this process correspond to trees involved in the merge. (Note that the r map is far from being uniquely determined.) □

Here again we can draw an analogy with the unsorted linear list, which obviously has the behavior stated in the lemma.

Armed with this result, we can determine the effects of various types of deletions on random binomial queues.

THEOREM 1. *Let Q be a random binomial queue of size m. Suppose that p_k, the k-th element inserted in the process of building Q, is deleted from Q and Q is renumbered. Then the resulting Q is a random binomial queue of size $m-1$.*

Proof. Consider the $m!$ equally-likely permutations used to build Q. When the kth element of each permutation is discarded and the permutation renumbered, each of the $(m-1)!$ possible permutations occurs m times. The same is true if some fixed rearrangement of the permutation is made just before the renumbering. Hence by Lemma 6 the $m!$ queues obtained by inserting all possible permutations of length m and then deleting the kth element (and renumbering) are just m copies of the $(m-1)!$ queues obtained by inserting all permutations of length $m-1$. □

THEOREM 2. *Let Q be a random binomial queue of size m. Suppose that k, the k-th smallest element inserted in the process of building Q, is deleted from Q and Q is renumbered. Then the resulting Q is a random binomial queue of size $m-1$.*

Proof. Consider the $m!$ equally-likely permutations used to build Q. For fixed j, there are $(m-1)!$ of these permutations with $p_j = k$; if we ignore p_j and renumber, we get all $(m-1)!$ possible permutations of $\{1, 2, \cdots, m-1\}$. The same is true if some fixed rearrangement of the permutation is made before renumbering. Hence by Lemma 6 the $(m-1)!$ queues obtained by inserting all permutations of length m with $p_j = k$ and then deleting k (and renumbering) are just the $(m-1)!$ queues obtained by inserting all permutations of length $m-1$. This is true for each j, so the result follows. □

COROLLARY 1. *If a random element (or randomly* placed *element) of the input is deleted from a random binomial queue of size m, the result is a random binomial queue of size* $m-1$.

Proof. The two statements are obviously equivalent; they follow immediately from Theorem 1 or Theorem 2. □

These results are sufficient to show that binomial queues stay random in many situations. The most important of these is when a queue is formed by a sequence of n random Insert operations intermixed with $m \leqq n$ occurrences of DeleteSmallest, arranged so that a deletion is never attempted when the queue is empty. Theorem 2 shows that a DeleteSmallest applied to a random queue leaves a random queue; a random Insert also preserves randomness. So under the most reasonable assumptions for priority queues, binomial queues can be treated as random. This is our rationale for assuming random binomial queues in the analysis of the next section.

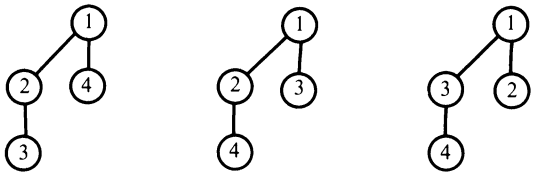
A similar argument shows that random binomial queues result when intermixed random deletions are performed; a simple argument appealing to Lemma 6 shows that intermixed deletions by age (how long an element has been in the queue) also lead to random queues. These types of deletions, especially deletions by age, are somewhat artificial for priority queues.

It is natural to ask whether randomness is preserved by the merging of binomial queues. Suppose that a random permutation of length m is given; its first k elements are inserted into one initially empty binomial queue, and the remaining $m-k$ elements are inserted into another. Then each of these queues is a random binomial queue, and the argument used to prove Lemma 6 shows that the result of merging these queues is also random as long as some fixed choice is made about which two trees to "add" when three are present during the merge. So in this sense merging does preserve randomness.

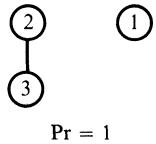
Sensitivity to deletions has been studied in the context of binary search trees by Knott [10] and Jonassen and Knuth [9]. The model used there considers a random insertion to be the insertion of a random real number drawn independently from some continuous distribution (for example, uniform on the interval [0, 1]). This definition is *not* equivalent to ours; Theorem 1 and Corollary 1 hold for deletions from binary search trees, but this does not imply that a tree built using intermixed random deletions is random. In fact, as Knott first noted, binary search trees are sensitive to deletions in this model.

Binomial queues, however, are not sensitive to deletions in the search tree model. In a general study of deletion insensitivity, Knuth showed that Theorem 2 implies insensitivity to random deletions, and Lemma 6 implies insensitivity to deletions by age in this model [14]. Binomial queues are sensitive to deletions by order (e.g., DeleteSmallest) in this model, but unsorted linear lists, as well as practically all other algorithms, are also sensitive to these deletions. So even with this alternative definition of a random insertion, random binomial queues tend to remain random when deletions are present.

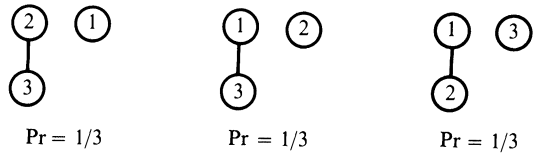
At this point it might seem that *nothing* can destroy a random binomial queue! This is not true; a deletion based on knowledge of the structure of the queue (or equivalently, knowledge of the entire input) can easily introduce bias. For example, random queues of size 4 are distributed as shown:



If we now delete the rightmost child of the root and renumber, we get:



This isn't random; random binomial queues of size 3 have the distribution



Since the analysis of binomial queue algorithms performed in the next section is based on random binomial queues, we are interested in the distribution of keys in these queues. By Lemma 5, the probability that a given element (e.g. the smallest) of a random permutation lies in a given binomial tree is simply the probability that the element lies in a certain block of positions within the permutation. Thus the probability that the jth largest element in a binomial queue of size m lies in a B_k tree is just $2^k/m$, assuming that a B_k tree is present in a queue of size m. This decompositon of the input into blocks reduces the study of random binomial queues to the study of random heap-ordered binomial trees (i.e., random binomial queues of size 2^k).

As we observed earlier, the smallest key in a heap-ordered binomial tree must be in the root. The distribution of larger keys is not so highly constrained. The following result characterizes the distribution of keys without explicit reference to the $n!$ possible input permutations.

THEOREM 3. *Let a* configuration *of a heap-ordered B_k tree be any assignment of the integers $1, 2, \cdots, 2^k$ to the nodes of a B_k tree such that the tree is heap-ordered. Then in a random heap-ordered B_k tree all $(2^k)!/2^{(2^k)-1}$ configurations are equally likely. (That is, there are $2^{(2^k)-1}$ distinct input permutations which generate each possible configuration.)*

Proof. We proceed by induction on k. The result is obvious for $k = 0$. Assume that for $k = j$ there are $2^{(2^j)-1}$ permutations of $\{1, 2, \cdots, 2^j\}$ which give rise to each possible configuration.

Now consider any fixed configuration X of a B_{j+1} tree. This tree can be decomposed into the two B_j trees Y and Z, as shown in Fig. 1. By the argument of Lemma 5,

any permutation giving rise to configuration X must consist of two blocks, one producing Y and the other Z; these blocks may appear in either order, since the relative position of Y and Z is determined by which tree contains the smallest key. By the induction hypothesis there are $2^{(2^i)-1}$ arrangements of the keys in tree Y which give rise to Y, and similarly for Z. So there are $2 \cdot 2^{(2i)-1} \cdot 2^{(2i)-1} = 2^{(2i+1)-1}$ permutations which produce X. Since this holds for any X, the result follows. □

In the inductive step above, we can note that the element 1 is equally likely to be contained in the first or the second block of a permutation producing X. This leads to an easy inductive proof of the proposition that the ith inserted element is equally likely to fall into each of the 2^k nodes of a random heap-ordered B_k tree.

Unfortunately, Theorem 3 does not help much in determining the exact distribution of keys in a random binomial tree. There are fewer configurations than permutations, but the number of configurations still increases rapidly with k.

5. Analysis of binomial queue algorithms. We are now prepared to analyze the performance of Insert and DeleteSmallest, when implemented using binomial queues; this will allow a comparison with other priority queue organizations. The binomial queue implementation to be analyzed is based on structure R, discussed in § 3 and pictured in Fig. 8(b). The priority queue structures to be used for comparison are the heap, leftist tree, sorted linear list, and unsorted linear list. (We do not perform a detailed comparison with 2-3 trees because they seem to be inferior to leftist trees when merging is needed, and to heaps when it is not.)

For each of the five structures, the operations Insert and DeleteSmallest have been carefully coded in the assembly language of a typical computer (the binomial queue and leftist tree implementations appear in [2]). By inspecting these programs, we can write expressions for their running time as a function of how often certain statements are executed. It then remains to determine the average values of these factors.

The running times (in memory references for instructions and data) of the binomial queue operations are

Insert	$16+19M+2E+6A$
DeleteSmallest	$38+11B+6T+4N-2L+4S+14U+2X$

where

- M is the number of merges required for the insertion;
- E is the number of exchanges performed during these merges in order to preserve the heap-order property;
- A is 1 if $M = 0$, and 0 otherwise;
- B is 1 if the queue contains no B_0 tree, and 0 otherwise;
- T is the number of binomial trees in the queue;
- N is the number of times that the value of the smallest key seen so far is changed during the search for the root containing the smallest key;
- L is 1 if the smallest key is contained in the leftmost root, and 0 otherwise;
- S is the number of offspring of the root containing the smallest key;
- U is the number of merges required for the deletion; and
- X is the number of exchanges performed during these merges.

(To keep the expression for DeleteSmallest simple, certain unlikely paths through the program have been ignored. The expression above always overestimates the time required for these cases.)

As a first step in the analysis we note that several of the factors above depend only on the structure of the binomial queue Q and not on the distribution of keys in Q.

Since the structure of Q is determined solely by its size, these factors are easy to determine. For example, if Q has size m then evidently M is the number of low-order 1 bits in the binary representation of m, and $A = 1$ if and only if m is even. Clearly $B = A$, and by Lemma 2 we can see that $T = \nu(m)$.

These factors are a bit unusual in that they do not vary smoothly with m. For example, when $m = 2^n - 1$ we have $M = T = n$, while for $m = 2^n$ this changes to $M = 0$ and $T = 1$. Since in practice we are generally concerned not with a specific queue size m but rather with a range of queue sizes in the neighborhood of m, it makes sense to average the performance of our algorithms over such a neighborhood.

Factors A and M can be successfully smoothed by this approach: averaging over the interval $[m/2, 2m]$ gives an expected value of $1/2 + O(1/m)$ for A and $1 + O((\log m)/m)$ for M. This agrees well with our intuition, since it says that about half of the integers in the interval are even, and that one carry is produced, on the average, by incrementing a number in the interval.

Properties of the factor $T = \nu(m)$ have been studied extensively. From [16] we find that

$$\left\lfloor \frac{1}{2} m \lg\left(\frac{3}{4} m\right) \right\rfloor \leqq \sum_{1 \leqq k \leqq m} \nu(k) \leqq \left\lfloor \frac{1}{2} m \lg m \right\rfloor,$$

where each bound is tight for infinitely many m; it follows that our neighborhood averaging process will not completely smooth the sequence $\nu(m)$. But we have bounds on an "integrated" version of $\nu(m)$, so differentiating the bounds puts limits on the average growth rate of $\nu(m)$. Carrying out the differentiation gives

$$\frac{1}{2}\left(\lg m + \frac{1}{\ln 2} - \lg\frac{4}{3}\right) \leqq T_{\text{avg.}} \leqq \frac{1}{2}\left(\lg m + \frac{1}{\ln 2}\right),$$

which is about what we expect: half of the bits are 1, on the average. The remaining uncertainty in the constant term is about .21.

While this averaging technique fails to smooth the sequence $\nu(m)$ completely, there are other methods which succeed. There is no single "correct" method for handling problems of this type: different techniques may give different answers, and the usefulness of a result depends on how "natural" the smoothing method is in a given context. The more powerful averaging techniques which succeed in smoothing $\nu(m)$ seem artificial in connection with our analysis, but the results are quite interesting mathematically. Lyle Ramshaw [20] has shown that

$$\nu(m) \cong \frac{\lg m}{2} + \left(\frac{\lg \pi}{2} - \frac{1}{4}\right) \doteq \frac{\lg m}{2} + 0.57575$$

using logarithmic averaging [5]; his result is based on the detailed analysis of $\sum_{1 \leqq k \leqq m} \nu(k)$ performed by Hubert Delange [4]. The naturalness of logarithmic averaging is indicated by the fact that it also leads to the logarithmic distribution of leading digits which has been observed empirically [19], [12, § 4.2.4], and the fact that it is consistent with several other averaging methods (such as repeated Cesaro summing) when those methods define an average.

This analysis of factor T completes the purely "structural" analysis; the remaining factors depend on the distribution of keys in the queue. For the average-case analysis we shall assume that Q is a random binomial queue of size m and that the insertion is random. These assumptions are well justified by the discussion of § 4.

The factors E and X are easy to dispose of. Since we only merge trees of equal size, our randomness assumptioin says that an exchange is required on half of the merges (on the average). More precisely, if there are n merges then the number of exchanges is binomially distributed with mean $n/2$ and variance $n/4$. The number of merges is just M in the case of E, and U in the case of X.

The factors L and S are also easy to analyze. We noted in § 4 that the probability of having the queue's smallest key in a given tree is just proportional to the size of the tree. Therefore if there is a binomial tree of size 2^k in a queue of size m, this tree contains the queue's smallest key with probability $2^k/m$. The root of such a binomial tree has k offspring by Lemma 1, so the expected value of S is $(1/m)F(m)$ where

$$F(m)=\sum_{\substack{k\geq 0\\ m=(b_l b_{l-1}\cdots b_0)_2}} b_k\cdot k\cdot 2^k.$$

While it seems hard to find a simpler closed form for $F(m)$, it is possible to derive good upper and lower bounds.

LEMMA 7. *The function $F(m)$ defined above satisfies $\lceil m\lg m-2m\rceil\leq F(m)\leq \lfloor m\lg m\rfloor$, $m\geq 1$, and the upper bound is tight for infinitely many values of m.*

Proof. (This argument is similar to the one used to prove Theorem 1 in [16].) From the definition of $F(m)$, if $m=2^k$ then

$$F(m)=F(2^k)=k\cdot 2^k=m\lg m.$$

It is also clear from the definition that

$$F(2^k+i)=F(2^k)+F(i),\qquad 0\leq i<2^k.$$

The upper bound on $F(m)$ is evidently attained whenever m is a power of two. It therefore holds when $m=1$, and assuming that it holds up to $m=2^k$, we have

$$\begin{aligned}F(m+i)&=F(2^k)+F(i)\qquad (0\leq i<2^k)\\ &\leq m\lg m+i\lg i\\ &\leq (m+i)\lg(m+i).\end{aligned}$$

So the upper bound holds for all m by induction.

The lower bound on $F(m)$ holds when $m=1$, and whenever m is a power of two. Suppose that a bound of the form

$$F(m)\geq m\lg m-cm$$

is true for some $c>0$ and all $m\leq 2^k$. Then

$$\begin{aligned}F(m+i)&=F(2^k)+F(i)\qquad (0\leq i<2^k)\\ &\geq m\lg m+i\lg i-ci.\end{aligned}$$

It follows that if the inequality

$$m\lg m+i\lg i-ci\geq (m+i)\lg(m+i)-c(m+i)\qquad (0\leq i<2^k)$$

holds, the lower bound will hold for all m by induction. Replacing i by xm and simplyfying gives another inequality which implies the result:

$$x\lg x\geq (1+x)\lg(1+x)-c\qquad (0\leq x<1).$$

But it is easy to verify that $x\lg x-(1+x)\lg(1+x)$ is decreasing on $[0,1]$, so we can take $x=1$ to determine $c=2$. (A tight lower bound can be found by using the value $F(2^k-1)=(k-2)2^k+2$.) □

According to these bounds, the average value of S lies between $\lg m - 2$ and $\lg m$. Lyle Ramshaw [20] has shown that the logarithmically averaged value of S is

$$\frac{F(m)}{m} \cong \lg m - C$$

where

$$C = \left(\frac{1}{\ln 2} \sum_{n \geq 1} \sum_{j > 2^n} \frac{(-1)^{j+1}}{j}\right) + \frac{1}{2} \doteq 1.10995.$$

The expected value of L is $2^{\lfloor \lg m \rfloor}/m$, which is between $1/2$ and 1.

Factor U is closely related to S. The number of merges required is equal to the number of trees (i.e., S) created by removing the node containing the smallest key, minus the number of these trees which are not merged. Since the first merge must take place with the smallest tree remaining in the original forest, we see that the number of trees excluded from merging is equal to the number of low-order 0 bits in m. Since the least significant bits of m are distributed almost uniformly, the average value of this quantity $S - U$ is the same as the average value of M.

Factor N is more interesting. One way to search for the smallest root in the forest is to use the key contained in the rightmost root as an initial estimate for the smallest key, and then scan the forest from right to left, updating the estimate as smaller keys are seen. Since the trees increase in size from right to left, trees in the left of the forest are more likely to contain the smallest key; thus the estimate of smallest key will be changed often during the scan. To be more precise, the expected number of changes while searching a forest of size $m = (b_n b_{n-1} \cdots b_0)_2$ is

$$\sum_{\substack{0 \leq k \leq n \\ b_k = 1}} \Pr(\text{estimate changes when the } B_k \text{ tree is examined})$$

$$= \sum_{\substack{0 \leq k \leq n \\ b_k = 1}} \frac{(\text{number of nodes in the } B_k \text{ tree})}{(\text{total number of nodes in all } B_l \text{ trees examined, } 0 \leq l \leq k)}$$

$$= \sum_{\substack{0 \leq k \leq n \\ b_k = 1}} \frac{2^k}{\sum_{0 \leq l \leq k,\, b_l = 1} 2^l}.$$

When $m = 2^n - 1$ this has the simple from

$$\frac{2}{3} + \frac{4}{7} + \frac{8}{15} + \cdots + \frac{2^{n-1}}{2^n - 1}$$

$$= \left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \cdots + \frac{1}{2}\right) + \frac{1}{2}\left(\frac{1}{3} + \frac{1}{7} + \frac{1}{15} + \cdots + \frac{1}{2^n - 1}\right)$$

$$= \frac{n}{2} + \frac{1}{2}(\alpha - 1) + O(2^{-n})$$

where

$$\alpha = \sum_{k \geq 1} \frac{1}{2^k - 1} = 1.60669\ldots.$$

(The constant α also arises in connection with Heapsort; see [13, 5.2.3(19)].)

A search strategy which intuitively seems superior to the one just described is to search the forest from left to right; for the above example the expected number of changes is reduced to

$$\frac{1}{3}+\frac{1}{7}+\frac{1}{15}+\cdots+\frac{1}{2^n-1}=\alpha-1+O(2^{-n}).$$

But this strategy is not practical; the links point in the wrong direction. With structure R we can improve the search by using the key contained in the leftmost root as our initial estimate in a right to left search. This makes the expected number of changes in a queue of size 2^n-1 equal to

$$\frac{1}{2^{n-1}+1}+\frac{2}{2^{n-1}+2+1}+\frac{4}{2^{n-1}+4+2+1}+\cdots+\frac{2^{n-2}}{2^n-1}+\frac{2^{n-1}}{2^n-1}$$

By writing this sum in reverse order we can derive its asymptotic value:

$$\begin{aligned}
&\frac{2^{n-1}}{2^n-1}+\frac{2^{n-2}}{2^n-1}+\frac{2^{n-3}}{2^n-2^{n-2}-1}+\frac{2^{n-2}}{2^n-2^{n-2}-2^{n-3}-1}+\cdots\frac{1}{2^{n-1}+1}\\
&\quad=\frac{1/2}{1-2^{-n}}+\frac{1/4}{1-2^{-n}}+\frac{1/8}{1-1/4-2^{-n}}+\frac{1/16}{1-1/4-1/8-2^{-n}}+\cdots+\frac{1/2^n}{1/2+2^{-n}}\\
&\quad=\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{2}\left(\frac{1}{3}+\frac{1}{5}+\cdots+\frac{1}{2^{\lfloor n/2\rfloor}+1}\right)\right)(1+O(2^{-n/2}))+O(2^{-n/2})\\
&\quad=\frac{3}{4}+\frac{1}{2}\left(\alpha'-\frac{1}{2}\right)+O(2^{-n/2})
\end{aligned}$$

where

$$\alpha'=\sum_{k\geq 0}\frac{1}{2^k+1}=1.26449\ldots.$$

(The constant α' arises in connection with merge sorting; see [13, exercise 5.2.4-13].) So the expected value of this factor is about 1.13 for large n; by modifying the search in this way we have effectively removed part of the inner loop.

This completes the analysis of Insert and DeleteSmallest for binomial queues. By plugging our average values into the running time expressions given above and simplifying, we get the results for binomial queues shown in Fig. 11. A much simpler analysis [11, pp. 94–99] gives the corresponding results for sorted and unsorted linear lists (also shown in Fig. 11).

Priority queue algorithms based on heaps and leftist trees have not been completely analyzed; partial results are known for heaps [13], [18] but not for leftist trees. Therefore experiments were performed to estimate the average values of factors controlling the running time of these algorithms. (Note that the experiments did not consist of simply running the programs and timing them, but rather of running suitably instrumented programs which kept track of the number of times certain statements were executed. The averages computed in this way were then used in the expression for the program's running time, just as the mathematically derived averages were used in the case of binomial queues and linear lists.) Leftist trees and heaps are deletion sensitive, so the averages were taken from stationary structures (obtained after repeated insertions and deletions) rather than from random structures. Figure 11 gives the experimentally determined running times for leftist trees and heaps.

Average case running times when $\|Q\| = m$.

Queue	Insert (x, Q)	DeleteSmallest (Q)	Insert (x, Q); *DeleteSmallest* (Q)
binomial queue	39	$22 \lg m + 19$	$22 \lg m + 58$
leftist tree	$17 \lg m + 35$	$35 \lg m - 27$	$52 \lg m + 8$
linear list	19	$6m + 2 \lg m + 20$	$6m + 2 \lg m + 39$
heap	32	$18 \lg m + 1$	$18 \lg m + 33$
sorted list	$3m + 17$	15	$3m + 32$

Worst case running times when $\|Q\| = m$.

Queue	Insert (x, Q)	DeleteSmallest (Q)	Insert (x, Q); DeleteSmallest (Q)
binomial queue	$21 \lg m + 16$	$30 \lg m + 46$	$51 \lg m + 62$
leftist tree	$32 \lg m + 23$	$64 \lg m - 7$	$96 \lg m + 16$
linear list	19	$9m + 15$	$9m + 34$
heap	$12 \lg m + 14$	$18 \lg m + 16$	$30 \lg m + 30$
sorted list	$6m + 20$	15	$6m + 35$

FIG. 11. *Comparison of methods.*

These results indicate that binomial queues completely dominate leftist trees. Not only do binomial queues require one fewer field per node, they also run faster, on the average, for $m \geqq 4$ when the measure of performance is the cost of an Insert followed by a DeleteSmallest. Linear lists are of course preferable to both of these algorithms for small m, but binomial queues are faster than unsorted linear lists, on the average, for $m \geqq 18$ at a cost of one more pointer per node. So the binomial queue is a very practical structure for mergeable priority queues.

In some applications the queue size may constantly be in a range which causes the insertion and deletion operations on binomial queues to run more slowly than our averages indicate, due to the smoothed average we computed. If the queue size can be anticipated then dummy elements added to the queue might actually speed up the algorithms. Using a redundant binary numbering scheme [7] it is possible to maintain the queue as a small number of binomial forests in such a way that each insertion is guaranteed to take only constant time. But the binomial queue algorithms as they stand still dominate algorithms using leftist trees, even if the leftist tree operations have average-case running times and the binomial queue operations always take the worst-case time. The only advantages which can apparently be claimed for leftist trees is that they are easier to implement and can take advantage of any tendency of insertions to follow a stack discipline.

The comparison of binomial queues with heaps and sorted linear lists is also interesting. The heap implementation stores pointers in the heap, instead of the items themselves; this is the usual approach when the items are large and should not be moved. In this situation heaps are slightly faster than binomial queues on the average, and considerably faster in the worst case. Heaps also save one pointer per node, so it seems that heaps are preferable to binomial queues when fast merging is not required. Binomial queues have an advantage when sequential allocation is a problem, or perhaps when arbitrary deletions must be performed. Sorted linear lists are better than both methods when m is small, but heaps are faster, on the average, when $m \geqq 30$.

Acknowledgments. The author would like to thank Robert Tarjan and Donald Knuth for helpful discussions relating to this work, and Daniel Boley, Lyle Ramshaw, and the referees for their criticisms of earlier drafts.

REFERENCES

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

[2] MARK R. BROWN, *The analysis of a practical and nearly optimal priority queue*, Ph.D. thesis, STAN-CS-77-600, Computer Science Dept., Stanford Univ., Stanford, CA, March 1977, 99 pp.

[3] DAVID CHERITON AND ROBERT ENDRE TARJAN, *Finding minimum spanning trees*, this Journal, 5 (1976), pp. 724–742.

[4] HUBERT DELANGE, *Sur la Fonction Sommatoire de la Fonction « Somme des Chiffres »*, Enseignement Math. 21 (1975), no. 1, pp. 31–47.

[5] PERSI DIACONIS, *Examples in the theory of infinite iteration of summability methods*, Technical Report No. 86, Stanford Univ., Dept. of Statistics, May 1976.

[6] MICHAEL J. FISCHER, *Efficiency of equivalence algorithms*, Complexity of Computer Computations, Raymond E. Miller and James W. Thatcher, eds., Plenum Press, New York, 1972.

[7] LEO J. GUIBAS, EDWARD M. MCCREIGHT, MICHAEL F. PLASS AND JANET R. ROBERTS, *A new representation for linear lists*, Proc. 9th Annual ACM Symposium on the Theory of Computing, Boulder, CO, 1977, pp. 49–60.

[8] ARNE JONASSEN AND OLE-JOHAN DAHL, *Analysis of an algorithm for priority queue administration*, Math. Inst., Univ. of Oslo, 1975.

[9] ARNE T. JONASSEN AND DONALD E. KNUTH, *A trivial algorithm whose analysis isn't*, J. Comput. System Sci., to appear.

[10] GARY D. KNOTT, *Deletion in binary storage trees*, Ph.D. thesis, STAN-CS-75-491, Computer Science Dept., Stanford Univ., Stanford, CA, May 1975, 93 pp.

[11] DONALD E. KNUTH, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.

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

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

[14] ———, *Deletions that preserve randomness*, IEEE Trans. Software Engrg., SE 3 (1977), pp. 351–359.

[15] DERRICK H. LEHMER, *The machine tools of combinatorics*, Applied Combinatorial Mathematics, Edwin F. Beckenbach, ed., John Wiley, New York, 1964.

[16] M. D. MCILROY, *The number of 1's in binary integers: Bounds and extremal properties*, this Journal, 3 (1974), pp. 255–261.

[17] EUGENIO MORREALE, *Computational complexity of partitioned list algorithms*, IEEE Trans Computers, TC 19 (1970), pp. 421–428.

[18] THOMAS PORTER AND ISTVAN SIMON, *Random insertion into a priority queue structure*, IEEE Trans. Software Engrg., SE 1 (1975), pp. 292–298.

[19] RALPH A. RAIMI, *The first digit problem*, Amer. Math. Monthly, 83 (1976), no. 7, pp. 521–538.

[20] LYLE RAMSHAW, personal communication.

[21] JEAN VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, to appear.

SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0006 $01.00/0

DEADLOCK PREDICTION: EASY AND DIFFICULT CASES*

E. MARK GOLD†

Abstract. The subject of this paper is the computational complexity of the *deadlock prediction problem* for resource allocation. This problem is the question "Is deadlock avoidable?" i.e. "Is there a feasible sequence in which to allocate all the resource requests?" given the current status of a resource allocation system. This status is defined by (1) the resource vector held by the banker, i.e. the quantity of resources presently available for allocation, and (2) the resource requests of the processes: Each process is required to make a *termination request* of the form "Give me resource vector y and I will eventually terminate and return resource vector z." Also, each process can make any number of *partial requests* of the form "If you can't give me y, then give me a smaller resource vector y' and I will be able to reach a point at which I can halt and temporarily return to z', although I will still need need $y-y'+z'$ to terminate."

If (1) the resources are reusable and (2) partial requests are not allowed, then Habermann's well known "banker's algorithm" solves the deadlock prediction problem in polynomial time. The results of this paper are as follows: If either constraint (1) or constraint (2) is removed, then the deadlock prediction problem is NP-complete. A generalization of the banker's algorithm solves this problem with the following constraint: For each process, the 0 vector and the profitability vectors $(z-y)$ of its partial and termination requests are linearly ordered with respect to size. This case includes systems with one type of resource or with Hansen's (1973) hierarchically partitioned reusable resources.

Key words. deadlock, resource allocation, operating systems, complexity, NP-complete, banker's algorithm

1. Introduction.

Deadlock avoidance reduces to deadlock prediction. The problem of *deadlock avoidance* for the resource allocation strategy (the *banker*) of an operation system, which was originally formulated by Habermann (1969), is familiar by now (e.g. Hansen (1973)): Initially, each process states the maximum quantity of resources which it will need to complete its job and terminate, and the quantity of resources which it will return by the time it terminates. From time to time a process (1) makes an *immediate request* for resources which it needs in order to continue, or (2) returns resources which it doesn't need presently. Each time a process makes an immediate request the banker must decide if he can safely allocate the resources now or he must delay the allocation until other processes have returned resources. The deadlock avoidance problem is to find a strategy for making these decisions which will grant immediate requests as soon as possible, while guaranteeing that the banker can eventually allocate the necessary resources to every process.

It is not necessary for the banker to remember the past. The correctness of the banker's decision at any time is determined by the current system resource status S_0, namely the resources presently in the banker's stockpile, the remaining resource requirements of the processes, and the resources they will return. Given S_0, the question "Can the banker safely allocate immediate request R now?" can be decided as follows: Let S_1 be the new system resource status which will result if R is allocated. Then R can be safely allocated now iff S_1 is "safe," i.e. iff for S_1 there is a feasible sequence in which to allocate the remaining resource requirements of the processes. Formal definitions are given in the following section.

* Received by the editors March 18, 1976, and in final revised form October 11, 1977.

† Département d'Informatique, Université de Montréal, Montréal, Québec, Canada. The author's present address is BM Box 5444, London WC1V 6XX, England. This work was performed with the aid of a grant from the National Research Council of Canada.

Thus, the deadlock avoidance problem reduces to the "deadlock prediction" problem Q-DP(S), namely the question "Is S safe?" The subject of this paper is the computational complexity of Q-DP(S).

In summary, this paper is not concerned with the immediate (resource) requests of the processes, but rather, for each process, its "termination request" and "partial requests": The termination request of each process specifies the resources required to complete its work and the resources it will return to the banker by the time it terminates. A partial request specifies the resources which a process needs to perform part of its work and reach a point where it can halt and temporarily return some resources to the banker. Since the banker is allowed to delay the allocation of immediate requests in the deadlock avoidance problem, in the deadlock prediction problem he is allowed to choose any sequence in which to allocate the partial and termination requests.

Easy vs. difficult cases. The basic result of this paper is that if S is allowed to range over all possible system resource statuses, as defined here, then Q-DP(S) is difficult, namely NP-complete. Various possible constraints on S are considered in order to investigate the question "How much can the range of S be constrained and still leave Q-DP(S) NP-complete, and for how general a range of S is Q-DP(S) easy, namely computable in polynomial time?"

Habermann's (1969) well known banker's algorithm shows that Q-DP(S) is computable in polynomial time if S is constrained by the requirements (1) partial requests are not allowed, and (2) resources are reusable (defined in § 3). This paper presents a generalization of the banker's algorithm which allows both of these requirements to be relaxed. Holt (1972) has given polynomial time algorithms which relax the second constraint. The generalization of the banker's algorithm proposed here was motivated by, and is a generalization of, Hansen's (1973) adaptation of the banker's algorithm to hierarchically partitioned resources.

Significance of NP-complete. NP-complete problems have the following properties (e.g. Karp (1975) and Aho, Hopcroft and Ullman (1974)): (1) All these problems are polynomial time equivalent in the sense that any one can be translated to any other in polynomial time. Therefore a polynomial time solution to any one of them would yield a polynomial time solution to all of them. (2) Many important, well known combinatoric problems which have been studied for a long time are included in this class. (3) No polynomial time solution has been found for any of these long-standing problems, so it is believed that NP-complete problems can't be solved in polynomial time, although this has yet to be proved.

2. Definitions of resource allocation model.

Resource status. A *resource allocation system* consists of:

1 banker,
n processes,
r resource types.

Since there are r resource types, a *resource vector* is an r-tuple of nonnegative real numbers. E.g., in the case $r = 3$ the resource vector (1, 0, 3.2) denotes 1 unit of the first type of resource and 3.2 units of the third type.

If $x = (x_1, \cdots, x_r)$ and $y = (y_1, \cdots, y_r)$ are resource vectors, then define

$x \leqq y$ means $x_i \leqq y_i$ for all i,

$x \geqq y$ means $x_i \geqq y_i$ for all i, i.e., $y \leqq x$,

$x < y$ means $x \leqq y$ and $x \neq y$, i.e., $x_i \leqq y_i$ for all i and $x_i < y_i$ for some i,
$x > y$ means $y < x$.

The system resource status S of a resource allocation system consists of the banker's status and the status of each of the processes:

$$S = (x, P_1, \cdots, P_n). \tag{1}$$

The banker's status is defined by a resource vector x which specifies the *banker's stockpile*, i.e. the resources presently available for allocation. The current status of the ith process P_i consists of a *termination request* R_{i0} and any finite number of *partial requests* $R_{i1}, \cdots, R_{im_i}$, possibly none:

$$P_i = (R_{i0}, \cdots, R_{im_i}), \qquad m_i \geqq 0. \tag{2}$$

Each request is specified by a pair of resource vectors

$$R_{ij} = (y_{ij}, z_{ij}), \tag{3}$$

such that

$$y_{i0} \geqq y_{i1} \geqq \cdots \geqq y_{im_i} \quad \text{for each process } P_i. \tag{4}$$

P_i is used ambiguously to denote either the ith process or its status.

The meaning of the requests $R_{i0}, R_{i1}, \cdots, R_{im_i}$ is as follows: The banker must eventually allocate all of y_{i0} to P_i for P_i to complete its work, return z_{i0} to the banker, and *terminate*, i.e. cease to exist. If the banker allocates $y_{i1} \leqq y_{i0}$ to P_i then P_i can perform part of its work and reach a halting point where it can return z_{i1} to the banker. P_i will need to get back z_{i1}, together with that part $y_{i0} - y_{i1}$ of y_{i0} which has not yet been allocated to P_i, in order to complete its work. If the banker allocates $y_{i2} < y_{i1}$ to P_i then P_i can perform a shorter initial segment of its work, halt, and return z_{i2} to the banker. $R_{i3}, \cdots, R_{im_i}$ are requests to perform successively shorter initial segments of P_i's work.

Also y_{ij} will be called the *cost* of R_{ij}, z_{ij} the *return* of R_{ij}, and $(z_{ij} - y_{ij})$ the *profitability* (to the banker) of R_{ij}.

In summary, a *system resource status* S is a data structure of the form

$$S = \begin{cases} x \\ (y_{10}, z_{10}) \cdots (y_{1m_1}, z_{1m_1}) \\ \vdots \\ (y_{n0}, z_{n0}) \cdots (y_{nm_n}, z_{nm_n}) \end{cases} \tag{5}$$

where x, y_{ij}, z_{ij} are r-tuples of real numbers such that

$$\text{AX1:} \quad x, y_{ij}, z_{ij} \geqq 0, \tag{6}$$

$r, n, m_1, \cdots, m_n$ are integers such that

$$\text{AX2:} \quad r \geqq 1, \qquad n, m_1, \cdots, m_n \geqq 0, \tag{7}$$

for each $i = 1, \cdots, n$,

$$\text{AX3:} \quad y_{i0} \geqq y_{i1} \geqq \cdots \geqq y_{im_i}. \tag{8}$$

Effect of allocation. The banker is *capable of allocating* request R_{ij} if the banker's stockpile is at least as large as the cost of R_{ij}, i.e. $x \geqq y_{ij}$. This paper is solely concerned with the question "Does a feasible allocation sequence exist for S?" When the

appropriate definitions are given it will be easily seen that we can limit our attention to allocation strategies such that after the banker allocates y_{ij} he waits for the ith process to return z_{ij} before making another allocation.

So, if $x \geqq y_{ij}$ the banker can allocate R_{ij} and the result is to change the system resource status S as follows:

x becomes $x+(z_{ij}-y_{ij})$;

If $j=0$, i.e. R_{ij} is a termination request, then P_i is deleted from S;

Else, i.e. $j \geqq 1$, i.e. R_{ij} is a partial request, then

$$R_{ik} \text{ for } k=j,\cdots,m_i \text{ are deleted from } S,$$

R_{ik} for $k=0,\cdots,j-1$ are modified according to y_{ik} becomes $y_{ik}+(z_{ij}-y_{ij})$.

R_{ij} is a request to obtain resources to perform an initial segment of the work of P_i which includes the shorter initial segments $R_{i,j+1},\cdots,R_{im_i}$. So allocation of R_{ij} causes $R_{i,j+1},\cdots,R_{im_i}$ to be deleted along with R_{ij}. Allocation of R_{ij} causes the banker's stockpile to be increased by the profitability $(z_{ij}-y_{ij})$ of R_{ij}. However, in order to allocate R_{ik}, $k<j$, the banker must return $(z_{ij}-y_{ij})$ to P_i. That is, the cost y_{ik} of R_{ik} is increased by $(z_{ij}-y_{ij})$.

Suppose that S satisfies the axioms AX1, AX2, AX3 of the definition of a system resource status, and the banker is capable of allocating R_{ij}, i.e. $x \geqq y_{ij}$. Then it is easily verified that the new S' which results from allocating R_{ij} will satisfy AX1, AX2, AX3.

Consider the requests $R_0, R_1,\cdots,R_m$ of process P. AX3

$$y_0 \geqq \cdots \geqq y_m \tag{9}$$

is necessary so that after allocation of R_i the new costs

$$y'_j = y_j+(z_i-y_i) \quad \text{for } j=0,\cdots,i-1 \tag{10}$$

will satisfy AX1,

$$y'_j \geqq 0 \quad \text{for } j=0,\cdots,i-1, \tag{11}$$

while preserving AX3

$$y'_0 \geqq \cdots \geqq y'_{i-1}. \tag{12}$$

Deadlock prediction problem Q-DP(S). Given system resource status S, the banker's objective is to eventually allocate the termination request R_{i0} of every process P_i. The banker is to choose his first allocation a_1 to be one of the requests R_{ij} of S which he is capable of allocating, i.e. $x \geqq y_{ij}$. Allocation a_1 changes S to S_1. Now the banker is to choose one of the requests of S_1 which he is capable of allocating to be his second allocation a_2. This changes the system resource status to S_2. And so on. The question is, is there such a sequence $a_1,\cdots,a_T$ of allocations for S which the banker is capable of allocating and which will terminate all the processes P_i, i.e. delete all requests R_{ij} from S?

An *allocation sequence* for S will denote a sequence

$$A = a_1,\cdots,a_T, \tag{13}$$

where each a_t is a request R_{ij} of S such that:

(14) each R_{ij} occurs at most once;

(15) each R_{i0} occurs exactly once;

(16) if R_{ij} and R_{ik} occur in A with $j>k$, then R_{ij} occurs in A before R_{ik}.

A *feasible allocation sequence* for S will denote an allocation sequence for S such that at each time $t = 1, \cdots, T$ after allocating $a_1, \cdots, a_{t-1}$ the banker will be capable of allocating a_t.

S will be called *safe* if there is a feasible allocation sequence for S, *unsafe* otherwise. The *deadlock prediction problem* is the predicate Q-DP(S) = "S is safe."

3. Constraints on system resource statuses.

Classification of requests. A request will be classified as a producer or consumer if its allocation increases or decreases the banker's stockpile, i.e. if it is profitable or unprofitable to the banker to allocate the request:

R_{ij} *produces resource type* k means $(z_{ij})_k - (y_{ij})_k \geqq 0$.

R_{ij} is a *producer* means $z_{ij} - y_{ij} \geqq 0$, i.e. R_{ij} produces all resource types.

R_{ij} is a *consumer* means $z_{ij} - y_{ij} \leqq 0$.

R_{ij} is a *mixture* means R_{ij} is neither a producer nor a consumer, i.e. $(z_{ij})_k - (y_{ij})_k > 0$ for some k and $(z_{ij})_k - (y_{ij})_k < 0$ for some k.

Thus, R_{ij} is both a producer and a consumer if $y_{ij} = z_{ij}$. If there is only 1 resource type then every request must be a producer or a consumer.

The requests $R_0, \cdots, R_m$ of process P will be called *linearly ordered* if the set of their profitability vectors, together with the 0 vector, is linearly ordered with respect to size: For all $i, j = 0, \cdots, m$

$$(z_i - y_i) \geqq (z_j - y_j) \quad \text{or } (z_i - y_i) \leqq (z_j - y_j), \tag{17}$$

$$(z_i - y_i) \geqq 0 \quad \text{or } (z_i - y_i) \leqq 0. \tag{18}$$

Thus, if there is only 1 resource type then the requests of every process are linearly ordered.

Classification of resource types. A resource type is ordinarily called "reusable" if, upon termination, each process will return precisely the quantity it has been allocated. This definition cannot be translated into a constraint on S because the system resource status, as defined here, does not record past allocations, only the present resource status.

Given S, for the purpose of this paper the kth resource type will be called *reusable* if, for each process P_i, the termination request produces this resource type and, furthermore, produces at least as much as any of the P_i's partial requests:

$$(z_{i0})_k - (y_{i0})_k \geqq 0 \quad \text{for all } P_i, \tag{19}$$

$$(z_{i0})_k - (y_{i0})_k \geqq (z_{ij})_k - (y_{ij})_k \quad \text{for all } P_i, \quad j = 1, \cdots, m_i. \tag{20}$$

Classification of constraints on S.

NO-PART: There are no partial requests, i.e. each process P_i makes just a termination request R_{i0}.

P-C: Every request is either a producer or a consumer, i.e. none are mixtures.

LIN-ORD: Every process has linearly ordered requests.

REUSE: Every resource type is reusable.

1-RES: There is only 1 resource type.

HIER: "Hierarchically partitioned resources" are defined in § 6.

Let $XXX > YYY$ mean that the set of S which satisfy XXX properly includes the set of S which satisfy YYY, i.e. XXX is *more general* than YYY. The constraints on S which are listed above are easily seen to have the relative generalities shown in Fig. 1, with the exception of HIER, for which the proof is given in § 6.

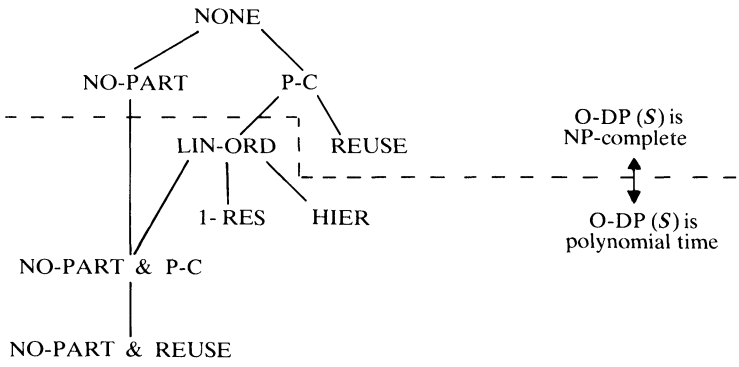


FIG. 1. *Relative generality of various constraints on S.*

4. Theorems: complexity of Q-DP(*S*). The following theorems establish the dividing line, shown in Fig. 1, between more general constraints on S for which the deadlock prediction problem Q-DP(S) in NP-complete and less general constraints for which Q-DP(S) is computable in polynomial time. Since the components of a system resource status S are defined to be real, "polynomial time" is stated formally in the theorems to mean a polynomial number of arithmetic operations (compare, add, subtract). For NP-complete to have meaning it is necessary for S to be discrete. Each of the sets of S for which it will be shown that Q-DP(S) is NP-complete will be shown to have this property with one of the following additional constraints: S is a 0-1 *system resource status* if every scalar resource quantity of S, i.e. every component of every resource vector of S, is 0 or 1. The components of a 0-2 *system resource status* are 0, 1 or 2.

The proofs of the following theorems are in the final sections.

THEOREM 1. *Let S range over* 0-1 *system resource statuses without partial requests. Then Q-DP(S) is NP-complete.*

THEOREM 2. *Let S range over* 0-2 *system resource statuses such that* (1) *all resources are reusable and* (2) *each process is allowed* 1 *partial request. Then Q-DP(S) is NP-complete.*

THEOREM 3. *Let S range over system resource statuses such that* (1) *there are no partial requests and* (2) *none of the* (*termination*) *requests is a mixture, i.e. each is a producer or a consumer. Then the banker's algorithm can be generalized to compute Q-DP(S) in* $rN \log N$ *arithmetic operations, where r is the number of resource types and N the number of requests.*

The following theorem subsumes Theorem 3.

THEOREM 4. *Let S range over the system resource statuses such that the requests of every process are linearly ordered. Then the banker's algorithm can be generalized to compute Q-DP(S) in* $rN \log N$ *arithmetic operations.*

5. Motivation and implications.

Reason for partial requests. The reason for allowing a process to make partial requests, in addition to its termination request, is as follows: If a process is only allowed to make a termination request, then there are circumstances in which the banker is forced to delay an immediate request, but could allocate it immediately if the process were allowed to state a partial request which includes a promise to return something temporarily before termination. For instance, suppose that a process were able to say "Give me x (perhaps a channel) and I will return x before asking for any

more resources." If the banker has x available, he can certainly allocate such a temporary request immediately without increasing the possibility of deadlock.

Example: partial requests for reusable resources. In the following example the resources are reusable in the ordinary sense, as opposed to the definition in § 3. That is, they are physical resources which are passed back and forth between the banker and the processes, never being created or destroyed. There are 2 resources and the resource vectors (a, b) are meant to be interpreted as a units of memory + b tape drives.

TABLE 1
Initial resource requirements of example.

Process	#1	#2
step 1	(30, 1)	(20, 2)
between steps	(20, 0)	(20, 0)
step 2	(20, 2)	(40, 2)
both steps	(30, 2)	(40, 2)

TABLE 2
Present resource distribution in example.

banker	(20, 0)
Process #1	(10, 1)
Process #2	(10, 1)

TABLE 3
Present S and results of allocations in example.

	S_0		S_1		S_2	
banker's stockpile x	(20, 0)		(10, 1)		(0, 2)	
	cost	return	cost	return	cost	return
Process #1 requests						
termination R_{10}	(20, 1) →	(30, 2)	(10, 2) →	(30, 2)	(10, 2) →	(30, 2)
partial R_{11}	(20, 0) →	(10, 1)				
Process #2 requests						
termination R_{20}	(30, 1) →	(40, 2)	(30, 1) →	(40, 2)	(20, 2) →	(40, 2)
partial R_{21}	(10, 1) →	(0, 2)	(10, 1) →	(0, 2)		

Note—S_0 is current system resource status. S_1, S_2 result from allocation of R_{11}, R_{21}.

There are 2 processes, each of which is to perform 2 steps. The resources required by each step are shown in Table 1. Each process has the option of halting temporarily after its first step and returning the resources used in the first step, except for 20 units of memory needed to retain intermediate results.

The banker initially had (40, 2), of which (10, 1) has already been allocated to each of the processes shown in Table 2. The present system resource status S_0 is shown in Table 3. Note that the termination requests show the resources required to complete both steps. The system resource status, as defined here, can't describe step 2 separately. S_1 and S_2 in Table 3 show the system resource statuses which the banker calculates will result from allocation of R_{11} followed by R_{21}. These are the only requests which the banker is capable of allocating. S_2 is deadlocked, i.e. the banker's stockpile is not adequate to allocate any of the outstanding requests. Therefore, S_0 is not safe. The purpose of this example is to make the following points:

(1) Since S_0 arose from allocation of resources which are reusable in the ordinary sense, the resources of S_0 are reusable in the sense of this paper. In particular, all termination requests are producers.

(2) Even though the resources are reusable in the ordinary sense, the partial request R_{11} of S_0 is a mixture.

(3) S_0 is unsafe in this particular case only because the definition of "system resource status" used in this paper does not allow the processes to include certain information in their request statements. Namely, after allocation of R_{11}, R_{21} it is true that the banker's stockpile will be (0, 2), but Table 1 shows that Process 1 will be holding (20, 0) between steps, so will only need (0, 2) to perform Step 2. If the banker allocates this, then he will receive back (20, 2) and be able to terminate Process 2 also. S_0 appears to be unsafe because it doesn't show that Process 1 needs less memory in Step 2 than in Step 1.

So, because the definition of system resource status used here does not allow certain types of information, the banker's decisions will be overly conservative in some cases. In this case, in the course of making immediate allocation decisions the banker would have delayed the immediate request which led to S_0, even though the actual resource disposition is safe.

(4) If the banker initially had 10 more units of memory, then x would be (30, 0) in S_0 and (10, 2) in S_2. Now S_0 is safe. However, if the partial requests weren't allowed then S_0 would be unsafe, indeed deadlocked. This shows that allowing partial requests, even of the limited type defined in this paper, permits the banker to behave less conservatively without danger of deadlock.

Reason for termination mixture requests. Nonreusable resources arise when fictitious resources are introduced to model synchronization constraints which are logically necessary. E.g., a process which is to send N messages via a small buffer will show in its termination request that it produces N messages and consumes N buffer spaces. A receiver of N messages produces N buffer spaces and consumes N messages. These are 2 examples of termination mixture requests.

Any set of termination requests, mixture or nonmixture, can arise in a system of communicating processes in which space in various buffers are the only scarce resources: The termination request "Give me some of resource types $T_1, T_2, \cdots$ and I will return some of $T_3, T_4, \cdots$" could mean "I need space to put some messages in buffers $T_1, T_2, \cdots$ and then I can finish my calculations, accepting some messages from buffers $T_3, T_4, \cdots$."

Any use of Dijkstra's P and V operations on a semaphore (e.g. Hansen (1973)) can be regarded as immediate request for, and return of, a unit of some type of resource. The associated semaphore value denotes the banker's stockpile of this type of resource.

Possible tradeoffs. Barring the apparently unlikely possibility of a polynomial time solution being found for the NP-complete problems, future research on the deadlock prediction problem, with nonreusable resources allowed or with partial requests allowed, can proceed in 3 directions: (1) fast solutions for restricted cases of this problem, as reported by Holt (1972); (2) heuristic solutions for the general problem which will "usually" solve it in polynomial time, even though exponential time will be required in the worst cases; and (3) fast algorithms for the general problem which will not always be correct but will always err on the conservative side: all unsafe system resource statuses will be correctly classified, but some safe system resource statuses will be classified unsafe. The third direction is a tradeoff between

computation time and efficiency of resource utilization, since it will cause the banker to delay some immediate requests which could be allocated immediately without causing deadlock.

6. Hierarchically partitioned resources. Hansen (1973) describes a generalization of the banker's algorithm for the deadlock prediction problem with resources which are reusable in the ordinary sense, in which partial requests of the following restricted types are allowed: The banker must partition his resources into divisions $D_0, \cdots, D_m$ a priori. Each process starts by stating the maximum quantity of resources which it will require from each of the resource divisions. The immediate requests of each process are constrained by the requirement that no additional resources be requested from D_i until all resources already allocated from $D_{i+1}, \cdots, D_m$ are returned.

I will now show that the partial and termination requests which can occur with hierarchically partitioned resources are linearly ordered. Therefore, this case is included in those solvable by the generalization of the banker's algorithm which will be described in the proof of Theorem 4 in the final section.

Consider any process P. Suppose that it originally stated that its resource vector requirements will be $u_0, \cdots, u_m$ from $D_0, \cdots, D_m$ and it has already been allocated $v_0, \cdots, v_m$. Since the resources are reusable, the present termination request of P is implicitly "Give me"

$$y_0 = (u_0 - v_0) + \cdots + (u_m - v_m), \tag{21}$$

"and I will return"

$$z_0 = u_0 + \cdots + u_m. \tag{22}$$

The constraints on the immediate requests imply the partial requests "Give me"

$$y_j = (u_j - v_j) + \cdots + (u_m - v_m), \tag{23}$$

"and I will return"

$$z_j = u_j + \cdots + u_m. \tag{24}$$

As is to be expected with reusable resources, (21, 22) shows that all termination requests are producers. (23) shows that the partial requests (23, 24) of each process satisfy AX3 in the definition of a system resource status. Finally, (23, 24) shows that the profitabilities of the requests of P

$$(z_j - y_j) = v_j + \cdots + v_m \tag{25}$$

are linearly ordered:

$$(z_0 - y_0) \geqq \cdots \geqq (z_m - y_m) \geqq 0. \tag{26}$$

The above proof that hierarchically partitioned reusable resources lead to linearly ordered requests remains valid if (1) the resources are partitioned differently for each process, and (2) the resource partitioning changes dynamically.

7. Proof of Theorem 1: termination requests is NP-complete. Q-DP(S) is NP since it can be computed by trying all allocation sequences. Let S range over system

resource statuses without partial requests. It remains to show that Q-DP(S) is NP-hard.

Let Q-CS(F) be Cook's (1970) prototype NP-complete problem, the satisfiability of propositional formulae F in conjunctive normal form: $b_1, \cdots, b_p$ are the binary variables and $C_1, \cdots, C_q$ are the clauses of F.

$$F(b_1, \cdots, b_p) = C_1 \,\&\, \cdots \,\&\, C_q, \tag{27}$$

where

$$C_i = x_{i1} \vee \cdots \vee x_{in_i}, \tag{28}$$

where each x_{ij} is a literal b_k or $\neg b_k$ for some $k = 1, \cdots, p$. Q-CS(F) = true iff there is a value assignment to $b_1, \cdots, b_p$ such that $F(b_1, \cdots, b_p)$ = true. In order to prove that Q-DP(S) is NP-hard it remains to construct a polynomial time reduction $F \to S$ such that Q-DP(S) = Q-CS(F).

Given F, for each $i = 1, \cdots, p$ let $C_{i1}^{+}, \cdots, C_{in_i}^{+}$ denote the clauses C_j of F such that $b_i \in C_j$, and $C_{i1}^{-}, \cdots, C_{in_i}^{-}$ denote the clauses C_j such that $\neg b_j \in C_j$. Now S can be constructed as follows. The resource types are $b_1, \cdots, b_p, C_1, \cdots, C_q$. The banker's stockpile is

$$x = 1 \text{ unit each of } b_1, \cdots, b_p. \tag{29}$$

For each $i = 1, \cdots, p$ there are 2 processes P_i^{+}, P_i^{-} competing for the allocation of the banker's unit of b_i: $P_i^{\pm}$ has the termination request "Give me"

$$y_i^{\pm} = 1 \text{ unit of } b_i, \tag{30}$$

"and I will return"

$$z_i^{\pm} = 1 \text{ unit each of } C_{i1}^{\pm}, \cdots, C_{in_i}^{\pm}. \tag{31}$$

There is one more process P_0 with the termination request "Give me"

$$y_0 = 1 \text{ unit each of } C_1, \cdots, C_q, \tag{32}$$

"and I will return"

$$z_0 = 1 \text{ unit each of } b_1, \cdots, b_q. \tag{33}$$

To see that S is safe iff F is satisfiable, note that for each $i = 1, \cdots, p$ the banker must decide whether to allocate his unit of b_i to P_i^{+} or to P_i^{-}. There is a feasible allocation sequence for S iff there is a way to make these p decisions such that the banker will receive in return at least 1 unit each of $C_1, \cdots, C_q$, in order to be capable of allocating the termination request of P_0. Termination of P_0 returns 1 unit each of $b_1, \cdots, b_p$ to the banker so that he can terminate the remaining P_i^{+} and P_i^{-}. There is a more detailed presentation of this type of proof in the following section.

8. Proof of Theorem 2: partial requests for reusable resources is NP-complete. Now the problem is to find a polynomial time reduction $F \to S$ such that each process of S has a termination request (y_0, z_0) which is a producer and a partial request (y_1, z_1) of any type subject to $(z_1 - y_1) \leqq (z_0 - y_0)$. Given F, define S as follows. Let the C_{ij}^{+}, C_{ij}^{-}, resource types, and the banker's stockpile x be defined as in the previous section. For $i = 1, \cdots, p$ let $P_i^{\pm}$ have the termination request "Give me"

$$y_{i0}^{\pm} = 1 \text{ unit each of } C_1, \cdots, C_q, b_i, \tag{34}$$

"and I will return"

$$z_{i0}^{\pm} = 1 \text{ unit each of } C_1, \cdots, C_q, b_i, C_{i1}^{\pm}, \cdots, C_{in_i}^{\pm}, \tag{35}$$

and the partial request "or give me only"

(36) $$y_{i1}^{\pm} = 1 \text{ unit of } b_i,$$

"and I will temporarily return"

(37) $$z_{i1}^{\pm} = 1 \text{ unit each of } C_{i1}^{\pm}, \cdots, C_{in_i}^{\pm}.$$

Let P_0 have just the termination request "Give me"

(38) $$y_{00} = 1 \text{ unit each of } C_1, \cdots, C_q,$$

"and I will return"

(39) $$z_{00} = 2 \text{ units each of } C_1, \cdots, C_q.$$

The following argument, similar to that of the previous section, shows that there is a feasible allocation sequence for S iff F is satisfiable.

Suppose there is a feasible allocation sequence A for S. Initially the banker's stockpile contains none of the C_i. In order to allocate any of the termination requests it is necessary to find a set of the partial requests which can be allocated with $x = 1$ unit each of $b_1, \cdots, b_p$ such that these allocations return at least 1 unit each of $C_1, \cdots, C_q$. As in the previous section these allocations consist of choosing for each b_i whether to allocate b_i to R_{i1}^{+} or R_{i1}^{-}. In order to satisfy F, set b_i = true or false depending on whether A allocates b_i to R_{i1}^{+} or to R_{i1}^{-}. Then all the clauses of F will be satisfied.

Conversely, suppose it is possible to set each b_i = true or false such that all the clauses of F are satisfied. Then it is possible to allocate each b_i to the partial request of either P_i^{+} or P_i^{-} in such a way tha the banker will receive at least 1 unit of each C_j. The banker can then allocate P_0, so he will have at least 2 units of each C_j. For those $P_i^{\pm}$ such that the partial request $R_{i1}^{\pm}$ was allocated, the termination request becomes "Give me"

(40) $$y_{i0}^{\pm} = 1 \text{ unit each of } C_i, \cdots, C_q, C_{i1}^{\pm}, \cdots, C_{in_i}^{\pm},$$

"and I will return"

(41) $$z_{i0}^{\pm} = 1 \text{ unit each of } C_1, \cdots, C_q, C_{i1}^{\pm}, \cdots, C_{in_i}^{\pm}, b_i.$$

So, the banker will be capable of allocating any of these termination requests. They are producers, so he will be capable of allocating all of them. Then the banker will have at least 1 unit of each b_i and each C_j. So he will be capable of allocating the termination request of any of the remaining $P_i^{\pm}$. They are producers, so he will be capable of allocating all of them.

9. Proof of Theorem 3: generalization of banker's algorithm for nonmixture termination requests.

Motivation. In this section the discussion is limited to system resource statuses with only termination requests. The following section will allow linearly ordered partial requests.

The original banker's algorithm applies to termination requests for reusable resources. That is, all requests are producers. The algorithm constructs a sequence of allocations by choosing the successive processes P_i to be terminated in any order subject only to the constraint that when P_i is terminated the banker has enough resources to do so. S is safe iff a feasible allocation sequence is found on the first try.

The basic idea of the banker's algorithm is that if system resource status S has a request R which is a producer, and the banker is capable of allocating R, then

immediate allocation of R can't hurt. Namely, if there is a feasible allocation sequence for S

$$a_1, \cdots, a_i, R, a_{i+2}, \cdots, a_N, \tag{42}$$

then there is a feasible allocation sequence for S such that R is the first allocation:

$$R, a_1, \cdots, a_i, a_{i+2}, \cdots, a_N. \tag{43}$$

Allocating R can only increase the banker's stockpile. So, if the banker is capable of allocating $a_1, \cdots, a_i$ before allocating R then he will still be capable of allocating $a_1, \cdots, a_i$ after allocating R.

More generally, if there is a feasible allocation sequence for termination requests then any producer allocation can be moved earlier and any consumer allocation can be moved later. So, in seeking a feasible allocation sequence for S the banker can limit his attention to sequences which allocate the producers first and the consumers last. The banker's algorithm can be used to determine if the producers can be allocated first. To determine if the consumers can be allocated last, reverse the direction of time: Start the banker with the stockpile which results from the allocation of all the requests of S and ask if he can "deallocate" the consumers. In this time-reversed dual problem the consumers become producers, because they are deallocated rather than allocated, so the banker's algorithm can be used.

In summary, producers should be allocated first, consumers last, and mixtures in the middle. The producers can be allocated in any feasible order. In the time-reversed dual problem the consumers can be deallocated in any feasible order. However, if there are mixtures then Theorem 1 says that it is difficult to determine if there is a feasible order for allocating them.

Generalized banker's algorithm. Let S contain only nonmixture termination requests. Then S is safe iff it passes the following 3 tests:

(1) Construct S' by deleting consumers from S. Use the banker's algorithm to determine if S' is safe.

(2) Compute the banker's stockpile after allocating all (termination) requests:

$$x_f = x + (z_{10} - y_{10}) + \cdots + (z_{n0} - y_{n0}). \tag{44}$$

For S to be safe it is necessary that

$$x_f \geqq 0. \tag{45}$$

(3) Construct the *time-reversed dual status* S'' for the consumers of S: In S'' the banker's stockpile is $x'' = x_f$; and for each consumer request (y, z) of S, S'' contains the producer request $(y'', z'') = (z, y)$. Use the banker's algorithm to determine if S'' is safe.

Proof of validity of generalized banker's algorithm. It only remains to show that a system resource status S without partial requests is safe iff its time-reversed dual S^* is safe. Suppose that there is a feasible allocation sequence for S which consists of allocating the requests

$$(y_{10}, z_{10}) \cdots (y_{n0}, z_{n0}) \tag{46}$$

in that order. Then the time-reversed dual allocation sequence

$$(z_{n0}, y_{n0}) \cdots (z_{10}, y_{10}) \tag{47}$$

is feasible for S^*. To see this, note that the definition of a feasible allocation sequence can be rephrased as follows: Start with banker's stockpile x, then subtract y_{10}, then

add z_{10}, then subtract y_{20}, and so on. The banker's stockpile must be $\geqq 0$ after each subtraction. If the dual allocation sequence is applied to S^*, then the banker's stockpiles after the subtractions will be the same in the reverse order.

The same argument shows that if there is a feasible allocation sequence for S^* then its time reversed dual allocation sequence is feasible for S.

Computation time for banker's algorithm. Holt has shown that the banker's algorithm can be performed as follows with $rN \log N$ comparisons and rN additions and subtractions, where r is the number of resource types and N the number of requests ($=n$, the number of processes, if there are no partial requests): (1) Let the (termination) requests be denoted $R_i = (y_i, z_i)$, $i = 1, \cdots, N$. Form the $r \times N$ matrix $Y = (y_1, \cdots, y_N)$ where the costs y_i are the column vectors. Order each of the r rows of Y. (2) Repeatedly extract the minimum element from any row k, $1 \leqq k \leqq r$, of Y so long as the element $(y_i)_k$ being extracted is $\leqq (x)_k$. When $(y_i)_k$ is extracted from Y add 1 to a counter associated with the request R_i. When the counter of any R_i reaches r, this means that the banker is capable of allocating R_i. So add $(z_i - y_i)$ to x. (3) S is safe iff this procedure extracts all the elements of Y.

10. Proof of Theorem 4: generalization of banker's algorithm for linearly ordered requests.

Simplification of S. Let S be a system resource status with partial requests such that the requests of each process are linearly ordered. Consider a process P with requests $R_0, R_1, \cdots, R_m$ where $R_i = (y_i, z_i)$. If the banker makes the sequence $R_m, \cdots, R_1, R_0$ of allocations for successively longer initial segments of P's task, then the banker's stockpile will pass through a sequence of values

$$x,\ x + (z_m - y_m),\ \cdots,\ x + (z_1 - y_1),\ x + (z_0 - y_0) \tag{48}$$

which are linearly ordered resource vectors. That is, they are definitely larger or smaller than each other, so can be depicted by a graph as in Fig. 2. To define the

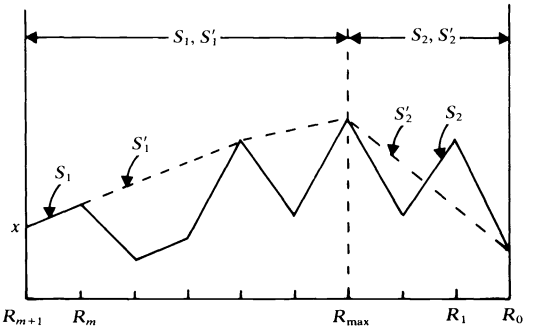


FIG. 2. *Relative size of successive banker's stockpiles when requests are linearly ordered.*

index *max* it is convenient to introduce an initial partial request which does nothing:

$$R_{m+1} = (0, 0). \tag{49}$$

Then *max* is defined by

$$(z_{\max} - y_{\max}) \geqq (z_i - y_i) \quad \text{for } i = m+1, m, \cdots, 0. \tag{50}$$

First S will be separated into S_1, S_2. Then S_1 will be simplified to S_1' and S_2 to S_2'.

S_1, S_2 are defined as follows: For each process P the requests up to and including $R_{\max}$ are put in S_1 and the requests after $R_{\max}$ in S_2:

$$R_{m+1}, R_m, \cdots, R_{\max} \in S_1, \tag{51}$$

$$R_{\max-1}, \cdots, R_1, R_0 \in S_2. \tag{52}$$

The banker's stockpile of S_1 is that of S:

$$x_1 = x. \tag{53}$$

The banker's stockpile of S_2 is the result of allocating all requests $R_{\max}$ in S:

$$x_2 = x + \sum_P (z_{\max} - y_{\max}), \tag{54}$$

where $\sum_P$ means sum over all processes of S.

Every process P of S is a process of S_1 with termination request $R_{\max}$. If $(z_i - y_i) < 0$ for $i = 0, \cdots, m$ then $R_{\max} = R_{m+1}$ and P has just the termination request R_{m+1} in S_1. If $R_{\max} = R_0$ then all the requests of P in S are included in S_1 and P does not occur in S_2.

S_1' is obtained from S_1 by deleting those partial requests which will become consumers after allocation of a previous partial request:

$$R_i \notin S_i' \quad \text{if } (z_k - y_k) \geqq (z_i - y_i) \text{ for some } k = m+1, m, \cdots, i+1. \tag{55}$$

So, the profitabilities of the requests of S_1' are isotonically increasing:

$$(z_{m'+1} - y_{m'+1}) < (z_{m'} - y_{m'}) < \cdots < (z_{\max} - y_{\max}). \tag{56}$$

This is depicted by the dashed line on the left side of Fig. 2.

S_2' is obtained from S_2 by deleting *all* partial requests. Thus, each process of S_2' has just a termination request which is a consumer. This is depicted by the dashed line on the right side of Fig. 2.

Motivation. It will be shown that

LEM1: S is safe $\Leftrightarrow$ S_1 is safe and S_2 is safe;

LEM2: S_1 is safe $\Leftrightarrow$ S_1' is safe;

LEM3: S_2 is safe $\Leftrightarrow$ S_2' is safe.

Furthermore, S_1' is safe iff the banker's algorithm yields a feasible allocation sequence. S_2' is safe iff the banker's algorithm yields a feasible allocation sequence for the time-reversed dual of S_2'.

It was shown in the previous section that the banker should allocate the producers first and the consumers last. This is the motication for LEM1. Actually, the banker should never allocate a consumer if he doesn't have to, and he is not required to allocate partial requests. This is the motivation for LEM2. I can't motivate LEM3, but its proof is trivial.

Generalized banker's algorithm. When S has been transformed into S_1', S_2' then the generalized banker's algorithm of the previous section becomes applicable to the more general problem of linearly ordered requests. Namely, S is safe iff it passes the following 3 tests:

(1) Use the banker's algorithm to determine if S_1' is safe. That is, choose a sequence of allocations for S_1' in any order which the banker is capable of making and see if all processes are terminated. The only adaption of the banker's algorithm

required by the fact that S_1' has partial requests is that when R_i is allocated all previous partial requests $R_m, \cdots, R_{i+1}$ must be deleted along with R_i.

(2) Determine what the banker's stockpile will be after allocating all requests of S:

$$x_f = x + \sum_P (z_0 - y_0). \tag{57}$$

For S to be safe it is necessary that

$$x_f \geqq 0. \tag{58}$$

(3) Construct the time-reversed dual S_2'' of S_2': The banker's stockpile of S_2'' is $x_2'' = x_f$ and the (termination) requests of S_2'' are those of S_2' with cost y and return z reversed. Use the banker's algorithm to determine if S_2'' is safe.

Proof of validity of generalized banker's algorithm. To prove LEM1, let A be a feasible allocation sequence for S_1 followed by a feasible allocation sequence for S_2. Then A is a feasible allocation sequence for S. Conversely, let A be any allocation sequence for S. A rule will be given for transforming A to A', and a second rule for transforming A' to A_1A_2, such that

(1) A_1 is an allocation sequence for S_1; A_2 is an allocation sequence for S_2;

(2) A feasible for $S \Rightarrow A'$ feasible for S;

(3) A' feasible for $S \Rightarrow A_1A_2$ feasible for S;

(4) A_1A_2 feasible for $S \Rightarrow A_1$ feasible for S_1 and A_2 feasible for S_2.

A is transformed to A' by inserting $R_{\max}$ in A for each process P for which $R_{\max}$ is not in A: Since R_0 must be in A, it follows that if $R_{\max}$ is not in A then A is of the form

$$A = (\cdots, R_i, \cdots), \tag{59}$$

where

$$\max > i, \tag{60}$$

and

$$R_{\max}, \cdots, R_{i+1} \text{ are not in } A. \tag{61}$$

Then insert $R_{\max}$ just before R_i:

$$A' = (\cdots, R_{\max}, R_i, \cdots). \tag{62}$$

A_1 is obtained from A' by deleting from A' all R_i such that $i < \max$. A_2 is obtained from A by deleting all R_i such that $i \geqq \max$.

Most of the above 4 properties claimed for A', A_1, A_2 are easily verified. I will show (3): A' feasible for $S \Rightarrow A_1A_2$ feasible for S.

Note that A_1A_2 is a permutation of A. Let

$$(A')_i = (\cdots, R_i), \tag{63}$$

$$(A_1A_2)_i = (\cdots, R_i) \tag{64}$$

be the prefixes of A' and A_1A_2 through allocation of R_i. I will show that the banker's stockpile after allocating $(A_1A_2)_i$ is $\geqq$ that after allocating $(A')_i$. Case 1: $i \geqq \max$, i.e. $(A_1A_2)_i$ is a prefix of A_1. Then $(A_1A_2)_i$ is obtained from $(A')_i$ by deleting allocations after $R'_{\max}$ of various processes P'. These allocations decrease the banker's stockpile when $(A')_i$ is allocated. Case 2: $i < \max$, i.e. $(A_1A_2)_i$ is obtained from $(A')_i$ by inserting the allocations through $R'_{\max}$ for some processes P'. These allocations increase the banker's stockpile when $(A_1A_2)_i$ is allocated.

To prove LEM2, let A be a feasible allocation sequence for S_1'. Then A is a feasible allocation sequence for S_1 since the requests of S_1 include those of S_1' and the banker's stockpile is the same in S_1 and S_1'. Conversely, let A be a feasible allocation sequence for S_1. Suppose some R_i is in A,

$$A = (\cdots, R_i, \cdots), \tag{65}$$

but the preceding requests

$$R_j, R_{j-1}, \cdots, R_{i+1} \quad \text{where } j > i \tag{66}$$

of this process are not in A. From AX3,

$$y_j \leqq y_{j-1} \leqq \cdots \leqq y_{i+1} \leqq y_i, \tag{67}$$

it follows that $R_j, \cdots, R_{i+1}$ can be inserted before R_i and the resulting allocation sequence

$$A' = (\cdots, R_j, R_{j-1}, \cdots, R_{i+1}, R_i, \cdots) \tag{68}$$

will be feasible. In this way A can be transformed into a feasible allocation sequence A' for S_1 which includes all partial requests, including those of S_1'. It only remains to show that if we delete from A' those partial requests which are not in S_1', then the resulting allocation sequence for S_1' was obtained from S_1 by deleting those partial requests which would decrease the banker's stockpile with respect to some previously attained value.

To prove LEM3, let A be a feasible allocation sequence for S_2'. Then A is a feasible allocation sequence for S_2. Conversely, let A be a feasible allocation sequence for S_2. Delete all allocations of partial requests from A to obtain an allocation sequence A' for S_2'. A' is feasible because the deleted allocations are requests after $R_{\max}$ in S, so reduce the banker's stockpile.

To see that the banker's algorithm can be used to determine if the time-reversed dual of S_2' is safe, note that S_2' has no partial requests. So this is a special case of the algorithm of the previous section.

To see that the banker's algorithm can be used to determine if S_1' is safe, note that all requests of S_1' are producers. So any allowed allocation can be chosen to be first, since it will only increase the banker's capability of allocating the other requests. Furthermore, for each process,the profitabilities of the successively larger partial requests are monotonically increasing. Therefore, after making the first allocation the new system resource status will retain the properties that all requests are producers and the profitabilities of the successively larger requests of each process are monotonically increasing. So the continued use of the banker's algorithm remains justified.

Problem asymmetry. S_1' was obtained from S_1 by deleting some of the partial requests. S_2' was obtained from S_2 by deleting all the partial requests. This asymmetry in the methods used to handle the partial requests of S before and after $R_{\max}$ is due to an asymmetry in the particular way I have chosen to allow the processes to make partial requests: P is allowed to state reduced needs in order for the computation to reach some intermediate point short of termination, but he is not allowed to say that his needs will be reduced after some intermediate point is passed. For example, in my formalism P can inform the banker that it needs $y_0 = (1, 1)$ to terminate and $y_1 = (1, 0)$ to reach an intermediate point, but P can't tell the banker that P will only need $(0, 1)$ after it has reached the intermediate point.

Acknowledgment. Many of the ideas presented here developed during discussions with Jean Vaucher. In particular, he suggested describing a process status by means of the quantities "give me" and "I will return," and considering the profitability to the banker of allocating a request.

REFERENCES

A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, chap. 10.

S. COOK (1970), *The complexity of theorem-proving procedures*, Conference Record of Third ACM Symposium of Theory of Computing, pp. 151–158.

A. N. HABERMANN (1969), *Prevention of system deadlocks*, Comm. ACM, 12, pp. 373–377, 385.

P. B. HANSEN (1973), *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ, pp. 42–49, 124–125.

R. C. HOLT (1972), *Some deadlock properties of computer systems*, ACM Computing Surveys, 4, pp. 179–196.

R. M. KARP (1975), *On the computational complexity of combinatorial problems*, Networks, 5, pp. 45–68.

SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0007 $01.00/0

APPROXIMATE REDUCTION AND LAMBDA CALCULUS MODELS*

CHRISTOPHER P. WADSWORTH†

Abstract. This paper gives the technical details and proofs for the notion of approximate reduction introduced in an earlier paper. The main theorem asserts that every lambda expression determines a set of approximate normal forms of which it is the limit in the lambda calculus models discovered by Scott in 1969. The proof of this theorem rests on the introduction of a notion of type assignments for the lambda calculus corresponding to the projections present in Scott's models; the proof is then achieved by a series of lemmas providing connections between the type-free lambda calculus and calculations with these type assignments.

As motivation for these semantic properties, we derive also some relations between the computational behavior of lambda expressions and their approximate normal forms, and we establish a syntactic analogue of the general considerations motivating the continuity of functions in Scott's lattice theoretic approach.

Key words. lambda calculus, Scott's models, approximations, approximate normal form, continuity, theory of computation

1. Introduction. In [12, § 5] we discussed a notion of approximate reduction as a tool for analyzing the λ-calculus models of Scott [9], [10], [11]. In this paper we develop the notion more formally and fill in the technical details and proofs omitted from the discussion in [12]; we assume the reader is familiar with [12] and with the general nature of Scott's lattice theoretic approach.

As in [12] our study concerns the relation between the syntactic and semantic aspects of the λ-calculus. The conventional notion of normal form is inadequate for such an investigation because, although we can take a normal form as a convenient *representation* of the result of a λ-calculus program when it has a normal form, we cannot consistently regard an expression as being "undefined" when it fails to have a normal form. The notions of approximate reduction and approximate normal form are introduced to overcome these limitations and show that the process of reduction is, in a limiting sense, complete for purposes of evaluation—every term determines, by reduction, a set of approximate normal forms of which it is the limit in Scott's $\mathbf{D}_\infty$-model. Informally, we can read this and the other results to be proved here as showing that the meaning of an expression in $\mathbf{D}_\infty$ captures exactly the input-output behavior determined by the possible computations involving the expression.

After a quick review of the $\mathbf{D}_\infty$-model in § 2, we define approximate reduction and related concepts in § 3 and develop their semantic properties. This leads to the main limit theorem (Theorem 3.5 below), the proof of which is the heart of the paper and will be given in § 4. Section 5 continues with several results relating the computational behavior of terms and their approximate normal forms which support our informal understanding of Theorem 3.5. Section 6 concludes the paper with a brief discussion of the related work of Hyland, Levy, and Welch.

2. Technical preliminaries. We consider a λ-calculus with one special constant, denoted by Ω, so the terms consist of all expressions fashioned out of variables and Ω by application and abstraction. Ω is used to stand for 'undetermined' parts of terms; an occurrence of Ω approximates a (sub-)term in the sense of giving no information about it, so terms containing Ω can be thought of as *partial* terms, or *approximate* terms.

* Received by the editors March 13, 1975, and in final revised form September 9, 1977.

† Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, Scotland. This work was supported by National Science Foundation under Grant GJ-41540 at the Department of Systems and Information Science, Syracuse University, Syracuse, New York and previously by a U.K. Science Research Council Grant to C. Strachey, Programming Research Group, Oxford University Computing Laboratory, Oxford, England.

We shall generally use lower case letters for variables and upper case letters for terms, and adopt the usual conventions (e.g. association to the left) regarding omission of parentheses. For the definitions of contexts, substitution, redexes, reduction and conversion (α-, β-, and η-), normal form, etc., and the concepts of solvability and head normal form, we refer the reader to [1], [3], [7], and [12].

Our semantics of this calculus is based on solutions of an isomorphism

$$\mathbf{D}_\infty \underset{\Psi}{\overset{\Phi}{\rightleftarrows}} [\mathbf{D}_\infty \to \mathbf{D}_\infty]$$

with $\mathbf{D}_\infty$ a complete lattice[1] and $[\mathbf{D}_\infty \to \mathbf{D}_\infty]$ the lattice of continuous functions from $\mathbf{D}_\infty$ to itself, where the functions Φ and Ψ forming the isomorphism are continuous. (Except for emphasis, hereafter the isomorphism functions Φ and Ψ will generally be omitted.) As usual, we use $\sqsubseteq$, $\bigsqcup$, and $\bot$ to denote, respectively, the partial ordering, the least upper bound operation, and the least element of complete lattices.

We consider Scott's solutions for $\mathbf{D}_\infty$, the construction of which provides an increasing sequence of projection[2] functions on $\mathbf{D}_\infty$ such that every element of $\mathbf{D}_\infty$ is the limit of its projections and satisfying a number of other useful laws. The following theorem summarizes, with subscripts denoting projections, the relevant facts we shall need in § 4; see [12, § 3] for a more detailed outline and informal discussion, and, e.g. [9] or [10] for a full account of Scott's construction with proofs.

THEOREM 2.1. *In Scott's solutions for* $\mathbf{D}_\infty$, *for each integer* $\geqq 0$ *there is a projection function on* $\mathbf{D}_\infty$ *such that the following hold*:

(P1) $x_n \sqsubseteq x_m \sqsubseteq x, \qquad 0 \leqq n \leqq m,$

(P2) $\bot_n = \bot,$

(P3) $x = \bigsqcup_{n=0}^{\infty} x_n,$

(P4) $(x_n)_m = x_{\min(n,m)},$

(P5) $(\bigsqcup X)_n = \bigsqcup \{x_n : x \in X\}, \quad \text{for } X \subseteq \mathbf{D}_\infty,$

(P6) $x(z) = y(z) \text{ for all } z \in \mathbf{D}_\infty \iff x = y,$

$x(z) \sqsubseteq y(z) \text{ for all } z \in \mathbf{D}_\infty \iff x \sqsubseteq y,$

(P7) $x(y) = \bigsqcup_{n=0}^{\infty} x_{n+1}(y_n)$

(P8) $\bot(y) = \bot,$

(P9) $x_0(y) = x_0 = x(\bot)_0,$

(P10) $x_{n+1}(y) = x(y_n)_n = x_{n+1}(y_n).$

[1] For those who prefer, complete partial orderings (partially ordered sets with a least element in which every directed subset has a least upper bound) may be used throughout without significantly affecting our development or results; we choose to work with complete lattices for ease of comparison to the referenced papers.

[2] To avoid ambiguity with other meanings in the literature, here a *projection* is a continuous function p from a complete lattice $\mathbf{D}$ to itself such that $p(x) \sqsubseteq x$ for all $x \in \mathbf{D}$.

Let **VAR**, **EXP**, and $\mathbf{ENV} = \mathbf{D}_\infty^{\mathbf{VAR}}$ denote the sets of all variables, terms, and environments, respectively, where $\mathbf{D}_\infty^{\mathbf{VAR}}$ is the set of all functions from **VAR** to $\mathbf{D}_\infty$, and make **ENV** a complete lattice by the "pointwise" partial ordering (i.e. $\rho \sqsubseteq_{\mathbf{ENV}} \rho'$ iff $\rho(x) \sqsubseteq_{\mathbf{D}_\infty} \rho'(x)$ for all $x \in \mathbf{VAR}$). As explained fully in [12, § 2], terms are interpreted in $\mathbf{D}_\infty$ by a mapping

$$\mathscr{V}: \mathbf{EXP} \to [\mathbf{ENV} \to \mathbf{D}_\infty]$$

defined, for $\rho \in \mathbf{ENV}$, by the clauses

(S1) $$\mathscr{V}[\![x]\!](\rho) = \rho[\![x]\!],$$

(S2) $$\mathscr{V}[\![MN]\!](\rho) = \Phi(\mathscr{V}[\![M]\!](\rho))(\mathscr{V}[\![N]\!](\rho)),$$

(S3) $$\mathscr{V}[\![\lambda x.M]\!](\rho) = \Psi(\lambda \mathbf{d} \in \mathbf{D}_\infty.\, \mathscr{V}[\![M]\!](\rho[\mathbf{d}/x])),$$

(S4) $$\mathscr{V}[\![\Omega]\!](\rho) = \bot,$$

where $\rho[\mathbf{d}/x] \equiv \rho' \in \mathbf{ENV}$ is given by

$$\rho'[\![x']\!] = \begin{cases} \mathbf{d}, & \text{if } x' \equiv x, \\ \rho[\![x']\!], & \text{if } x' \not\equiv x. \end{cases}$$

Then, corresponding to equality and the partial ordering in $\mathbf{D}_\infty$, $\mathscr{V}$ determines an equivalence relation and a quasiordering between terms, defined by

$$M =_{\mathbf{D}_\infty} N \quad \text{iff} \quad \mathscr{V}[\![M]\!](\rho) = \mathscr{V}[\![N]\!](\rho) \text{ for all } \rho \in \mathbf{ENV},$$

$$M \precsim N \quad \text{iff} \quad \mathscr{V}[\![M]\!](\rho) \sqsubseteq \mathscr{V}[\![N]\!](\rho) \text{ for all } \rho \in \mathbf{ENV}.$$

THEOREM 2.2. *The relation* $=_{\mathbf{D}_\infty}$ *provides a model for the* λ*-calculus system based on* α*-*β*-*η*-conversion*:

$$M \alpha\text{-}\beta\text{-}\eta\text{-}\mathbf{cnv}\ N \quad \textit{implies} \quad M =_{\mathbf{D}_\infty} N,$$

and the ordering $\precsim$ *has the following substitutivity property*:

$$M \precsim N \quad \textit{implies} \quad \mathbf{C}[M] \precsim \mathbf{C}[N] \textit{ for all contexts } \mathbf{C}[\ ].$$

Further, $\Omega \precsim X$ *for all terms* X*, and two* Ω*-conversion rules are valid in* D_∞:

$$(\Omega_1) \qquad \Omega X = \Omega,$$

$$(\Omega_2) \qquad \lambda x.\Omega = \Omega.$$

Proof. See [12]. □

For several results in §§ 4, 5, we need the notions of descendants and ancestors [7]. When terms are being reduced, these provide a means of associating parts of the later terms in a reduction sequence with parts of the initial term in the sequence (from which they are "descended"), and vice versa. We shall need the notions only for β-conversion, though there are similar definitions for α- and η-conversion.

First, suppose X is reduced to X' by contraction of a β-redex $R \equiv (\lambda x.M)N$ with contractum $R' \equiv [N/x]M$. We define a function *father* from parts of X' to parts of X such that every subterm S' of X' has a unique father in X, as follows:

1. If S' is not part of R', its father is the corresponding subterm of X.
2. If S' is all or part of an occurrence of N which was substituted for x in M, its father is the corresponding part of the occurrence of N in R.
3. If S' is any other part of R', its father is the corresponding part of the occurrence of M in R.

The relation *son* is the inverse of father. Every subterm of X thus has a unique son in X', except that neither the redex R being contracted nor its rator nor any of the free occurrences of x in M have any sons, while parts of N have k sons, where k is the number of free occurrences of x in M.

The relations *descendant* and *ancestor* are the natural extensions, by transitivity, of son and father to reduction sequences. It is easily seen that any descendant of an occurrence of Ω is itself an occurrence of Ω, that any descendant of a redex is itself a redex, and that after each step in a reduction sequence a redex fails to have a descendant iff either it is the redex being contracted or it is part of the rand N of the redex $(\lambda x.M)N$ being contracted and there are no free occurrences of x in M.

Redexes which are descendants of redexes are also called *residuals*, a notion heavily used in the classical proofs of the Church–Rosser Theorem. In particular, it leads to a definition of a restricted kind of reduction which always terminates (not in general in a normal form). Let $\mathscr{R}$ be a set of redexes in a term X. Then a reduction

$$X \equiv X_0 \xrightarrow{R_1} X_1 \xrightarrow{R_2} X_2 \longrightarrow \cdots \longrightarrow X_{n-1} \xrightarrow{R_n} X_n \equiv X' \tag{2.1}$$

is called a *reduction relative to* $\mathscr{R}$ if the redex R_i contracted in each step $X_{i-1} \to X_i$ is a residual of a redex in $\mathscr{R}$, and this reduction is *complete* if there are no residuals of $\mathscr{R}$ in the last term X'.

The following are well known:

LEMMA 2.3 (The Lemma of Parallel Moves). *If $\mathscr{R}$ is any set of β-redexes in a term X, there is a complete reduction of X relative to $\mathscr{R}$, and all complete reductions relative to $\mathscr{R}$ end in the same term X'.*

THEOREM 2.4 (The Church–Rosser Theorem). *If X* **cnv** *Y there is a term Z such that X* **red** *Z and Y* **red** *Z. Hence, if a term has two normal forms X and Y, then X α-***cnv** *Y.*

Not all methods of reduction are equally effective in reducing a term to normal form (if it has one). However, any reduction can be standardized in the following sense. If R and S are two β-redexes in a term X, we call R *senior* to S if the left-hand end of R lies to the left of the left-hand end of S in X. Then a reduction (2.1) is called a *standard reduction* iff for each $i = 1, 2, \cdots, n-1$, R_{i+1} is not a descendant of a redex in X_{i-1} senior to R_i; or, equivalently, iff R_{i+1} lies in or to the right of the contractum of R_i in X_i. (Note that this does not imply that R_{i+1} is not a descendant of a redex, only that if it is then no ancestor of R_{i+1} is senior to any redex contracted earlier in the reduction.)

A special case of standard reduction is *normal order reduction*, in which each step is determined by contraction of the *leftmost* redex, i.e. the (unique) redex senior to all other redexes in a term.

THEOREM 2.5 (The Standardization Theorem). *If X β-***red** *X' there is a standard reduction from X to X'. Hence, if X has a normal form, X can always be reduced to normal form by normal order reduction.*

Proofs of Lemma 2.3, Theorem 2.4, and Theorem 2.5 may be found in [1] or [3].

3. Approximate reduction. Intuitively, approximate reduction is intended as a means of determining (partial) information about the values of terms, by considering the form of expressions to which terms can be transformed by (β-)reduction. We restrict ourselves here to the formal definitions and supporting concepts and refer the reader to [12, § 5] for a general orientation.

DEFINITIONS. An Ω-*redex* is a subterm of the form ΩX or $\lambda x.\Omega$. A sequence of zero or more replacements of Ω-redexes by Ω is called an Ω-*reduction*. An Ω-*normal form* is a term which does not contain an Ω-redex.

A term A is said to Ω-*match* M if A and M are identical except at subterms which are occurrences of Ω in A, and to Ω-*approximate* M if A can be Ω-reduced to an Ω-match of M. Then, A is said to be a *direct approximant* of M if A Ω-approximates M and is in β-normal form, and is called the *best* direct approximant of M if A is the Ω-match obtained from M by replacing its (outermost) β-redexes by Ω. The set $\mathscr{A}(M)$ of *approximate normal forms* of M is defined by

$$\mathscr{A}(M) = \{A: \exists M'.M\beta\text{-}\mathbf{red}\ M' \text{ and } A \text{ is a direct approximant of } M'\}.$$

Example 1. Suppose

$$M \equiv \lambda x.x(R_1y)(yR_2)(x(\lambda z.R_3))$$

reduces to

$$M' \equiv \lambda x.x(x(\lambda w.R_4x)y)(yR_2)(x(\lambda z.z(R_5y)))$$

where R_1, R_2, R_3, R_4, R_5 are β-redexes, and let

$$A_1 \equiv \lambda x.x(\Omega)(y\Omega)(\Omega),$$

$$A_2 \equiv \lambda x.x(\Omega y)(y\Omega)(x(\lambda z.\Omega)),$$

$$A_3 \equiv \lambda x.x(x(\lambda w.\Omega x)y)(y\Omega)(x(\lambda z.z(\Omega y))),$$

$$A_4 \equiv \lambda x.x(x\Omega y)(y\Omega)(x(\lambda z.z\Omega)).$$

Then A_1 is a direct approximant of M (and of M'), A_2 and A_3 are the best direct approximants of M and M' respectively, and A_4 is another direct approximant of M' and is in Ω-normal form (as is A_1). All four (and any term Ω-reducible to any one of them) belong to both $\mathscr{A}(M)$ and $\mathscr{A}(M')$.

The definition of direct approximants just given is one of several alternatives. In particular, it differs slightly from that of [12, § 5] where terms Ω-reducible to Ω-matches were not considered. Since Ω-reduction preserves meanings (by Theorem 2.2), the wider definition adds no new meanings in $\mathbf{D}_\infty$. Its use here, at the suggestion of one of the referees, is convenient for precision in stating several auxiliary lemmas and their proofs; in particular, it enables equations like $\mathscr{A}(M) = \mathscr{A}(M')$ to mean exactly that, without the need for a qualification of "up to Ω-conversion" or something similar.

Note also that every term can be Ω-reduced to one which is in Ω-normal form (because Ω-reduction always decreases the length of terms). All the results to be proved remain true, exactly as stated, if the definition of direct approximants is restricted to allow only those in Ω-normal form. (One or two of the proofs, however, would be more cumbersome to express.)

The following two lemmas list some simple consequences of our definitions.

LEMMA 3.1. Ω-*approximation is transitive.*

Proof. Since Ω-reduction is transitive, it suffices to show that when A Ω-matches M and M reduces to M' by contraction of an Ω-redex W, there is an Ω-match A' of M' such that $A\Omega\text{-}\mathbf{red}\ A'$. If W is part of a subterm of M replaced by Ω in A, then A is an Ω-match of M' and there is nothing to prove; otherwise, the subterm of A corresponding to W in M is an Ω-redex, contraction of which yields the required A'. □

LEMMA 3.2. (a) *If A Ω-approximates M, then $A \sqsubseteq M$.*

(b) *$A \sqsubseteq M$ for all $A \in \mathscr{A}(M)$.*

Proof. Part (a) follows from the minimality of Ω and the substitutivity property for $\sqsubseteq$ mentioned in Theorem 2.2; then (b) follows using the validity of β-reduction. □

For the main result below (Theorem 3.5) it is definitely necessary to consider *all* approximate normal forms of a term, or at least a subset larger than one determined using any particular order of reduction. For instance, normal order reduction alone is not sufficient:

Example 2. Consider $M \equiv \lambda x.x(\Delta\Delta)(R)$, where $\Delta \equiv \lambda x.xx$ and R is any β-redex which does not have value $\perp$ in $\mathbf{D}_\infty$. Then $\Delta\Delta$ is the leftmost β-redex in M, so, since $\Delta\Delta$ reduces only to itself, the successive terms in the normal order reduction of M are all identical to M (i.e. the reduction never gets around to operating on the redex R). Hence, the successive best direct approximants in this reduction are all equal to $A \equiv \lambda x.x(\Omega)(\Omega)$, for which $A \sqsubseteq M$ but $A \neq_{\mathbf{D}_\infty} M$.

In other words, the successive best direct approximants in any particular reduction of a term M may converge to a limit smaller than the value of M; however, they do form an increasing sequence of approximations to M, and the set $\mathscr{A}(M)$ is always a directed set, as shown by the next two lemmas.

LEMMA 3.3 *Suppose M β-**red** M'. Then*:

(a) *If B and B' are the best direct approximants of M and M', respectively, then B Ω-approximates B'.*

(b) *Every direct approximant of M is a direct approximant of M'.*

(c) *M and M' have the same set of approximate normal forms.*

Proof. Let $R_1, R_2, \cdots, R_n$ be the outermost β-redexes in M.

For (a), by transitivity it is sufficient to consider the case where M is deduced to M' by contraction of a single β-redex R.

If R is not one of the outermost redexes, then R must be internal to one of them and disjoint from the rest. M and M' are then identical except at some subterm internal to one of their outermost redexes, so $B \equiv B'$ in this case.

Now suppose R is one of the outermost redexes in M, say $R = R_1$ for definiteness. Let X be the contractum of R, and let $\mathbf{C}[\ ]$ be the context of R and X in M and M', respectively, i.e. the context such that $M \equiv \mathbf{C}[R]$ and $M' \equiv \mathbf{C}[X]$. The redex R is now disjoint from the other outermost redexes in M, so, for $i = 2, 3, \cdots, n$, R_i has a unique residual in M' identical to R_i and this occurs as part of the context $\mathbf{C}[\ ]$. Letting $\mathbf{C}'[\ ]$ denote the result of replacing $R_2, R_3, \cdots, R_n$ in $\mathbf{C}[\ ]$ by Ω, then $B \equiv \mathbf{C}'[\Omega]$. For B', there are two cases, depending on the structure of the contractum X of R:

Case 1. *X is a variable or a combination.* Then X cannot be contained in an outermost redex Q of M' (else Q would be the residual of a redex in M containing R, contradicting R being an outermost redex of M). Thus, the outermost redexes of M' consist of $R_2, R_3, \cdots, R_n$ and the outermost redexes of X. So if B_X denotes the best direct approximant of X, then $B' \equiv \mathbf{C}'[B_X]$, which is Ω-matched by $\mathbf{C}'[\Omega] \equiv B$.

Case 2. *X is an abstraction.* If X is not the rator of a combination in M', then $B' \equiv \mathbf{C}'[B_X]$ follows as in Case 1. So suppose X is the rator of a combination XZ in M'. Then XZ is the unique descendant of a combination RZ in M and is an outermost redex in M' (else R could not have been an outermost redex in M). Some of the R_i, say $R_2, R_3, \cdots, R_k$ $(1 \leqq k \leqq n)$, may occur as parts of the rand Z of XZ in M', and are therefore not outermost redexes in M'. Thus, the outermost redexes of M' consist of $R_{k+1}, R_{k+2}, \cdots, R_n$ and XZ.

Let $\mathbf{C}_0[\ ]$ be the context such that $M \equiv \mathbf{C}_0[RZ]$ and $M' \equiv \mathbf{C}_0[XZ]$. Let B_Z be the best direct approximant of Z (i.e. the result of replacing $R_2, R_3, \cdots, R_k$ in Z by Ω) and let $\mathbf{C}_0'[\ ]$ be the result of replacing $R_{k+1}, R_{k+2}, \cdots, R_n$ in $\mathbf{C}_0[\ ]$ by Ω. Then $B \equiv \mathbf{C}_0'[\Omega B_Z]$, which is Ω-reducible to $\mathbf{C}_0'[\Omega] \equiv B'$.

This completes the proof of (a). Part (b) now follows using the transitivity of Ω-approximation from the observation that every direct approximant of a term Ω-approximates its best direct approximant.

For (c), $\mathscr{A}(M') \subseteq \mathscr{A}(M)$ is immediate, for every direct approximant of a term to which M' is reducible is, since $M\ \beta\textbf{-red}\ M'$, clearly also a direct approximant of a term to which M is reducible. For the reverse inclusion, suppose $A \in \mathscr{A}(M)$ is a direct approximant of a term M'' to which M is reducible. Then, by the Church–Rosser Theorem, there is a term M''' such that $M'\ \beta\textbf{-red}\ M'''$ and $M''\ \beta\textbf{-red}\ M'''$. The latter implies, by (b), that A is a direct approximant of M''' and hence, since $M'\ \beta\textbf{-red}\ M'''$, is an approximate normal form of M'. □

LEMMA 3.4. *For every term M, the set $\mathscr{A}(M)$ is directed with respect to the quasiordering $\sqsubseteq$.*

Proof. The proof is straightforward and uses the Church–Rosser Theorem and Lemmas 3.2 and 3.3 by an argument similar to that just given for Lemma 3.3(c). □

Now we can state the main theorem that, passing to values in $\mathbf{D}_\infty$, every term is the limit of its approximate normal forms:

THEOREM 3.5. *For all terms M and environments ρ,*

$$\mathscr{V}[\![M]\!](\rho) = \bigsqcup\{\mathscr{V}[\![A]\!](\rho) : A \in \mathscr{A}(M)\}.$$

COROLLARY 3.6. *For all terms M, $\mathscr{V}[\![M]\!](\rho) = \bot$ for all environments ρ iff M is unsolvable iff M does not have a head normal form.*

Proof. See [12, Cor. 5.3]. □

Of course, these last two results essentially show that every term has, in a certain sense, an *infinite* normal form. This is pursued further by Nakajima [8].

4. Proof of Theorem 3.5.

$$\textit{To show:}\quad \mathscr{V}[\![M]\!](\rho) = \bigsqcup\{\mathscr{V}[\![A]\!](\rho) : A \in \mathscr{A}(M)\}$$

That the r.h.s. $\sqsubseteq$ l.h.s. we know already, since $A \in \mathscr{A}(M)$ implies $A \sqsubseteq M$. For the reverse ordering, we proceed via a notion of *type assignment* for the λ-calculus appropriate to these models, suggested to us by J. M. E. Hyland in early 1972. Our "types" will be integers and a type assignment will consist of an association of (arbitrary) integers with *every* subterm of a term; the intended interpretation is that the corresponding projection of the value of the subterm is to be taken. A typed analogue of β-reduction will then be defined and shown to preserve the values of terms with type assignments. Then, as with all typed λ-calculi, every typed term T will be reducible to, and hence equivalent to, a typed term T' in normal form. The connection with untyped approximate normal forms is then achieved by observing that the untyped version of T' is one of the approximate normal forms of the untyped term corresponding to T.

We break the proof into five major steps as follows:

I. Represent type assignments formally by writing the integers associated with subterms as superscripts on the subterms. (We choose superscripts rather than subscripts to keep the distinction clear as to when we are talking of the (typed) *terms* and when we are talking of the projections, denoted by subscripts, of their *values*.) Then,

for a typed term T, we shall write T^* for the corresponding untyped term obtained by deleting the superscripts, and, for an untyped term M, we shall write $\tau(M) = \{T: T^* = M\}$ for the set of typed terms representing type assignments for M.

II. Define the interpretation of typed terms by a mapping

$$\mathscr{T}: \{\text{typed terms}\} \to [\mathbf{ENV} \to \mathbf{D}_\infty]$$

(defined as for $\mathscr{V}$, with the addition that superscripts are interpreted as the corresponding projection) and show that, for any untyped term M, the join of the values $\mathscr{T}[\![T]\!](\rho)$ taken over all $T \in \tau(M)$ is equal to $\mathscr{V}[\![M]\!](\rho)$.

III. Define a concept of typed β-reduction which preserves values under $\mathscr{T}$. Note that a β-redex $(\lambda x.M)N$ with a type assignment has the form

$$Q \equiv ((\lambda x.T)^{(i)}W)^{(j)}, \qquad T \in \tau(M), \quad W \in \tau(N), \quad i, j \geqq 0;$$

then, roughly, corresponding to the properties (P9) and (P10) of projections, we shall define the contractum Q' of Q as

(a) if $i = 0$, then $Q' \equiv [\Omega^{(0)}/x]T^{(0)}$;

(b) if $i = n+1 > 0$, then $Q' \equiv ([W^{(n)}/x]T^{(n)})^{(j)}$.

IV. Show that typed reductions can always be chosen so as to reduce any typed term T to a T' in normal form (by successively contracting the rightmost redex of maximal degree, where the *degree* of a redex is the integer, i in III, assigned to its rator).

V. Show that $(T')^*$ is a member of $\mathscr{A}(M)$, where T' is obtained from $T \in \tau(M)$ as in IV (by extending the notion of Ω-matching to typed terms and determining an ordinary reduction of M Ω-matched by the typed reduction from T to T').

With I–V achieved, we then have, for all $T \in \tau(M)$,

$$\begin{aligned}
\mathscr{T}[\![T]\!](\rho) &= \mathscr{T}[\![T']\!](\rho) && \text{for } T' \text{ determined by IV} \\
&\sqsubseteq \mathscr{V}[\![(T')^*]\!](\rho) && \text{since, compared to } \mathscr{T}, \\
& && \mathscr{V} \text{ does not apply projections} \\
&\sqsubseteq \bigsqcup\{\mathscr{V}[\![A]\!](\rho): A \in \mathscr{A}(M)\} && \text{since } (T')^* \in \mathscr{A}(M) \text{ by } \mathscr{V}
\end{aligned}$$

which will, by II, complete the proof of Theorem 3.5. The rest of this section formalizes each of the steps I to V.

Step I. *Syntax of typed terms.* Formally, we define the set of *typed terms* inductively as follows:

1. For $n \geqq 0$, $x^{(n)}$ and $\Omega^{(n)}$ are typed terms.
2. If T and W are typed terms, so is $(TW)^{(n)}$ for every $n \geqq 0$.
3. If T is a typed term and x is a variable, then $(\lambda x.T)^{(n)}$ is a typed term for every $n \geqq 0$.
4. If T is a typed term, so is $T^{(n)}$ for every $n \geqq 0$.

Compared to our informal discussion above, the last clause has been added here for the technical convenience of allowing multiple superscripts.

In 1 to 4, the superscript n on the whole term will be called the *type* of the term. Although we call our system a "typed" λ-calculus, notice that there is no type restriction on the formation of function applications (unlike other typed calculi). If we wished, we could impose such a restriction—e.g., based on the construction of $\mathbf{D}_\infty$, that $(TW)^{(n)}$ be admitted as being well-formed only when T and W are of type $n+1$ and n, respectively—but this would only complicate matters and is unnecessary here, since the isomorphism between $\mathbf{D}_\infty$ and $[\mathbf{D}_\infty \to \mathbf{D}_\infty]$ allows every term to be meaningfully applied to terms of any other or the same type.

The notation T^* is given by

$$(x^{(n)})^* \equiv x,$$
$$(\Omega^{(n)})^* \equiv \Omega,$$
$$((TW)^{(n)})^* \equiv T^* W^*,$$
$$((\lambda x.T)^{(n)})^* \equiv \lambda x.(T^*),$$
$$(T^{(n)})^* \equiv T^*,$$

and the sets $\tau(M)$ are given explicitly by

$$\tau(x) = \{x^{(n)}: n \geqq 0\}$$
$$\tau(MN) = \{(TW)^{(n)}: T \in \tau(M), W \in \tau(N), n \geqq 0\},$$
$$\tau(\lambda x.M) = \{(\lambda x.T)^{(n)}: T \in \tau(M), n \geqq 0\},$$
$$\tau(\Omega) = \{\Omega^{(n)}: n \geqq 0\}.$$

Step II. *Interpretation of typed-terms.* We define the semantic function $\mathscr{T}$ by

(T1) $$\mathscr{T}[\![x^{(n)}]\!](\rho) = (\rho[\![x]\!])_n,$$

(T2) $$\mathscr{T}[\![(TW)^{(n)}]\!](\rho) = (\Phi(\mathscr{T}[\![T]\!](\rho))(\mathscr{T}[\![W]\!](\rho)))_n,$$

(T3) $$\mathscr{T}[\![(\lambda x.T)^{(n)}]\!](\rho) = (\Psi(\lambda \mathbf{d} \in \mathbf{D}_\infty.\mathscr{T}[\![T]\!](\rho[\mathbf{d}/x])))_n,$$

(T4) $$\mathscr{T}[\![\Omega^{(n)}]\!](\rho) = \bot,$$

(T5) $$\mathscr{T}[\![T^{(n)}]\!](\rho) = (\mathscr{T}[\![T]\!](\rho))_n.$$

COROLLARY 4.1. *For all typed terms T, if n is the type of T, then*

(a) $$\mathscr{T}[\![T]\!](\rho) \sqsubseteq (\mathscr{V}[\![T^*]\!](\rho))_n \sqsubseteq \mathscr{V}[\![T^*]\!](\rho)$$

(b) $$\mathscr{T}[\![(T^{(m)})^{(k)}]\!](\rho) = \mathscr{T}[\![T^{(\min(m,k))}]\!](\rho), \quad \textit{for all } m, k \geqq 0$$

(c) *for all ordinary terms M and environments ρ, the set*

$$\{\mathscr{T}[\![T]\!](\rho): T \in \tau(M)\}$$

is directed, and

$$\mathscr{V}[\![M]\!](\rho) = \bigsqcup\{\mathscr{T}[\![T]\!](\rho): T \in \tau(M)\}.$$

Proof. All parts are straightforward from the definitions except the second part of (c), which follows by structural induction on M, using continuity and the property (P3) of projections. □

Note that Corollary 4.1(b) shows that multiple superscripts provide a "re-typing" operation in the syntax corresponding to the composition law (P4) for projections. We assume hereafter that such simplifications—replacing multiple superscripts by their minimum—are applied to typed terms automatically without explicit mention.

Step III. *Typed reduction.* As for ordinary β-reduction, we first define a substitution operation. The definition is just what one would expect by saying that we do the substitution as usual while carrying the type assignments along—if $T \in \tau(M)$ and $W \in \tau(N)$, then the *typed substitution* $[W/x]T \equiv T' \in \tau(M')$ denotes the ordinary substitution $[N/x]M \equiv M'$ with the type assignment T' for M' as follows:

1. *Proper* subterms of occurrences of N substituted for x in M are assigned the same type as under the type assignment W for N.

2. Occurrences of N itself are assigned the smaller of the type of W and the type assigned by T to the occurrence of x in M replaced by the occurrence of N.
3. All other subterms of M' are assigned the same type as the corresponding subterm of M under the type assignment T for M.

LEMMA 4.2 (The typed substitution lemma). *For all typed terms T, W, variables x, and environments ρ,*

$$\mathscr{T}[\![[W/x]T]\!](\rho) = \mathscr{T}[\![T]\!](\rho[\mathscr{T}[\![W]\!](\rho)/x]).$$

Proof. The proof is a straightforward, but tedious, structural induction on T, using Corollary 4.1(b) and the composition law for projections. □

A typed term will be called a *typed redex* if it is of the form

$$Q \equiv ((\lambda x.T)^{(i)}W)^{(j)}, \qquad i, j \geqq 0.$$

The *typed contractum* Q' of Q is then defined as given in the statement of Step III above, with typed substitution read as just given.

LEMMA 4.3 (Validity of typed reduction). *For any typed redex Q with contractum Q', for all environments ρ,*

$$\mathscr{T}[\![Q]\!](\rho) = \mathscr{T}[\![Q']\!](\rho).$$

Proof. Let Q, Q' be as given above, with $i \geqq 0$ the degree of Q. There are two cases to consider: (a) $i = 0$, and (b) $i = n+1 > 0$.

(a)
$$\begin{aligned}
\mathscr{T}[\![Q]\!](\rho) &\equiv \mathscr{T}[\![((\lambda x.T)^{(0)}W)^{(j)}]\!](\rho) \\
&= ((\lambda \mathbf{d} \in \mathbf{D}_\infty.\mathscr{T}[\![T]\!](\rho[\mathbf{d}/x]))_0(\mathscr{T}[\![W]\!](\rho)))_j && \text{by (T2), (T3) (with applications of } \Phi, \Psi \text{ omitted)} \\
&= ((\lambda \mathbf{d} \in \mathbf{D}_\infty.\mathscr{T}[\![T]\!](\rho[\mathbf{d}/x]))(\bot))_0 && \text{by (P9) and (P4)} \\
&= (\mathscr{T}[\![T]\!](\rho[\bot/x]))_0 && \text{by def. of function application} \\
&= \mathscr{T}[\![T^{(0)}]\!](\rho[\mathscr{T}[\![\Omega^{(0)}]\!](\rho)/x]) && \text{by (T4) and (T5)} \\
&= \mathscr{T}[\![[\Omega^{(0)}/x]T^{(0)}]\!](\rho) \equiv \mathscr{T}[\![Q']\!](\rho) && \text{by Lemma 4.2.}
\end{aligned}$$

(b)
$$\begin{aligned}
\mathscr{T}[\![Q]\!](\rho) &\equiv \mathscr{T}[\![((\lambda x.T)^{(n+1)}W)^{(j)}]\!](\rho) \\
&= ((\lambda \mathbf{d} \in \mathbf{D}_\infty.\mathscr{T}[\![T]\!](\rho[\mathbf{d}/x]))_{n+1}(\mathscr{T}[\![W]\!](\rho)))_j && \text{as in case (a)} \\
&= ((\mathscr{T}[\![T]\!](\rho[(\mathscr{T}[\![W]\!](\rho))_n/x]))_n)_j && \text{using (P10)} \\
&= (\mathscr{T}[\![T^{(n)}]\!](\rho[\mathscr{T}[\![W^{(n)}]\!](\rho)/x]))_j && \text{by (T5) twice} \\
&= (\mathscr{T}[\![[W^{(n)}/x]T^{(n)}]\!](\rho))_j && \text{by Lemma 4.2} \\
&= \mathscr{T}[\![([W^{(n)}/x]T^{(n)})^{(j)}]\!](\rho) \equiv \mathscr{T}[\![Q']\!](\rho) && \text{by (T5)} \quad \square
\end{aligned}$$

Step IV. *Typed terms always have normal forms.* The proof here is a standard one for typed λ-calculi; compare, e.g., [7, p. 107]. First, a term will be said to be of *maximal degree n* if it contains a redex of degree n but none of higher degree.

LEMMA 4.4 (The normalization theorem).

(a) *Every typed term of maximal degree $n+1$ can be reduced to one of maximal degree $\leqq n$.*

(b) *Every typed term of maximal degree 0 can be reduced to one in β-normal form.*

Hence, by induction on the maximal degree,

(c) *every typed term can be reduced to one in β-normal form.*

Proof. For (a), suppose X is of maximal degree $n+1$, and let

$$Q \equiv ((\lambda x.T)^{(i)}W)^{(j)}$$

be a redex in X of degree $n+1$ whose operand W does not contain a redex of degree $n+1$ (e.g. choose the rightmost redex of degree $n+1$). We show that the result X' of contracting Q in X contains one less redex of degree $n+1$ and no redex of higher degree. Part (a) then follows by induction on the number of redexes of degree $n+1$ in X.

Let P' be a redex in X'. Then P' must be the son of a combination P in X whose rator must be either (i) an abstraction, (ii) a variable, or (iii) a combination.

In case (i), P itself is a redex in X, so, by definition of the type assignment for X' in terms of that for X, the degree of P' is equal to that of P. Now, if P had degree $n+1$, then P could not be part of W (by choice of Q), so P' is the unique son of P in X'. Since the redex contracted is of degree $n+1$ and has no sons in X', this implies there is one less redex of degree $n+1$ (and no redex of higher degree) in X' whose father was a redex in X.

In case (ii), an abstraction must have replaced a variable in the combination P. Since the free occurrences of x in T are the only variables replaced when Q is contracted, the rator of P' must be an occurrence of W which has been substituted for x. But, by definition, such subterms of X' are assigned a type $\leqq$ one less than the degree of the redex being contracted, so P' has degree $\leqq n$ in this case.

In case (iii), an abstraction has replaced a combination. Since the redex being contracted is the only combination which can be so modified by a contraction, the rator of P' must be the contractum of Q, which, by definition, is assigned a type $\leqq n$. Hence again P' is of degree $\leqq n$ for this case.

Thus, redexes in X' have degree $n+1$ iff their father in X was a redex of degree $n+1$, and there is one less redex of degree $n+1$ in X' and no redex of higher degree.

For (b), now suppose X is of maximal degree 0, and let Q be *any* redex as above of degree $i=0$ in X. Clearly, the result X' of replacing Q in X by its contractum $[\Omega^{(0)}/x]T^{(0)}$ is of shorter length than X, so it suffices to show that X' is of maximal degree 0, for then (b) follows by induction on the length of X.

Let P' be any redex in X' and consider cases as above. In case (i), as before P' and its father in X have the same degree, which must be 0 as X is of maximal degree 0. Case (ii) cannot arise this time since occurrences of x in T are replaced by Ω. In case (iii), the rator of P' must be the contractum of Q as before, so P' has degree 0 by definition of the type assigned to the contractum of a redex of degree 0. □

Step V. *Connecting typed and approximate reductions.* Call a typed term T an *Ω-match* of an untyped term M if T^* is an Ω-match of M. (Note this includes the special case where $T \in \tau(M)$, for then $T^* \equiv M$.)

LEMMA 4.5. *Suppose T Ω-matches M and T reduces to T' by contraction of a typed redex Q. Then there is a β-redex R in M such that T' Ω-matches the result M' of contracting R in M.*

Proof. Since T is an Ω-match of M, the subterm of M corresponding to Q in T must be a β-redex. Choosing R as this redex, the result follows easily from the definitions by a case analysis on the degree of Q. □

COROLLARY 4.6. *For every $T \in \tau(M)$, there is a typed reduction of T to a typed term T' such that $(T')^* \in \mathscr{A}(M)$.*

Proof. Let T' be any typed term in β-normal form for which there is, by Lemma 4.4, a typed reduction from T to T'. If we apply Lemma 4.5 to each step in this typed reduction, there is an untyped term M' such that $M\ \beta$-**red** M' and T' Ω-matches M'. Since T', and hence also $(T')^*$, is in β-normal form, this implies that $(T')^*$ is a direct approximant of M', whence $(T')^* \in \mathscr{A}(M)$. □

5. Computational properties of approximate normal forms. In this section we establish several results which provide technical support for expecting properties such as Theorem 3.5 to hold in any "reasonable" model of the λ-calculus. Specifically, we shall show that a knowledge of the approximate normal forms of a term is sufficient to determine its overall computational behavior (discussed in [12] without the proofs).

THEOREM 5.1. *For all terms M and contexts $\mathbf{C}[\ ]$, $\mathbf{C}[M]$ has an Ω-free normal form (a head normal form) iff $\mathbf{C}[A]$ has the same Ω-free normal form (a similar head normal form) for some $A \in \mathscr{A}(M)$.*

Theorem 5.1 can be proved directly (by specializing the argument below to normal order reductions and head reductions), but we shall derive it here as a corollary of a more general result of some interest in itself:

THEOREM 5.2. *For all terms M, β-normal forms A', and contexts $\mathbf{C}[\ ]$, $A' \in \mathscr{A}(\mathbf{C}[M])$ iff there is an $A \in \mathscr{A}(M)$ such that $A' \in \mathscr{A}(\mathbf{C}[A])$.*

Equivalently, Theorem 5.2 can be stated as

$$\mathscr{A}(\mathbf{C}[M]) = \bigcup\{\mathscr{A}(\mathbf{C}[A]) : A \in \mathscr{A}(M)\}$$

which gives an interesting syntactic analogue of the reasoning underlying the continuity of functions in Scott's work. In words, for the special case $\mathbf{C}[M] \equiv FM$ of a function application, it says: to obtain an approximate normal form of FM requires only an approximate normal form of the argument M.

Theorem 5.1 is immediate from Theorem 5.2 and the observations that (a) if a term has an Ω-free normal form N, then N is always one of its approximate normal forms, and (b) if a term has a head normal form $\lambda x_1 x_2 \cdots x_n.zX_1X_2 \cdots X_m$, where $m, n \geqq 0$ and z is a variable, then $\lambda x_1 x_2 \cdots x_n.z\Omega\Omega \cdots \Omega$ is one of its approximate normal forms.

For the proof of Theorem 5.2, we need first a series of lemmas comparing the reductions of terms and their Ω-approximates, to enable us to substitute arbitrary terms, in particular Ω, for subterms inessential to a reduction. To facilitate the treatment of several results, we introduce the notation

$$X \equiv X_0 \xrightarrow{R_1} X_1 \xrightarrow{R_2} X_2 \longrightarrow \cdots \longrightarrow X_{n-1} \xrightarrow{R_n} X_n \equiv X' \tag{5.1}$$

for a β-reduction of length $n \geqq 0$ from X to X', where R_i is the redex contracted in t e ith step from X_{i-1} to X_i, for $i = 1, 2, \cdots, n$.

LEMMA 5.3. *Suppose $\mathscr{R}$ is a set of β-redexes in a term X and $X\ \beta$-**red** X' without contraction of a residual of a redex in $\mathscr{R}$. Let Z be the Ω-match of X obtained by replacing the outermost members of $\mathscr{R}$ by Ω. Then there is an Ω-match Z' of X', obtainable by replacing a set of β-redexes in X' by Ω, such that $Z\ \beta$-**red** Z'.*

Proof. Let $\mathscr{R}'$ be the set of residuals in X' of members of $\mathscr{R}$, and let Z' be the Ω-match of X' obtained by replacing the outermost members of $\mathscr{R}'$ by Ω. Let (5.1) be the given reduction from X to X' of length n. By transitivity it suffices to consider the case $n = 1$, for which, since R_1 is not a member of $\mathscr{R}$ (by the condition on the given reduction from X to X'), either R_1 is part of a member of $\mathscr{R}$ or there is a β-redex S in Z corresponding to R_1 in X_0. In the former case, $Z' \equiv Z$ and there is nothing to prove; in the later case, it is easily seen that Z' is the result of contracting the redex S in Z. □

The next three lemmas are directed toward showing that taking Ω-approximates can be postponed across a β-reduction. Recall that Ω-approximation was defined by combining Ω-matching and Ω-reduction, and includes these two relations as particular cases (by reflexivity). Postponement of taking Ω-matches is easily shown:

LEMMA 5.4. *If Z Ω-matches X and Z β-**red** Z', there is a term X' such that Z' Ω-matches X' and X β-**red** X'.*

Proof. By transitivity, we may assume that Z' results from Z by contraction of a single β-redex S. Since Z Ω-matches X, the subterm of X corresponding to S in Z must be a β-redex, contraction of which yields the required X'. □

For Ω-reduction, more care is required as, somewhat surprisingly perhaps, it is not always true that when Z Ω-**red** X and Z β-**red** Z' there is a term X' such that Z' Ω-**red** X' and X β-**red** X'. (A counterexample would be given by taking $Z \equiv (\lambda x.\Omega)N$, $Z' \equiv \Omega$, and $X \equiv \Omega N$, where N is any term.) In general, one can only conclude the existence of a term X' such that Z' Ω-approximates X' (and X β-**red** X'). We show first:

LEMMA 5.5. *Suppose Z Ω-**red** X and Z β-**red** Z' by contraction of a single β-redex S in Z. Then either Z' Ω-approximates X or there is a term X' such that Z' Ω-**red** X' and X β-**red** X' by contraction of a single β-redex in X.*

Proof. We argue by induction on the length of the Ω-reduction. If $Z \equiv X$, the result is trivially true (by taking $X' \equiv X$), so suppose Z reduces to Y by contracting an Ω-redex W and Y Ω-**red** X. Let $S \equiv (\lambda x.U)V$ and let $\mathbf{C}[\ ]$ be the context of S in Z, i.e. $Z \equiv \mathbf{C}[S]$. Two cases arise, according as W is the rator of S or not.

If W is the rator $\lambda x.U$ of S, then $U \equiv \Omega$ else W would not be an Ω-redex. Then $Z' \equiv \mathbf{C}[\Omega]$ Ω-matches $\mathbf{C}[\Omega V] \equiv Y$ Ω-**red** X, whence Z' Ω-approximates X by transitivity of Ω-approximation (since Ω-matching and Ω-reduction are special cases).

If W is not the rator of S, the subterm of Y corresponding to S in Z is a β-redex R and the descendants of W in Z' are Ω-redexes $W_1, W_2, \cdots, W_k$ (where $k = 1$ unless W is part of V, in which case k is the number of occurrences of x in U). Let Y' be the term obtained by contracting R in Y, then note that Z' Ω-**red** Y' by contracting $W_1, W_2, \cdots, W_k$ in any order. The result now follows by applying the induction hypothesis to Y. □

LEMMA 5.6. *Suppose Z β-**red** Z' and Z Ω-approximates X. Then X is β-reducible to a term X' which is Ω-approximated by Z'.* (See Fig. 1, where horizontal arrows denote β-reductions and vertical dashed lines denote that the lower term Ω-approximates the upper term.)

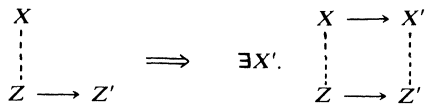


FIG. 1

Hence, if Z has an Ω-free normal form (a head normal form) then any term Ω-approximated by Z has the same Ω-free normal form (a similar head normal form).

Proof. For the first part, by transitivity it suffices to consider the case that Z β-**red** Z' by contraction of a single β-redex, for which case the result is straightforward from the definition of Ω-approximation and Lemmas 5.4 and 5.5. The second part then follows from the observations that an Ω-free normal form Ω-approximates only itself and any term Ω-approximated by a head normal form is a similar head normal form. □

COROLLARY 5.7 (Postponement of taking Ω-approximates). *If X can be transformed to Z' by a sequence of operations each of which is either a β-reduction or the replacement of subterms by an Ω-approximate, there is a term X' such that X β-**red** X' and Z' Ω-approximates X'.*

Proof. Since both β-reduction and Ω-approximation are reflexive and transitive relations, we can assume without loss of generality that the transformations from X to Z' consist of an alternating sequence of Ω-approximates and β-reductions, beginning with an Ω-approximate and ending with a β-reduction, as in Fig. 2(a). If we apply Lemma 5.6 inductively, for $i = 0, 1, \cdots, n-1$, with $X \equiv X_i$, $Z \equiv Z_{i+1}$, and $Z' \equiv Z'_{i+1}$, there are terms X_{i+1} as shown in Fig. 2(b), from which the result follows by setting $X' \equiv X_n$. □

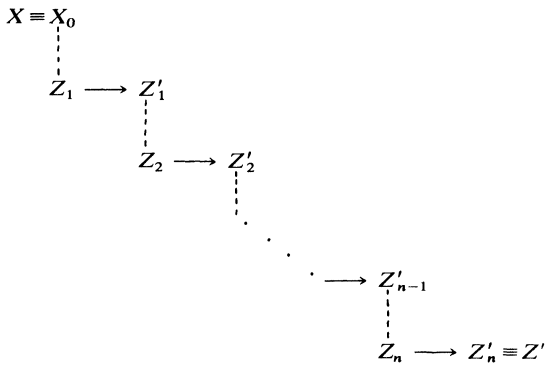


FIG. 2(a)

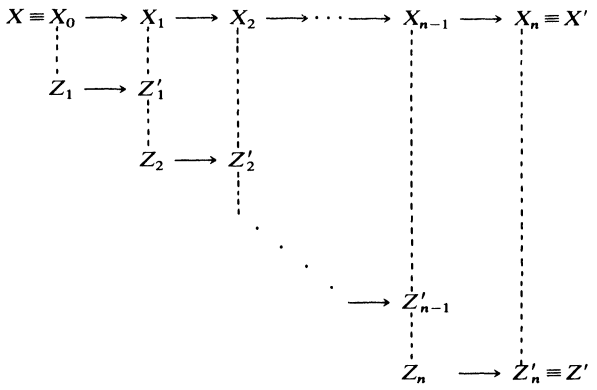
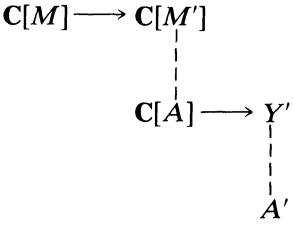


FIG. 2(b)

COROLLARY 5.8 ('if' part of Theorem 5.2). *If $A \in \mathscr{A}(M)$ and $A' \in \mathscr{A}(\mathbf{C}[A])$, then $A' \in \mathscr{A}(\mathbf{C}[M])$.*

Proof. By definition of $\mathscr{A}(M)$ and $\mathscr{A}(\mathbf{C}[A])$, when $A \in \mathscr{A}(M)$ and $A' \in \mathscr{A}(\mathbf{C}[A])$ there exist terms M' and Y' such that



from which the result follows by Corollary 5.7 (with $X \equiv \mathbf{C}[M]$ and $Z' \equiv A'$). □

For the "only if" part of Theorem 5.2, we show first that a certain restricted kind of outermost reduction is sufficient to generate all approximate normal forms of a term (much as normal order reduction suffices to reduce a term to normal form, when one exists). Then we shall take advantage of the restricted form of these reductions to set up an induction on their length and hence determine an approximate normal form $A \in \mathscr{A}(M)$ sufficient to generate the given $A' \in \mathscr{A}(\mathbf{C}[M])$.

DEFINITIONS. By the *reach* of a redex R in a term M we shall mean the largest subterm of M of the form $RX_1X_2 \cdots X_n$ $(n \geqq 0)$. A redex will be called a *stable outermost* (s.o. for short) *redex* just when it is not within the reach of any other redex. (Equivalently, a redex is an s.o. redex iff it is what we might call a *sub-head* redex; that is, iff it is the head redex of the largest subterm not in head normal form containing it.)

As an example, consider the term

$$\lambda x.x(\lambda y.R_1(y(xR_2)R_3)R_4)(x(R_5R_6(\lambda z.R_7x))(R_8R_9))$$

where $R_1, R_2, \cdots, R_9$ are redexes. Then $R_1, R_2, \cdots, R_9$ are all outermost redexes, but only R_1, R_5, R_8 are s.o. redexes; for instance, the reach of R_1 is the subterm $R_1(y(xR_2)R_3)R_4$ and the reach of R_5 is the subterm $R_5R_6(\lambda z.R_7x)$.

The motivation for these rather technical definitions is that the notion of a redex being an s.o. redex provides sufficiently general conditions for *both* the following two facts:

COROLLARY 5.9.

(a) *If two terms are identical except at a non-s.o. redex of either, they have the same direct approximants; in particular, therefore, this holds if either is the result of contracting a non-s.o. redex in the other.*

(b) *Suppose R is an s.o. redex in a term X and Z is the result of contracting any other redex in X. Then R has a unique residual which is an s.o. redex in Z.*

We leave the proof to the reader. Both parts are straightforward consequences of the definitions and, for (a), the observation that if a redex is not an s.o. redex there is a unique s.o. redex within whose reach it lies.

Corollary 5.9(b) is, of course, the reason for the terminology *stable* outermost. By contrast, it should be noted that Corollary 5.9(b) fails for outermost redexes in general—if R is an outermost (but not s.o.) redex, it does have a unique residual but this is not necessarily an outermost redex in Z (e.g. consider the residual of R after contracting $S \equiv (\lambda x.\lambda y.A)B$ in the term SR, where A and B are any terms). (In fact, in an earlier draft we made just this mistake in presuming that residuals of outermost redexes are always outermost redexes; this fails because a containing redex can be generated by a reduction, as in the example.)

A (β-)reduction will be called an *s.o. reduction* if each step in the reduction is defined by contraction of an s.o. redex; if such a reduction is also a standard reduction, we shall call it an *s.s.o.* (standard, stable outermost) *reduction*.

LEMMA 5.10. *If X is any term and $A' \in \mathscr{A}(X)$, there is an s.s.o. reduction of X to a term Y' for which A' is a direct approximant of Y'.*

Proof. By definition, $A' \in \mathscr{A}(X)$ implies A' is a direct approximant of a term X' to which X is reducible. By the Standardization Theorem, we can assume a reduction (5.1) from X to X' which is a standard reduction of length $n \geqq 0$. Then, roughly stated, the required reduction is obtained by simply deleting those steps in (5.1) for which the redex contracted is not an s.o. redex; more rigorously, we argue the existence of Y' by induction on n as follows.

If $n = 0$ there is nothing to prove, so suppose $n > 0$. We can assume also that R_n is an s.o. redex, otherwise A' is a direct approximant of X_{n-1} (by Corollary 5.9(a)) and the result follows by the induction hypothesis.

Now let j be the *least* integer $(0 \leqq j < n)$ such that the part of the standard reduction (5.1) from X_j to $X_n \equiv X'$ is an s.o. reduction.

If $j = 0$ we are done, so suppose $0 < j < n$. By choice of j, R_{j+1} is an s.o. redex in X_j, while R_j is contained within the reach of some s.o. redex S in X_{j-1}. Then S has a unique residual S' which is an s.o. redex in X_j (by Corollary 5.9(b)), and the contractum C_j of R_j is contained within the reach of S'. Since (5.1) is a standard reduction, R_{j+1} lies in or to the right of C_j; but since R_{j+1} is an s.o. redex and C_j lies within the reach of S', this can occur only if R_{j+1} lies entirely to the right of C_j (in fact, entirely to the right of the reach of S').

The standardness of the reduction (5.1) now implies that the part of (5.1) from X_j to $X_n \equiv X'$ is a (s.s.o.) reduction lying entirely to the right of C_j in X_j. Hence, there is a corresponding s.s.o. reduction

$$X_{j-1} \equiv Y_j \xrightarrow{R_{j+1}} Y_{j+1} \xrightarrow{R_{j+2}} \cdots \xrightarrow{R_n} Y_n \equiv Y'$$

lying entirely to the right of the non-s.o. redex R_j in X_{j-1}, so that X' matches Y' except at a non-s.o. occurrence of R_j in Y'.

Hence, the direct approximant A' of X' is a direct approximant of Y'. Since

$$X \equiv X_0 \to X_1 \to \cdots \to X_{j-1} \equiv Y_j \to Y_{j+1} \to \cdots \to Y_n \equiv Y'$$

is now a standard reduction of length $n-1$, the result follows by the induction hypothesis. □

Next we need the following lemma comparing the relative lengths of certain s.o. reductions:

LEMMA 5.11. *Suppose*

(a) *the reduction* (5.1) *from X to X' is an s.o. reduction,*

(b) *Z is the result of contracting any β-redex S in X, and*

(c) *Z' is the result of a complete reduction of X' relative to the residuals of S in X'.*

Then there is an s.o. reduction from Z to Z' of length $m \leqq n$, with equality iff none of the redexes $R_1, R_2, \cdots, R_n$ contracted in (5.1) *is a residual of S (with R_1 considered a residual of S iff it is S).*

The lemma is, in fact, a partial converse to one derived in [3, p. 140] in the proof of the Standardization Theorem, and we shall prove it using essentially the same technique. The converse is only partial because the method of proof depends, in Case 2 below, on the given reduction being an s.o. reduction. If (5.1) is an arbitrary reduction, there is still a reduction from Z to Z', but its length may be greater than n; we shall need the decrease in the length of reduction in order to apply the lemma inductively in the succeeding corollary.

Proof. We may assume that S is not R_1, as all claims are trivially obtained in that case.

For $i = 1, 2, \cdots, n$, let Σ_i denote the set of residuals of S in X_i. We shall determine a reduction

$$Z \equiv Z_0 \xrightarrow{P_1} Z_1 \xrightarrow{P_2} Z_2 \longrightarrow \cdots \xrightarrow{P_m} Z_m \equiv Z' \tag{5.2}$$

and integers $k_0, k_1, \cdots, k_n$ satisfying the conditions: for $i = 0, 1, 2, \cdots, n$:

(i) for $j = k_i$, P_{j+1} is an s.o. redex in Z_j,

(ii) $k_0 = 0$, and either $k_{i+1} = k_i$ or $k_{i+1} = k_i + 1$,

(iii) if $j = k_i$, then a complete reduction relative to Σ_i converts X_i to Z_j.

(The various quantities involved are depicted in Fig. 3, where the dotted arrows labeled with Σ_i's denote complete reductions relative to Σ_i.) We prove (i), (ii), and (iii) simultaneously by induction on an index q over the range $0 \leqq q \leqq n$.

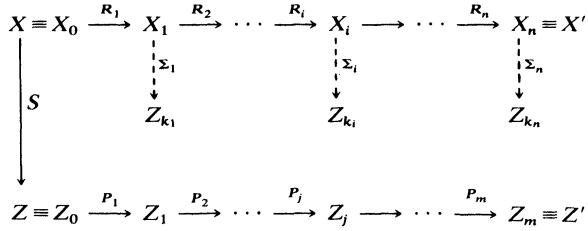


FIG. 3

For $q = 0$ we need only set $Z_0 \equiv z$ and $k_0 = 0$ and take P_1 as the residual (unique by Corollary 5.9(b)) of R_1 in Z for (i)–(iii) to hold.

Now assume (i), (ii), (iii) are satisfied for all $i \leqq q$ for some q in the range $0 \leqq q < n$. Let $p = k_q$, and let W denote the result of a complete reduction of X_{q+1} relative to Σ_{q+1}. There are two cases to consider, according as R_{q+1} is or is not a residual of S in X_q:

Case 1. *R_{q+1} is a residual of S in X_q.* Then, setting $k_{q+1} = k_q$ gives (i) and (ii) for $i = q + 1$. For (iii) notice that

$$X_q \xrightarrow{R_{q+1}} X_{q+1} \xrightarrow{\Sigma_{q+1}} W$$

is a complete reduction of X_q relative to Σ_q; hence $W \equiv Z_p$ by the lemma of parallel moves and the induction hypothesis (iii) for $i = q$, which gives (iii) for $i = q + 1$.

Case 2. *R_{q+1} is not a residual of S in X_q.* Since R_{q+1} is an s.o. redex in X_q (by supposition (a)), by applying Corollary 5.9(b) to each step in any complete reduction relative to Σ_q from X_q to Z_p, we see that R_{q+1} must have a unique residual, say R'_{q+1}, which is an s.o. redex in Z_p. Setting $P_{p+1} \equiv R'_{q+1}$ and $k_{q+1} = p + 1 = k_q + 1$ gives (i) and (ii) for $i = q + 1$. For (iii), Fig. 4 shows two complete reductions relative to the set $\Sigma_q \cup \{R_{q+1}\}$ of redexes in X_q; hence $W \equiv Z_{p+1}$ by the lemma of parallel moves.

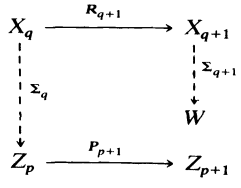


FIG. 4

This completes the induction. Now setting $m = k_n$, from (iii) with $i = n$ we have $Z_m \equiv Z'$. From (ii) and the manner in which the k_i were defined, we have $m = k_n \leqq n$ with equality holding iff Case 2 applies for all q. □

COROLLARY 5.12. *Suppose there is an s.o. reduction from* $\mathbf{C}[M]$ *to a term* X'. *Then there are terms* M' *and* Z^* *such that* M β-**red** M', X' β-**red** Z^*, *and* $\mathbf{C}[M']$ *can be reduced to* Z^* *without contraction of a residual of a redex in* M'. (See Fig. 5, where

the wavy arrow denotes a reduction without contraction of a residual of a redex in M'.)

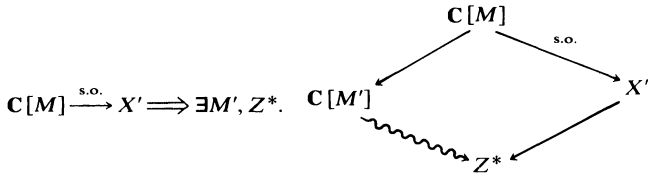


FIG. 5

Proof. Taking $X \equiv \mathbf{C}[M]$, let (5.1) be the given s.o. reduction of $\mathbf{C}[M]$ to X'. If none of the redexes $R_1, R_2, \cdots, R_n$ in (5.1) is a residual of a redex in M, we are done. Otherwise, let S be a redex in M for which some R_i is a residual of S (e.g. choose S such that i is least), and let M' be the result of contracting S in M. If we take $Z \equiv \mathbf{C}[M']$ in Lemma 5.11, there is a term Z' such that X' β-**red** Z' and $\mathbf{C}[M']$ can be reduced to Z' by an s.o. reduction of length $< n$, from which the result follows by induction on the length of s.o. reductions. □

COROLLARY 5.13 ("only if" part of Theorm 5.2). *If* $A' \in \mathscr{A}(\mathbf{C}[M])$ *there is an* $A \in \mathscr{A}(M)$ *such that* $A' \in \mathscr{A}(\mathbf{C}[A])$.

Proof. Assume $A' \in \mathscr{A}(\mathbf{C}[M])$. By Lemma 5.10 there is an s.o. reduction of $\mathbf{C}[M]$ to a term Y' for which A' is a direct approximant of Y'. Let M' and Z^* be the terms determined by Corollary 5.12 (with $X' \equiv Y'$) such that M β-**red** M', Y' β-**red** Z^*, and $\mathbf{C}[M']$ can be reduced to Z^* without contraction of a residual of a redex in M'. Let $A \in \mathscr{A}(M)$ be the best direct approximant of M', obtained by replacing the set $\mathscr{R}$ of outermost redexes in M' by Ω, and let Z' be the Ω-match of Z^* determined by Lemma 5.3 (with $X \equiv \mathbf{C}[M']$, $X' \equiv Z^*$, and $Z \equiv \mathbf{C}[A]$), so that $\mathbf{C}[A]$ β-**red** Z'. See Fig. 6.)

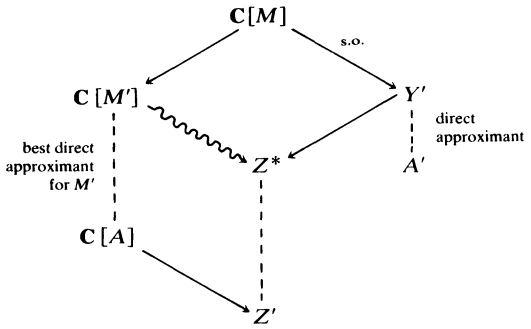


FIG. 6

Then, from Lemma 5.3, Z^* matches Z' except at a set of redexes in Z^*, so Z^* and Z' have the same direct approximants; and, by Lemma 3.3(b), every direct approximant of Y' is a direct approximant of Z^*. Hence, A' is a direct approximant of Z', which together with $\mathbf{C}[A]$ β-**red** Z' gives $A' \in \mathscr{A}(\mathbf{C}[A])$. □

This completes the proof of Theorem 5.2.

6. Related work. The proceedings of the Rome conference [2] contains much valuable related material on appropriate normal forms, in particular in the papers of Hyland [4], [5], Levy [6], and Welch [13]. To aid comparison with those papers (see also the comments of Hyland [4, p. 94]), it may be noted that the sets $\mathscr{A}(M)$ considered here are the same (except that only those members in Ω-normal form are

included) as the sets $\omega(M)$ of Hyland, but different from the sets $\mathbf{A}(M)$ of Levy (in that Levy's treatment specifically does *not* take $\lambda x.\Omega$ as an Ω-redex).

These minor differences apart, there is the following connection between the results. As was pointed out to us by one of the referees, an easy consequence of Corollary 5.13 is

COROLLARY 6.1. *If* $\mathscr{A}(M) \subseteq \mathscr{A}(N)$ *then* $\mathscr{A}(\mathbf{C}[M]) \subseteq \mathscr{A}(\mathbf{C}[N])$.

Or, in words: the relation $\mathscr{A}(M) \subseteq \mathscr{A}(N)$ defines what Hyland [4, p. 83] calls a *partial order relation* on *terms* of the λ-calculus. (Such a relation is also sometimes referred to as a *congruence relation*, with respect to the formation rules for terms.) This is Corollary 2.4 of Hyland [4], and is the same as Levy's Theorem 3 if $\mathscr{A}$ is replaced by $\mathbf{A}$.

Two further points of comparison are worth noting:

1. Though the results are similar, the proof methods differ. Where Levy and Welch use *inside-out* reductions for their proofs, we used *stable outermost* reductions above. For the proofs in § 5 this has the advantage of being somewhat shorter, mainly because the proof could then proceed directly, in Lemma 5.10, via the standardization theorem for the basic λ-calculus, rather than first having to establish a Church–Rosser-like property for an extended calculus. (However, inside-out reductions remain interesting in their own right and, indeed, have important applications in other directions; e.g., in the Tait/Martin–Löf proof of the Church–Rosser theorem, and in Levy's work on labeled λ-calculi [6] and, as yet unpublished, on "optimal" reduction procedures.)

2. There is a basic difference in the approach of Hyland and myself versus that of Levy and Welch. For our study, the interpretation of terms is given independently of the notions associated with approximate normal forms; then Theorem 3.5 and Corollary 3.6 are read off as *properties* of the resulting models, with Theorem 5.2 and Corollary 6.1 provoiding (some of) the supporting motivation. In Levy's and in Welch's work, Theorem 3.5 is adopted as the *definition* of an interpretation of terms (in certain syntactic domains, based on sets of approximate normal forms, but differing slightly between their two papers); then Corollary 6.1 is needed in order to show that such a syntactic interpretation does provide a model (i.e. the axioms are satisfied) for the λ-calculus. Whichever approach one follows, however, the thesis is the same: in any "reasonable" model for the λ-calculus, one expects (the interpretation of) every term to be determined as a limit of (the interpretations of) its approximate normal forms. (Other conclusions, and further references, were given in [12].)

Acknowledgments. I am especially grateful to Professor Dana Scott, who first suggested the idea of investigating approximate normal forms to me, and to Martin Hyland for the notion of type assignment used in the proof of Theorem 3.5.

REFERENCES

[1] H. P. BARENDREGT, *Some extensional term models for combinatory logics and* λ*-calculi*, Ph.D. thesis, Utrecht, The Netherlands, 1971.

[2] C. BÖHM, ED., *Lambda Calculus and Computer Science Theory*, Proc. Rome Symposium, Lecture Notes in Computer Science, 37, Springer-Verlag, Heidelberg, 1975.

[3] H. B. CURRY AND R. FEYS, *Combinatory Logic*, vol. 1, North-Holland, Amsterdam, 1958.

[4] J. M. E. HYLAND, *A survey of some useful partial order relations on terms of the lambda calculus*, [2, pp. 83–95].

[5] ———, *A syntactic characterization of the equality in some models for the lambda calculus*, J. London Math. Soc. (2), 12 (1976), pp. 361–370.

[6] J.-J. LEVY, *An algebraic interpretation of the* λ*-*β*-*K*-calculus and a labelled* λ*-calculus*, [2, pp. 147–165].

[7] J. H. MORRIS, *Lambda calculus models of programming languages*, Ph.D. thesis, Project MAC, Mass. Inst. of Tech., Cambridge, 1968.

[8] R. NAKAJIMA, *Infinite norms for λ-calculus*, [2, pp. 62–82].

[9] D. SCOTT, *Lattice-theoretic models for the lambda calculus.* Princeton Univ., Princeton, NJ, unpublished, 1969.

[10] ———, *Data types as lattices*, unpublished notes, Amsterdam, 1972.

[11] ———, *Lattice-theoretic models for various type-free calculi*, IVth International Congress for Logic, Methodology, and the Philosophy of Science (Bucharest), North-Holland, Amsterdam, 1973.

[12] C. P. WADSWORTH, *The relation between computational and denotational properties for Scott's* $\mathbf{D}_\infty$*-models of the lambda calculus*, this Journal, 5 (1976), pp. 488–521.

[13] P. H. WELCH, *Continuous semantics and inside-out reductions*, [2, pp. 120–146].

SIAM J. COMPUT.
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0008 $01.00/0

CORRECTING COUNTER-AUTOMATON-RECOGNIZABLE LANGUAGES*

ROBERT A. WAGNER† AND JOEL I. SEIFERAS‡

Abstract. Correction of a string x into a language L is the problem of finding a string $y \in L$ to which x can be edited at least cost. The edit operations considered here are single-character deletions, single-character insertions, and single-character substitutions, each at an independent cost that does not depend on context. Employing a linear-time algorithm for solving single-origin graph shortest distance problems, it is shown how to correct a string of length n into the language accepted by a counter automaton in time proportional to n^2 on a RAM with unit operation cost function. The algorithm is uniform over counter automata and edit cost functions; and it is shown how the correction time depends on the size of the automaton, the nature of the cost function, and the correction cost itself. For less general cases, potentially faster algorithms are described, including a linear-time algorithm for the case that very little correction is necessary and that the automaton's counter activity is determined by the input alone. Specializing the main result to counter automata which do not use their counters gives general linear-time correction into regular languages.

Key words. correction, syntactic error correction, error correction, counter automaton, regular languages, finite automaton, nondeterministic automaton, counter automaton languages, formal languages, edit operations, strings, editing strings

1. Introduction. Every practical language processor must provide for handling malformed input. The method called "correction" proceeds by modifying the input string x into a new string y such that y is an acceptable (correct) string which minimizes the distance between x and y according to some measure of the nearness of two strings. We adopt, as a flexible measure of distance, the weighted cost of modifying x into y when the only allowable modification operations are single-character deletion, insertion, and change operations.

Previous papers [2], [4], [5], [7] have investigated the time required to correct an input string x into various types of languages. The results have depended heavily on the language class. In terms of the string x to be corrected, Table 1 gives the main results. In Table 1, s is the number of nonterminals in a regular grammar for L, and t is the number of productions in a context-free grammar for L.

TABLE 1

Language L	Correction time
Singleton $\{y\}$	$O(\|y\| \cdot \|x\|)$
Regular	$O(s^2 \cdot \|x\|)$
Context-free	$O(t \cdot \|x\|^3)$

This paper presents three new results, all simple applications of a "flow" algorithm for computing the shortest distance to each vertex of a graph from a specified origin. These results reduce correction time to

* Received by the editors April 2, 1976, and in revised form August 15, 1977. This work was supported in part by the National Science Foundation under Grant GJ-33014 to Vanderbilt University and Grant MCS77-06613 to Pennsylvania State University.

† Department of Systems and Information Science, Vanderbilt University, Nashville, Tennessee 37235.

‡ Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802.

(1) $O(t \cdot |x|)$ for regular languages generated by regular grammars with t productions,

(2) $O(st|x|^2 + s^2 t|x|)$ for languages recognizable by nondeterministic counter automata with s states and t state transitions,

(3) $O(dt|x|)$ for languages recognizable by the "counter-consistent" subclass of nondeterministic counter automata with t state transitions, where d is the actual cost or weight of the cheapest correction, a quantity not known a priori.

Results (1) and (2) above are proved under extremely general assumptions about the assignment of weights to the edit operations. Result (3) requires that every nontrivial edit operation have positive weight.

The algorithm for result (2) does require time proportional to $|x|^2$ in the worst case, so it is not very practical. Since d can be proportional to $|x|$, result (3) is not significantly better in the worst case. However, result (3) *can* be used in a "practical" correction algorithm which runs in time $O(n \cdot f(n))$ by arbitrarily "rejecting" any input string x whose correction cost exceeds $f(n)$. (The "rejection" action could produce information about the first encountered error or errors, as more conventional language processing systems do.) Here f can be any desired sublinear function; e.g., $f(n)$ can be constant, $\log_2 n$, or $\sqrt{n}$.

Even the class of counter-consistent counter automata in result (3) contains relatively small machines which recognize languages of some practical interest. For example, an endmarked version $E@$ of the set E of comma-free FORTRAN arithmetic expressions can be recognized by a deterministic counter-consistent counter automaton having just four states. (See Fig. 1 for a grammar which generates the language E.) Therefore, one can correct x quickly into E by using result (3) to correct $x@$ into $E@$.

Note that result (2) is worse than linear even in terms of the size of the counter automaton. By restricting the weight assignment as in result (3) and by slightly restricting the counter automaton itself so that it cannot increment its counter without advancing its input, however, we can reduce correction time to

(4) $O(t|x|^2 + dt|x|)$ for the languages recognized by such automata with t state transitions, where d is an a priori upper bound on the cost of cheapest correction.

Result (4), applicable to a wider class of counter-automaton languages than result (3), gives a bound on correction time which grows quadratically with $|x|$ but only linearly with the size of the state transition diagram for the automaton.

We first present a graph model of the correction process, which converts an automaton M_1 and a string x into a new automaton M whose language is the set of all sequences of edit operations that modify x into strings accepted by M_1. By solving a single-origin shortest distance problem in a graph obtained from the state transition diagram of M, we can then find an edit sequence of minimum weight accepted by M. The construction of M and the graph appear in the proof of Theorem 1 in §3.

To solve the single-origin shortest distance problem of Theorem 1, we use the algorithm *FLOW* [6]. This algorithm, outlined and briefly justified in the Appendix, solves such problems in time proportional to $\max(E, V, D)$ for a directed graph with E edges, V vertices, and greatest calculated shortest distance D. We show in the Appendix how to modify *FLOW* to specialize it to the application in Theorem 1. The resulting algorithm *FLOWR* generates the graph of Theorem 1 directly from M_1, x, and the weight assignment. The generation is so linked to the shortest distance calculation that only those vertices within distance D, the minimum cost of correcting x into $L(M_1)$, of the origin and the edges that connect them are generated. Thus

FLOWR requires only time $O(\max(E', D))$ to solve the shortest distance problem, where E' is the number of edges leaving vertices within distance D of the origin.

The graph of Theorem 1 is composed of multiple copies of the state transition diagram of M_1. One copy occurs for each pair consisting of a prefix of x and a possible counter contents for M_1 after it has read an "edit image" of that prefix. To use Theorem 1 we must limit the counter values we must consider to a finite set. Result (1) is obtained trivially as Corollary 1 by limiting the counter to a single value. Result (4) is obtained as the slightly less trivial Corollary 2. In § 4 we derive a general $O(|x|)$ bound on the counter contents, and this yields result (2) as Theorem 2. By separately limiting the counter values that can be paired with each prefix of x for counter-consistent automata, we obtain result (3) as Theorem 3 in § 6. In § 5 we consider cases in which the a priori bound d of result (4) can be *computed.*

Finally, let us note that the various correction algorithms presented here and elsewhere can be combined to give a single algorithm which runs asymptotically as fast as the fastest algorithm in every case. This can be done simply by running the finitely many algorithms in parallel, but more practical combinations can be developed to take advantage of the algorithms' similarities.

2. Definitions. We choose to view a counter automaton as a pushdown automaton with a dedicated bottom-of-store symbol (\$) and only one other pushdown symbol (¢). Formally, then, a *counter automaton* (CA) is a quintuple $M = (K, \Sigma, \delta, q_0, F)$, where

K is a finite set of *states,*

$q_0 \in K$ is a designated *start state,*

$F \subseteq K$ is a set of designated *final states,*

Σ is a finite *input alphabet,* and

δ is a (nondeterministic) *next-state function.*

The choice of next state depends on the current state, possibly the next input symbol (but each input symbol is considered only once), the top counter symbol (¢ unless only the bottom-of-store symbol \$ remains), and how many counter symbols are to replace the top counter symbol (0 for decrement, 1 for no change, or 2 for increment). Therefore, δ is a function from $K \times (\Sigma \cap \{\lambda\}) \times \{¢, \$\} \times \{0, 1, 2\}$ into the power set of K. If $\delta(q, a, b, 2) = \varnothing$ for all q, a, and b (i.e., M never increments its counter), then M is a *finite automaton* (FA).

Frequently below we will denote by s and t, respectively, the number of states and the number of transitions of a CA $M = (K, \Sigma, \delta, q_0, F)$. In terms of our formalization,

$$s = |K| \quad \text{and} \quad t = \sum_{q,a,b,j} |\delta(q, a, b, j)|.$$

Each total state of the CA $M = (K, \Sigma, \delta, q_0, F)$ is a triple

(state, unread suffix of input string, counter contents from top to bottom)
$\in K \times \Sigma^* \times \{¢, \$\}^*$.

Define the relation $\underset{M}{\vdash\!\!\!-\!\!\!-}$ on $K \times \Sigma^* \times \{¢, \$\}^*$ as follows:

$$(q, ax, by) \underset{M}{\vdash\!\!\!-\!\!\!-} (q', x', y') \text{ if}$$

$$q' \in \delta(q, a, b, 0), \quad x' = x, \quad y' = y, \quad \text{or}$$
$$q' \in \delta(q, a, b, 1), \quad x' = x, \quad y' = by, \quad \text{or}$$
$$q' \in \delta(q, a, b, 2), \quad x' = x, \quad y' = ¢by.$$

Define $(q, ax, by) \underset{M,d}{\vdash\!\!\!-\!\!\!-} (q', x', y')$ to mean $(q, ax, by) \underset{M}{\vdash\!\!\!-\!\!\!-} (q', x', y')$ and $|by|, |y'| \leqq d$. Define $\overset{k}{\underset{M}{\vdash\!\!\!-\!\!\!-}}, \overset{k}{\underset{M,d}{\vdash\!\!\!-\!\!\!-}}$ to be the k-fold compositions of $\underset{M}{\vdash\!\!\!-\!\!\!-}, \underset{M,d}{\vdash\!\!\!-\!\!\!-}$, respectively (i.e. $Q \overset{k}{\underset{M}{\vdash\!\!\!-\!\!\!-}} R$ if there are $Q_0, \cdots, Q_k$ with $Q = Q_0 \underset{M}{\vdash\!\!\!-\!\!\!-} Q_1 \underset{M}{\vdash\!\!\!-\!\!\!-} \cdots \underset{M}{\vdash\!\!\!-\!\!\!-} Q_k = R$, and similarly for $\overset{k}{\underset{M,d}{\vdash\!\!\!-\!\!\!-}}$). Let $\overset{*}{\underset{M}{\vdash\!\!\!-\!\!\!-}}, \overset{*}{\underset{M,d}{\vdash\!\!\!-\!\!\!-}}$ be the respective reflexive transitive closures of $\underset{M}{\vdash\!\!\!-\!\!\!-}, \underset{M,d}{\vdash\!\!\!-\!\!\!-}$ (i.e., $Q \overset{*}{\underset{M}{\vdash\!\!\!-\!\!\!-}} R$ if $Q \overset{k}{\underset{M}{\vdash\!\!\!-\!\!\!-}} R$ for some k, and similarly for $\overset{*}{\underset{M,d}{\vdash\!\!\!-\!\!\!-}}$). When no confusion can arise, we will omit the subscript M from these relations.

We define acceptance by simultaneous empty store and final state. The *language accepted by* the CA $M = (K, \Sigma, \delta, q_0, F)$ is

$$L(M) = \{x \in \Sigma^* | (q_0, x, \$) \overset{*}{\underset{M}{\vdash\!\!\!-\!\!\!-}} (q, \lambda, \lambda) \text{ for some } q \in F\},$$

and the sublanguage accepted *with counter bounded by* d is

$$L(M, d) = \{x \in \Sigma^* | (q_0, x, \$) \overset{*}{\underset{M,d}{\vdash\!\!\!-\!\!\!-}} (q, \lambda, \lambda) \text{ for some } q \in F\}.$$

Let Σ be a finite alphabet. The members of the alphabet $\Delta = \{(a \to b) | a, b \in \Sigma \cup \{\lambda\}, ab \neq \lambda\}$ are called *edit operations* over Σ. The edit operation $(a \to b)$ is an *insertion*, *deletion*, or *change* if $a = \lambda \neq b$, $a \neq \lambda = b$, or $a \neq \lambda \neq b$, respectively. An edit sequence $S \in \Delta^*$ *edits* $x \in \Sigma^*$ *to* $y \in \Sigma^*$, denoted $x \underset{S}{\to} y$, if

either $S = \lambda$ and $x = y = \lambda$
or the following hold for some a, b, S', x', y':

$$S = (a \to b)S',$$
$$x = ax',$$
$$y = by',$$
$$x' \underset{S'}{\to} y'.$$

Informally, note that every original character is edited exactly one time (perhaps by a trivial change $(a \to a)$) and that no new character is ever edited. For $L \subseteq \Sigma^*$, define $x \underset{S}{\to} L$ (*S corrects x into L*) to mean that $x \underset{S}{\to} y$ holds for some $y \in L$. Note that y is determined by x and S.

Any *edit cost function* $W: \Delta \to N$ (where N is the set of nonnegative integers) can be extended to a cost function $W: \Delta^* \to N$ by the rule $W(SS') = W(S) + W(S')$. We consider only cost functions which satisfy

$$W(a \to a) = 0,$$

$$W(a_1 \to a_3) \leqq W(a_1 \to a_2) + W(a_2 \to a_3).$$

Informally, these conditions assure that no editing cost could be saved by editing some original character less than once or by editing the results of earlier editing. When we occasionally consider a *partial* edit cost function, prohibiting those edit operations not in the domain of W, we will interpret the second condition above as insisting that $W(a_1 \to a_3)$ be defined if both $W(a_1 \to a_2)$ and $W(a_2 \to a_3)$ are. Frequently below we will denote by c the cost of the most expensive edit operation allowed (i.e., $c = \max\{W(S) | S \in \Delta \text{ and } W(S) \text{ is defined}\}$).

For any edit cost function W as above and any nonempty language $L \subseteq \Sigma^*$, define

$$C_W(x, L) = \min\{W(S) | x \underset{S}{\to} L\}.$$

(The set $\{W(S) | x \underset{S}{\to} L\}$ is nonempty if W is total, since x can be edited to any $y \in L$ by a sequence of $|x|$ deletions followed by $|y|$ insertions.) We call the problem of determining $C_W(x, L)$ and some S with $W(S) = C_W(x, L)$, $x \underset{S}{\to} L$ (some *minimum-cost correction sequence*) the problem of *correcting x into L*. In the case that W is only a partial function on Δ, the problem also includes deciding whether correction is possible using only the allowed edit operations (those in the domain of W).

The string-to-string correction problem, the problem of correcting an arbitrary string into an arbitrary singleton language, can be solved in time proportional to the product of the lengths of the two strings [7]. The problem of correcting an arbitrary string into any fixed context-free language can be solved in time proportional to the cube of the length of the string [2]. We show below that a string can be corrected into a CA language in time proportional to just the square of the length of the string and into a regular language in time proportional to just the string length (earlier reported in [5]). In neither case, moreover, does the time grow very quickly with the size of the CA or FA specifying the language.

3. The graph construction. The key to our results is the construction in the proof of Theorem 1 of this section of a directed graph from any given counter automaton M_1, string x, and positive integer m. The graph G is constructed as the "configuration transition diagram" of a new automaton M with $L(M) = \{S | x \underset{S}{\to} L(M_1, m)\}$. By choosing m sufficiently large and labeling each edge in G with the edit operation and cost associated with the corresponding transition of M, we transform the problem of correcting x into $L(M_1)$ to a single-origin shortest distance problem. The relevant properties of the algorithm we use for solving this single-origin shortest distance problem are summarized in Lemma 1 below.

DEFINITION. *A weighted directed graph* is a triple $G = (V, E, W)$, where

V is a finite set of *vertices*,

$E \subseteq V \times V$ is a set of *edges*,

$W : E \to N$ is an assignment of *weights* to the edges.

A *path from vertex* s *to vertex* v is a sequence

$$(v_1, v_2)(v_2, v_3)\cdots(v_{k-1}, v_k)\in E^*,$$

where $s = v_1$, $v = v_k$. Extend W to $W: E^* \to N$ by the rule $W(pq) = W(p) + W(q)$. For $s, v \in W$, define

$$D_s(v) = \min\{W(p) | p \text{ is a path from } s \text{ to } v\}.$$

LEMMA 1. *Given a weighted directed graph* $G = (V, E, W)$, *a single designated origin vertex* $s \in V$, *and a designated set* $F \subseteq V$ *of destination vertices, a RAM with unit operation cost function* [1] *can find a path* p *from* s *to a member of* F *with*

$$W(p) = \min\{D_s(v) | v \in F\}$$

in time proportional to

$$W(p) + |\{(u, v) \in E | D_s(u) \leqq W(p)\}|.$$

Proof. In order of increasing distance from s, mark each unmarked vertex with both the weight of a shortest path to it and the preceding vertex on that path, stopping when a member of F gets marked. "Partly traversed" edges can be stored sorted in a circular array of linked lists according to weight yet to be traversed. A Pidgin ALGOL version of this algorithm is included as an appendix to this paper; further details appear in [6]. □

THEOREM 1. *Given*

(i) *an arbitrary CA* $M_1 = (K, \Sigma, \delta_1, q_0, F)$ *with* $t = \sum_{q,a,b,j} |\delta_1(q, a, b, j)|$ *state transitions,*

(ii) *an arbitrary edit cost function* $W: \Delta \to N$, *where* $\Delta = \{(a \to b) | a, b \in \Sigma \cup \{\lambda\}, ab \neq \lambda\}$,

(iii) *an arbitrary word* $x \in \Sigma^*$, *and*

(iv) *an integer* m *so large that* $C_W(x, L(M_1, m)) = C_W(x, L(M_1))$,

correction of x *into* $L(M_1)$ *can be performed in time proportional to*

$$\max(C_W(x, L(M_1)), t \cdot m \cdot |x|)$$

on a RAM with unit cost function.[1]

Proof. Let M_1, t, W, Δ, x, m be as described, with $x = a_1 \cdots a_n$, $a_i \in \Sigma$ for $1 \leqq i \leqq n = |x|$. We will describe a new CA M with input alphabet Δ. Informally, M will apply its input edit sequence to x (if possible) and behave like M_1 on the result. The string x will be built into M, the second component of each state indicating how many characters of x have been edited so far. Formally, $M = (K \times \{0, \cdots, n\}, \Delta, \delta, [q_0, 0], F \times \{n\})$, where δ is defined as follows for each accessible $q \in K$ (of which there are at most $t+1$), $i \in \{0, \cdots, n\}$, $a' \in \Sigma$, $b \in \{¢, \$\}$, $j \in \{0, 1, 2\}$:

$$\delta([q, i], (\lambda \to a'), b, j) = \delta_1(q, a', b, j) \times \{i\}$$

(insertion of a' somewhere between a_i and a_{i+1}),

$$\delta([q, i], (a_{i+1} \to a'), b, j) = \delta_1(q, a', b, j) \times \{i+1\}$$

(change of a_{i+1} to a'),

$$\delta([q, i], (a_{i+1} \to \lambda), b, 1) = \{[q, i+1]\}$$

[1] For simplicity, we slightly abuse the proportionality terminology in such statements. The correct statement here uses a product such as $(t+1)(m+1)(|x|+1)$ in place of $t \cdot m \cdot |x|$, but the distinction matters only in the degenerate case that one of the factors is zero.

(deletion of a_{i+1}),

$$\delta([q, i], \lambda, b, j) = \delta_1(q, \lambda, b, j) \times \{i\}.$$

For arguments not mentioned above, the value of δ is $\varnothing$.

By design, $L(M, m) = \{S \in \Delta^* | x \underset{S}{\rightarrow} L(M_1, m)\}$. Since m is so large that $C_W(x, L(M_1, m)) = C_W(x, L(M_1))$, it follows that

$$C_W(x, L(M_1)) = \min \{W(S) | S \in L(M, m)\}.$$

Informally, then, $C_W(x, L(M_1))$ is just the cost of a cheapest path from the start state to a final state in a state transition diagram for a finite automaton accepting $L(M, m)$. In particular, let G be the weighted directed graph with vertex set $K \times \{0, \cdots, n\} \times \{0, \cdots, m\}$ and with an edge of weight $W(a \rightarrow b)$ or 0 from $[q, i, j+1]$ to $[q', i', |y|]$ whenever

$$([q, i], (a \rightarrow b), \mathfrak{c}^j\$) \underset{M}{\vdash} ([q', i'], \lambda, y)$$

or

$$([q, i], \lambda, \mathfrak{c}^j\$) \underset{M}{\vdash} ([q', i'], \lambda, y),$$

respectively. The minimum-cost sequences in $L(M, m) \subseteq \Delta^*$ correspond to the paths of minimum cost in G from the vertex $[q_0, 0, 1]$ to vertices in $\{[q, n, 0] | q \in F\}$. By the algorithm of Lemma 1, such a path of minimum cost can be found in time proportional to the larger of the minimum cost ($C_W(x, L(M_1))$) and the number of edges in G. The number of edges in G is at most mt' if M has t' transitions. From the states in $K \times \{i\}$ for each particular i, M has at most t insertion-transitions, at most t change-transitions, at most $2(t+1)$ deletion-transitions (at most two from $[q, i]$ for each of at most $t+1$ accessible states $q \in K$), and at most t λ-transitions; hence $t' \leqq 5(t+1)(n+1)$, and the number of edges is at most proportional to mtn. □

Remark. A path of minimum cost above need not go through any vertex twice, so for $s = |K|$ the minimum cost cannot exceed the number $s(n+1)(m+1)$ of vertices times the weight c of the costliest edge. This gives a time bound proportional to max ($csmn$, tmn).

If we disallow some of the edit operations in Δ and omit their transitions from M, then the algorithm will still discover an edit sequence S with $x \underset{S}{\rightarrow} L(M_1)$, $W(S) = C_W(x, L(M_1))$ in time proportional to max $(C_W(x, L(M_1)), tmn)$ *if there is one.* Because the minimum cost cannot exceed $s(n+1)(m+1)c$, on the other hand, the shortest path algorithm of Lemma 1 can conclude, after time proportional to max ($csmn$, tmn) without detecting such a sequence, that none exists. This gives a time bound proportional to max ($csmn$, tmn) for both determination of correctability and actual correction.

Every finite automaton M_1 is a counter automaton which satisfies $L(M_1, 1) = L(M_1)$. If we restrict attention to FA's in Theorem 1, therefore, the parameter m need not be supplied; $m = 1$ will always suffice. The result is Corollary 1.

COROLLARY 1. *Given*

(i) *an arbitrary FA M_1 with input alphabet Σ and t state transitions,*

(ii) *an arbitrary edit cost function W: $\{(a \rightarrow b) | a, b \in \Sigma \cup \{\lambda\}, ab \neq \lambda\} \rightarrow N$, and*

(iii) *an arbitrary word $x \in \Sigma^*$,*

correction of x into $L(M_1)$ can be performed in time proportional to $\max(C_W(x, L(M_1)), t \cdot |x|)$.

In terms of the number s of states of the FA M_1 and the length n of x, the algorithm earlier reported in [5] requires time proportional to s^2n. Corollary 1 represents an improvement when $t < s^2$ and $C_W(x, L(M_1)) < s^2n$. In practice, both conditions often do hold. (Note that s, t, respectively, are the numbers of nonterminals, productions in the regular grammar corresponding to M_1.)

For any fixed m, there is a counter automaton M_1 with $L(M_1, m) = \varnothing \neq L(M_1)$. In the case of general counter automata, therefore, there is no fixed m which can serve as above. The following argument, however, shows that m can still be left implicit (to be computed) in a setting that is only slightly restricted if an a priori bound on $C_W(x, L(M_1))$ is available. We show in the next section that m can always be left implicit.

COROLLARY 2. *Given*

(i) *an arbitrary CA* $M_1 = (K, \Sigma, \delta_1, q_0, F)$ *with* t *state transitions and with* $\delta_1(q, \lambda, ¢, 2) = \varnothing$ *for every state* q,

(ii) *an arbitrary edit cost function* W *and integer* $e > 0$ *with* $W(\lambda \to a) \geqq e$ *for each* $a \in \Sigma$ *satisfying* $\delta_1(q, a, ¢, 2) \neq \varnothing$ *for some state* q,

(iii) *an arbitrary word* $x \in \Sigma^*$, *and*

(iv) *an upper bound* d *on* $C_W(x, L(M_1))$,

correction of x *into* $L(M_1)$ *can be performed in time proportional to*

$$\max\left(d, t|x|^2, \frac{d}{e}t|x|\right) \leqq \max(t|x|^2, dt|x|).$$

Proof. Let M_1, W, e, x, d be as described. (Of course e can actually be computed quickly from M_1 and W.) Take S with $W(S) = C_W(x, L(M_1)) \leqq d$, and take M as in the proof of Theorem 1. By showing

$$S \in L(M, |x| + d/e + 2) = \left\{S' | x \underset{S'}{\to} L(M_1, |x| + d/e + 2)\right\},$$

we will conclude that

$$C_W(x, L(M_1, |x| + d/e + 2)) = C_W(x, L(M_1)).$$

Hence, correction can be performed by computing $m = |x| + d/e + 2$ and applying Theorem 1.

By design, $S \in L(M)$. Consider any accepting computation of S by M. Because $\delta([q, i], \lambda, ¢, 2) = \delta_1(q, \lambda, ¢, 2) \times \{i\} = \varnothing$ for every q and i, no λ-transition in this computation increments the counter, except possibly from \$ to ¢\$. By design of M, no deletion-transition increments the counter either. There are at most n change transitions, since each one increments the second component of the state. Except for insertion-transitions, therefore, the counter is incremented at most $n = |x|$ times (except from \$ to ¢\$). Since $\delta([q, i], (\lambda \to a'), ¢, 2) = \delta_1(q, a', ¢, 2) \times \{i\}$, the cost of the insertion consumed by each insertion-transition which increments the counter (except from \$ to ¢\$) is at least e. From $W(S) \leqq d$, it follows that there are at most d/e such transitions. In the entire computation, therefore, the counter is incremented at most $n + d/e$ times, except from \$ to ¢\$. Therefore, the longest possible counter is $¢^{n+d/e}¢\$$. □

4. A general bound on m. We show in this section that any s-state CA M_1, edit cost function W, and string x to be corrected into $L(M_1)$ satisfy

$$C_W(x, L(M_1)) = C_W(x, L(M_1, m))$$

for $m = s(|x|+s)+1$. This value of m is easy to compute, so it will follow by Theorem 1 that such correction can be performed in time proportional to

$$\max(C_W(x, L(M_1)), st|x|^2, s^2t|x|),$$

where M_1 has t state transitions.

The bound $m = s(|x|+s)+1$ is obtained by recalling that

$$\{S | x \underset{S}{\rightarrow} L(M_1, m)\} = L(M, m)$$

for the CA M described in the proof of Theorem 1 and by showing that for each edit sequence $S \in L(M)$ there is a sequence $S' \in L(M, m)$ with $W(S') \leqq W(S)$. We assure $W(S') \leqq W(S)$ by obtaining S' as a permutation of a subsequence of S. To this end, we show in Lemma 2 below how to permute accepting computations by M to limit the number of long subcomputations of the form $([q, i], x, y) \overset{*}{\vdash} ([q', i'], x', y')$ for $i = i'$; we show in Lemma 3 how to excise from the resulting computations some of the few remaining long subcomputations of this form.

Lemmas 2 and 3 are stated and proved for any fixed CA M having certain essential features of the CA described in the proof of Theorem 1. Let $M = (K \times \{1, \cdots, n\}, \Delta, \delta, [q_0, 1], F \times \{n\})$ be any fixed CA with

$$\delta([q, i], a, b, j) \subseteq K \times \{i, i+1\},$$

$$\delta([q, i], a, b, j) \cap (K \times \{i\}) = \delta_1(q, a, b, j) \times \{i\}$$

for every q, i, a, b, j, where δ_1 does not depend on i. The first condition forces the second components of the states in any computation to form a nondecreasing sequence. The second condition will enable us to replace certain subcomputations beginning and ending in state $[q, i]$ by relocated but otherwise similar subcomputations beginning and ending in a state $[q, i']$ with $i' \neq i$.

The easiest subcomputations to manipulate depend only on the top counter symbol and result in a net counter change of at most 1. Any accepting computation in which the counter contents reaches $¢^{m+1}\$$ can be parsed into a sequence of $2m+3$ such subcomputations, the first $m+1$ of which increment the counter and the last $m+2$ of which decrement the counter. This leads us to define an *m-level derivation of* $x \in \Delta^*$ to be a sequence of $2m+2$ subcomputations which chain together into an accepting computation by M of x and

each of which except the last depends only on the top counter symbol,
the first $m+1$ of which result in net counter increments of 1 each,
the next m of which result in net counter decrements of 1 each,
the last of which results in a net counter decrement of 2 (emptying the counter).

Formally, an m-level derivation consists of

a sequence $[q_0, 1] = [q_0, i_0], \cdots, [q_{m+1}, i_{m+1}] = [p_{m+1}, j_{m+1}], \cdots, [p_0, j_0] \in F \times \{n\}$ of states,
a sequence $v_0, \cdots, v_m, w_m, \cdots, w_0$ of input subwords with $x = v_0 \cdots v_m w_m \cdots w_0$, and

a sequence $k_0, \cdots, k_m, l_m, \cdots, l_0$ of subcomputation lengths

such that

$$([q_0, i_0], v_0, \$) \overset{k_0}{\vdash} ([q_1, i_1], \lambda, \cent\$),$$

$$([q_d, i_d], v_d, \cent) \overset{k_d}{\vdash} ([q_{d+1}, i_{d+1}], \lambda, \cent\cent) \quad \text{for } d = 1, \cdots, m,$$

$$([p_{d+1}, j_{d+1}], w_d, \cent) \overset{l_d}{\vdash} ([p_d, j_d], \lambda, \lambda) \quad \text{for } d = m, \cdots, 1,$$

$$([p_1, j_1], w_0, \cent\$) \overset{l_0}{\vdash} ([p_0, j_0], \lambda, \lambda).$$

The *length* of such an m-level derivation D is the integer $|D| = k_0 + \cdots + k_m + l_m + \cdots + l_0$. (Note that each k_d, l_d is positive.) Partition $\{1, \cdots, m+1\}$ into the sets

$$L_D(i) = \{d \geqq 1 | i_d = i\},$$

and partition each set $L_D(i)$ into the sets

$$L_D(i, q) = \{d \geqq 1 | [q_d, i_d] = [q, i]\}.$$

Similarly define

$$R_D(i) = \{d \geqq 1 | j_d = i\},$$
$$R_D(i, q) = \{d \geqq 1 | [p_d, j_d] = [q, i]\}.$$

LEMMA 2. *Let D be an m-level derivation of x. Then there is an m-level derivation D' of some permutation x' of x, such that $|D'| = |D|$ and, for each $q \in K$, there is at most one $i \in \{1, \cdots, n\}$ with $|L_{D'}(i, q)| > 1$ and at most one $i \in \{1, \cdots, n\}$ with $|R_{D'}(i, q)| > 1$.*

Proof. The proof is by induction on the weighted sum

$$S_D = \sum_{i=1}^{n} (n-i) \cdot |L_D(i)| + \sum_{i=1}^{n} i \cdot |R_D(i)| \geqq 0.$$

The lemma is trivial for $S_D = 0$, so we give only the induction step. If $D' = D, x' = x$ does not already satisfy the lemma, then there is some $q \in K$ such that either $|\{i | |L_D(i, q)| > 1\}| > 1$ or $|\{i | |R_D(i, q)| > 1\}| > 1$. The arguments are similar, so assume $|\{i | |L_D(i, q)| > 1\}| > 1$; i.e., assume there are integers e_1, f_1, e_2, f_2 with

$$1 \leqq e_1 < f_1 < e_2 < f_2 \leqq m+1,$$
$$i_{e_1} = i_{f_1} < i_{e_2} = i_{f_2},$$
$$q_{e_1} = q_{f_1} = q_{e_2} = q_{f_2} = q,$$

where the components of D are as in the formal definition above. (Our argument makes use of only e_1, f_1, e_2.) Consider the following permutation x' of x:

$$(v_0 \cdots v_{e_1-1})\underbrace{(v_{e_1} \cdots v_{f_1-1})}(v_{f_1} \cdots v_{e_2-1})(v_{e_2} \cdots v_m w_m \cdots w_0)$$
$$= v_{\pi(0)} \cdots v_{\pi(m)} w_m \cdots w_0 = x',$$

where

$$\pi(d)=\begin{cases} d & \text{for } 0 \leqq d < e_1, \\ d+f_1-e_1 & \text{for } e_1 \leqq d < e_1+e_2-f_1, \\ d-e_2+f_1 & \text{for } e_1+e_2-f_1 \leqq d < e_2, \\ d & \text{for } e_2 \leqq d \leqq m+1. \end{cases}$$

Define

$$\rho(d)=\begin{cases} \pi(d) & \text{for } 0 \leqq d < e_1+e_2-f_1, \\ e_2 & \text{for } e_1+e_2-f_1 \leqq d < e_2, \\ \pi(d) & \text{for } e_2 \leqq d \leqq m+1. \end{cases}$$

Let D' consist of

state sequence $[q_{\pi(0)}, i_{\rho(0)}], \cdots, [q_{\pi(m+1)}, i_{\rho(m+1)}], [p_{m+1}, j_{m+1}], \cdots, [p_0, j_0]$,

subword sequence $v_{\pi(0)}, \cdots, v_{\pi(m)}, w_m, \cdots, w_0$, and

length sequence $k_{\pi(0)}, \cdots, k_{\pi(m)}, l_m, \cdots, l_0$.

Because δ_1 above "does not depend on i," D' is an m-level derivation of x'. Because π is a permutation, $|D'| = |D|$. Note that

$$\begin{aligned} e_1+e_2-f_1 \leqq d < e_2 &\Rightarrow e_1 \leqq \pi(d) < f_1 \\ &\Rightarrow i_{e_1} \leqq i_{\pi(d)} \leqq i_{f_1} \\ &\Rightarrow i_{\pi(d)} = i_{e_1} \quad (\text{since } i_{e_1} = i_{f_1}). \end{aligned}$$

Because ρ is obtained from the permutation π by changing the values at these $f_1 - e_1$ arguments d to e_2, there are only two exceptions to $|L_{D'}(i)| = |L_D(i)|$ and $|R_{D'}(i)| = |R_D(i)|$:

$$|L_{D'}(i_{e_1})| = |L_D(i_{e_1})| - (f_1 - e_1)$$

and

$$|L_{D'}(i_{e_2})| = |L_D(i_{e_2})| + (f_1 - e_1).$$

Therefore, $S_{D'}$ equals $S_D - (n - i_{e_1})(f_1 - e_1) + (n - i_{e_2})(f_1 - e_1)$, which is smaller than S_D (since $e_1 < f_1$ and $i_{e_1} < i_{e_2}$). By the induction hypothesis, therefore, there is an m-level derivation D'' of some permutation x'' of x' (and thus of x, too) such that $|D''| = |D'| = |D|$ and such that, for each $q \in K$, there is at most one i with $|L_{D''}(i, q)| > 1$ and at most one i with $|R_{D''}(i, q)| > 1$. □

LEMMA 3. *Let $x \in L(M)$. Then a permutation of some subsequence of x lies in $L(M, m+1)$ for $m = |K| \cdot (n + |K| - 1)$.*

Proof. The proof is by induction on

$$k_x = \min \{k \mid ([q_0, 1], x, \$) \overset{k}{\vdash} ([q, n], \lambda, \lambda) \text{ for some } q \in F\} \geqq 0.$$

The lemma is trivial for $k_x = 0$, so we give only the induction step. If $x \in L(M, m+1)$ does not already hold, then there is an m-level derivation D of x with $|D| = k_x$. By Lemma 2, therefore, there is an m-level derivation D' of some permutation x' of x, such that $|D'| = k_x$ and, for each $q \in K$, there is at most one $i \in \{1, \cdots, n\}$ with

$|L_{D'}(i, q)| > 1$ and at most one $i \in \{1, \cdots, n\}$ with $|R_{D'}(i, q)| > 1$. Let D' consist of the sequences

$$[q_0, i_0], \cdots, [q_{m+1}, i_{m+1}], [p_{m+1}, j_{m+1}], \cdots, [p_0, j_0],$$

$$v_0, \cdots, v_m, w_m, \cdots, w_0,$$

$$k_0, \cdots, k_m, l_m, \cdots, l_0.$$

The nonempty sets $L_{D'}(i, q)$ form a partition of $\{1, \cdots, m+1\}$, so

$$\sum_{q \in K} \sum_{i=1}^{n} |L_{D'}(i, q)| = m + 1 > |K| \cdot (n + |K| - 1).$$

For at least one particular $q \in K$, therefore, we must have

$$|L_{D'}(1, q)| + \cdots + |L_{D'}(n, q)| \geqq n + |K|.$$

At most one $i \in \{1, \cdots, n\}$ can satisfy $|L_{D'}(i, q)| > 1$, so we must actually have $|L_{D'}(i, q)| \geqq |K| + 1$ for some particular i. Similarly, there must be some particular $p \in K$, $j \in \{1, \cdots, n\}$ with $|R_{D'}(j, p)| \geqq |K| + 1$.

Either $|L_{D'}(i)|$ does or does not exceed $|R_{D'}(j)|$. The arguments are similar for the two cases, so assume $|L_{D'}(i)| \leqq |R_{D'}(j)|$. Then we can choose h so that

$$\{d + h | d \in L_{D'}(i)\} \subseteq R_{D'}(j).$$

Then $A = \{d + h | d \in L_{D'}(i, q)\}$ is a subset of $R_{D'}(j)$ with $|A| = |L_{D'}(i, q)| > K$. Because $|A| > |K|$ and $\{p_e | e \in A\} \subseteq K$, we must be able to find $e, e' \in A$ such that $p_e = p_{e'}$, $e < e'$. Expressing $e, e' \in A$ as $e = d + h$, $e' = d' + h$, we then get

$$d' - d = e' - e > 0,$$

$$[q_d, i_d] = [q_{d'}, i_{d'}] \quad (\text{since } q_d = q_{d'}, i_d = i_{d'} = i),$$

$$[p_e, j_e] = [p_{e'}, j_{e'}] \quad (\text{since } p_e = p_{e'}, j_e = j_{e'} = j).$$

Let x' be the following subsequence of x: $v_0 \cdots v_{d-1} v_{d'} \cdots v_m w_m \cdots w_{e'} w_{e-1} \cdots w_0$. Then the following sequences make up an $(m - (d' - d))$-level derivation D'' of x' with $|D''| < |D'|$:

$$[q_0, i_0], \cdots, [q_{d-1}, i_{d-1}], [q_{d'}, i_{d'}], \cdots, [q_{m+1}, i_{m+1}],$$

$$[p_{m+1}, j_{m+1}], \cdots, [p_{e'}, j_{e'}], [p_{e-1}, j_{e-1}], \cdots, [p_0, j_0],$$

$$v_0, \cdots, v_{d-1}, v_{d'}, \cdots, v_m, w_m, \cdots, w_{e'}, w_{e-1}, \cdots, w_0,$$

$$k_0, \cdots, k_{d-1}, k_{d'}, \cdots, k_m, l_m, \cdots, l_{e'}, l_{e-1}, \cdots, l_0.$$

We must have $k_{x'} \leqq |D''| < |D'|$. By the induction hypothesis, therefore, a permutation of some subsequence of x' (and thus of x, too) lies in $L(M, m+1)$. □

THEOREM 2. *Given*

(i) *an arbitrary CA M_1 with s states and t state transitions,*

(ii) *an arbitrary edit cost function W, and*

(iii) *an arbitrary word x,*

correction of x into $L(M_1)$ can be performed in time proportional to

$$\max (C_W(x, L(M_1)), st|x|^2, s^2 t|x|)$$

on a RAM with unit operation cost function.

Proof. Consider the CA M described in the proof of Theorem 1. By the discussion at the beginning of this section, we need only show that for each $S \in L(M)$ there is some $S' \in L(M, s(|x|+s)+1)$ which is a permutation of a subsequence of S.

Suppose $S \in L(M)$. Note that M is of the form required for Lemma 3, with n and $|K|$ in Lemma 3 equal to $|x|+1$ and s here, respectively. By Lemma 3, therefore, a permutation of some subsequence of S lies in

$$L(M, s((|x|+1)+s-1)+1) = L(M, s(|x|+s)+1). \quad \square$$

Remark. Recall that even if we prohibit some edit operations and make W only a partial function, correction can be performed in time proportional to $\max(csm|x|, tm|x'|)$, where c is the cost of the most expensive edit operation allowed. Thus general CA correction can be performed in time proportional to

$$\max(cs^2|x|^2, st|x|^2, cs^3|x|, s^2t|x|).$$

5. A potentially better bound on m. Recall that Corollary 2, when it applies, gives an upper bound on m (for Theorem 1) that depends on an a priori upper bound on $C_W(x, L(M_1))$. If the latter bound is small, as it commonly is, then the bound on m can be significantly smaller than the general one. Usually W is a total function, in which case one a priori bound on $C_W(x, L(M_1))$ is very easy to compute: the cost of deleting all the characters of x and then inserting all the characters of a minimum-length member of $L(M_1)$.

COROLLARY 3. *Given*

(i) *an arbitrary CA* $M_1 = (K, \Sigma, \delta_1, q_0, F)$ *with* t *state transitions and with* $\delta_1(q, \lambda, \$, 2) = \varnothing$ *for every state* q,

(ii) *an upper bound* k *on* $\min\{|w| \mid w \in L(M_1)\}$,

(iii) *an arbitrary (total) edit cost function* W *with* $W(a \to b) \geqq c$ *for every edit operation* $(a \to b)$ *and with* $W(\lambda \to a) \geqq e$ *for each* $a \in \Sigma$ *satisfying* $\delta_1(q, a, \$, 2) \neq \varnothing$ *for some state* q, *where* $c \geqq e > 0$, *and*

(iv) *an arbitrary word* $x \in \Sigma^*$,

correction of x *into* $L(M_1)$ *can be performed in time proportional to*

$$\max\left(c|x|, ck, \frac{c}{e}t|x|^2, \frac{c}{e}kt|x|\right).$$

Proof. Compute $d = (|x|+k) \cdot c$ and apply Corollary 2. $\square$

By Theorem 2 with $W(a \to b) = 1$ for $a \neq b$ and with $x = \lambda$, we can actually *compute* $k = \min\{|w| \mid w \in L(M_1)\}$ above in time proportional to $|K|^2\, t$. This value need be computed only once for any fixed CA M_1, so the time is not significant if many strings are to be corrected into the same language $L(M_1)$. Otherwise, there is a general value of k which can be used.

LEMMA 4. *If* M_1 *is an* s*-state CA with* $L(M_1) \neq \varnothing$ *and* W *is any edit cost function, then the length of a shortest string* $x \in L(M_1)$ *with* $C_W(\lambda, \{x\}) = C_W(\lambda, L(M_1))$ *does not exceed* $s^3 + s$. *In particular, the length of the shortest string in* $L(M_1)$ *does not exceed* $s^3 + s$.

Proof. Let $M_1 = (K, \Sigma, \delta, q_0, F)$ be any fixed CA, and let d be any fixed positive integer. By induction on

$$k_x = \min\left\{k \mid (q_0, x, \$) \overset{k}{\underset{M_1, d}{\vdash}} (q, \lambda, \lambda) \text{ for some } q \in F\right\} \geqq 0,$$

we prove that each $x \in L(M_1, d)$ has a subsequence $x' \in L(M_1, d)$ with $|x'| \leqq ds$. This will suffice because Lemma 3 (with $n = |\lambda| + 1 = 1$) implies that at least one member of $L(M_1)$ with the desired property lies in $L(M_1, s^2 + 1)$.

The assertion to be proved is trivial for $k_x = 0$, so we give only the induction step. If $|x| \leqq ds$ does not already hold, then we must have $k_x \geqq |x| > ds$. Fix some computation

$$(q_0, x, \$) = (q_0, x_0, y_0) \underset{M_1, d}{\vdash\!\!\!\longrightarrow} \cdots \underset{M_1, d}{\vdash\!\!\!\longrightarrow} (q_{k_x}, x_{k_x}, y_{k_x}) = (q, \lambda, \lambda) \quad \text{with } q \in F.$$

Since $1 \leqq |y_i| \leqq d$ for each $i < k_x$ and $k_x > ds$, there must be some particular i, j with $0 \leqq i < j < k_x$, $q_i = q_j$, $y_i = y_j$. If z is the prefix of x with $x = zx_i$, then $x' = zx_j$ is a subsequence of x with $x' \in L(M_1, d)$, $k_{x'} \leqq k_x - (j - i) < k_x$. By the induction hypothesis, therefore, there is a subsequence x'' of x' (and thus of x, too) with $x'' \in L(M_1, d)$, $|x''| \leqq ds$. □

Remark. We conjecture that the minimum length is actually closer to s^2 than to s^3.

COROLLARY 4. *Given*

(i) *an arbitrary CA* $M_1 = (K, \Sigma, \delta_1, q_0, F)$ *with* t *state transitions and with* $\delta_1(q, \lambda, ¢, 2) = \varnothing$ *for every state* q,

(ii) *an arbitrary (total) edit cost function* W *with* $W(a \to b) \leqq c$ *for every edit operation* $(a \to b)$ *and with* $W(\lambda \to a) \geqq e$ *for each* $a \in \Sigma$ *satisfying* $\delta_1(q, a, ¢, 2) \neq \varnothing$ *for some state* q, *where* $c \geqq e > 0$, *and*

(iii) *an arbitrary word* $x \in \Sigma^*$,

correction of x *into* $L(M_1)$ *can be performed in time proportional to*

$$\max\left(c|x|, c|K|^3, \frac{c}{e} t|x|^2, \frac{c}{e} |K|^3 t|x|\right).$$

Proof. Compute $k = |K|^3 + |K|$ and apply Corollary 3. □

An edit cost function W with $W(a \to b)$ always the same positive constant for $a \neq b$ is a common practical choice for W. Such a choice simplifies Corollary 4.

COROLLARY 5. *Given*

(i) *an arbitrary CA* $M_1 = (K, \Sigma, \delta_1, q_0, F)$ *with* t *state transitions and with* $\delta_1(q, \lambda, ¢, 2) = \varnothing$ *for every state* q,

(ii) *a (total) edit cost function* W *with* $W(a \to b) = c > 0$ *whenever* $a \neq b$, *and*

(iii) *an arbitrary word* $x \in \Sigma^*$,

correction of x *into* $L(M_1)$ *can be performed in time proportional to*

$$\max\,(t|x|^2, |K|^3 t|x|).$$

Proof. Apply Corollary 3 to perform the correction with $W(a \to b) = 1$ whenever $a \neq b$. Then just multiply the result by c. □

6. Counter-consistent automata. An ideal language correction scheme would require only time proportional to the length n of the string to be corrected. Even if there is a small a priori bound on correction cost (d in Corollary 2), the schemes we have presented so far require time proportional to n^2; so these schemes are probably not practical. In this section we show that correction into the language accepted by a CA which is "counter-consistent" (defined below) can be performed in time proportional to n times the minimum correction cost (even if this cost is not known a priori), provided no nontrivial edit operation is free. Although this still amounts to quadratic time for strings which err at any fixed rate (such as once every million

characters), it is asymptotically much faster for strings which are correct or nearly correct as submitted. If we are willing to revert to more traditional error recovery whenever correction cost exceeds some fixed level, then this gives a linear-time algorithm for such correction. (Just cut off the algorithm after time proportional to n times the fixed level has elapsed.) More realistically, if we wish to allow correction up to cost $\log n$ or $n^{\frac{1}{2}}$, then we can perform such correction in time proportional to $n \log n$ or $n^{\frac{3}{2}}$, respectively.

DEFINITION. A CA $M = (K, \Sigma, \delta, q_0, F)$ is *counter-consistent* if there is some function $j: \Sigma \cup \{\lambda\} \to \{0, 1, 2\}$ such that

$$j(\lambda) = 1,$$

$$\delta(q, a, b, j) = \varnothing \quad \text{unless} \quad j = j(a).$$

The class of languages accepted by counter-consistent counter automata may be of some practical interest. In Fig. 1 we describe a subset of the FORTRAN arithmetic expressions whose concatenation with the endmarker @ is accepted by the counter-consistent CA shown in Fig. 2.

$$P: := I|S(E)|(E)$$

$$E: := P|E+P|E-P|E*P|E/P$$

FIG. 1. *A grammar for a subset E of the* FORTRAN *arithmetic expressions. I represents a scalar variable or constant name, S represents the name of a singly-subscripted variable or single-argument function.*

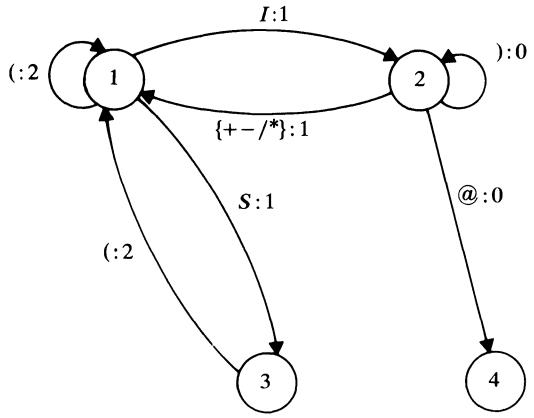


FIG. 2. *The state transition diagram of a counter automaton which accepts E@, where E is the language generated by the grammar given in Fig.* 1. *The start state is* 1, *and the only final state is* 4. *The arcs from q labeled ab: j represent the transitions in* $\delta(q, a, b, j)$. *An arc labeled a: j abbreviates a pair of arcs labeled* $a\text{¢}: j$ *and* $a\$: j$, *and an arc labeled* $\{a_1 \cdots a_k\}: j$ *abbreviates arcs labeled* $a_i: j$ *for* $i = 1, \cdots, k$.

THEOREM 3. *Given*

(i) *an arbitrary counter-consistent CA* $M_1 = (K, \Sigma, \delta_1, q_0, F)$ *with t state transitions,*

(ii) *an arbitrary edit cost function W with* $W(a \to b) = 0$ *only for* $a = b$, *and*

(iii) *an arbitrary word* $x \in \Sigma^*$,

correction of x into $L(M_1)$ *can be performed in time proportional to* $t \cdot |x| \cdot C_W(x, L(M_1))$.

Proof. Let M_1, t, W, $x = a_1 \cdots a_{|x|}$ be as described, and let $j: \Sigma \cup \{\lambda\} \to \{0, 1, 2\}$ satisfy

$$j(\lambda) = 1,$$

$$\delta_1(q, a, b, j) = \varnothing \quad \text{unless } j = j(a).$$

Construct a new CA M with state set $K \times \{0, \cdots, |x|\}$ as in the proof of Theorem 1. As in the earlier proof, the number of transitions in M from the states in $K \times \{i\}$ for each i does not exceed $5(t+1)$. Because M_1 is counter-consistent and every nontrivial edit operation has cost at least 1, $([q_0, 0], S, \$) \overset{*}{\underset{M}{\vdash}} ([q, i], \lambda, y)$ implies that $|y|$ differs from $1 + \sum_{1 \leqq i' \leqq i} (j(a_{i'}) - 1)$ (the counter length in the case of no correction to $a_1 \cdots a_i$) by at most $2 \cdot W(S)$ (2 for each correction made) in either direction. If we employ the algorithm of Lemma 1 as in the proof of Theorem 1, therefore, we discover some edit sequence in $L(M)$ of minimum cost $C_W(x, L(M_1))$ without examining more than

$$(|x|+1) \cdot (1 + 4 \cdot C_W(x, L(M_1))) \cdot 5(t+1)$$

edges in the graph we construct. The CA M is of such a regular nature that from just M_1 the necessary edges can be generated quickly on demand without constructing the whole graph (see the Appendix). In this way, since the bound on the number of examined edges exceeds the number of transitions t and the minimum cost $C_W(x, L(M_1))$, correction can be performed in time proportional to $t \cdot |x| \cdot C_W(x, L(M_1))$. □

Remark. The edit cost function need not be total for the above result.

Appendix. The algorithm FLOWR. Let $G = (V, E, W)$ be a weighted directed graph with maximum edge weight c, and let $s \in V$ be a designated origin vertex. The algorithm *FLOW* [6] is designed to preserve the following invariant as DIST is stepped through the values $0, 1, 2, \cdots$:

$$D[v] = \begin{cases} D_s(v) & \text{if } D_s(v) \leqq DIST, \\ -1 & \text{otherwise;} \end{cases}$$

$$\text{BACK}[v] = \text{some } u \text{ for which } D_s(v) = D_s(u) + W((u, v)) \text{ if } D_s(v) \leqq DIST \text{ and } v \neq s;$$

$$SORT[i] = \{(u, v) \mid D_s(u) \leqq DIST \text{ and } D_s(u) + W((u, v)) = i > DIST\}$$
(note then that $SORT[i]$ will be empty unless $DIST < i \leqq DIST + c$);

the edge $e = (u, v)$ has been handled exactly once if

$$D_s(u) \leqq DIST < D_s(u) + W(e)$$

and exactly twice if

$$DIST \geqq D_s(u) + W(e).$$

If we suppress some inessential details, such as the arc-linkage mechanisms and computing indices into SORT modulo $c+1$ to save space, then *FLOW* can be outlined as follows, assuming $D[v]$ is initially -1 for every vertex v:

```
1  procedure FLOW: begin
2    DIST := 0; ENTER(s);
3    while edges remain in SORT do begin
4      while SORT[DIST] is not empty do begin
```

```
 5          remove any edge (u, v) from SORT[DIST];
 6          if D[v] = −1 then begin BACK[v] := u; ENTER(v) end
 7        end; [Now the invariant holds.]
 8        DIST := DIST + 1
 9      end
10    end
11    procedure ENTER(v): begin
12      D[v] := DIST;
13      for each edge e leaving v do put e into SORT[DIST + W(e)]
14    end
```

Note that the time spent restoring the invariant (from line 7 back around to line 7 again) is proportional to 1 plus the number of edges handled. By the invariant, the cumulative number of edges handled through line 7 can be at most twice the number of edges (u, v) for which $D_s(u) \leqq DIST$. Therefore, the cumulative time spent by *FLOW* through line 7 is at most proportional to

$$DIST + |\{(u, v) \in E | D_s(u) \leqq DIST\}|.$$

For this paper, three modifications to *FLOW*, applied to the graph described in the proof of Theorem 1, are needed:

1) a criterion which stops *FLOW* as soon as *DIST* reaches $D_{[q_0,0,1]}([q, n, 0])$ for some $q \in F$ (this value will be $C_W(x, L(M_1))$);
2) a mechanism added to *ENTER* which generates each edge in the graph of Theorem 1 directly from $x = a_1 \cdots a_n$ and a representation of M_1; and
3) a special scheme which avoids initializing $D[v]$ for any vertex v not actually needed during the calculation, and a similar scheme to initialize the entries of *SORT*.

We call the resulting algorithm *FLOWR*. For each vertex $v = [q, n, 0]$ with q an accessible final state of M_1, *FLOWR* initializes $D[v]$ to -2. All other vertices are implicitly initialized to -1, with actual initialization postponed until the procedure *REF* below is called right before the vertex is first referenced. A device using a stack, suggested by Aho, Hopcroft, and Ullman [1, p. 71, exercise 2.12], makes this postponement possible. The same device could be used in lines 29–33 to initialize newly-used entries of *SORT* to $\varnothing$; alternatively, the method of [6], which limits *SORT*'s size to only $c + 1$ entries, may be preferable.

```
 1  procedure FLOWR: begin
 2    STK := 0;
 3    for each accessible q ∈ F do
 4      begin REF([q, n, 0]); D[q, n, 0] := −2 end;
 5    DIST := 0; REF([q0, 0, 1]); ENTERR([q0, 0, 1]);
 6    while edges remain in SORT do begin
 7      while SORT[DIST] is not empty do begin
 8        remove any edge (u, v) from SORT[DIST];
 9        REF(v); t := D[v];
10        if t < 0 then begin
11          BACK[v] := u; ENTERR(v);
12          if t = −2 then terminate the next time the end of line 14 is reached
13        end
14      end; [Now the invariant holds.]
15      DIST := DIST + 1
16    end
```

```
17  end
18  procedure REF(v): begin
19    if (0 ≦ Z[v] < STK and ST[Z[v]] = v)
20      then [v has been referenced before]
21      else begin ST[STK] := v; Z[v] := STK;
22                 D[v] := −1; STK := STK + 1 end
23  end
24  procedure ENTERR(v): begin
25    D[v] := DIST;
26    let v be [r, i, j];
27    let T be
            {[p, a, k] | p ∈ δ1(r, a, ¢, k), k < 2}   if 1 < j = m,
            {[p, a, k] | p ∈ δ1(r, a, ¢, k)}          if 1 < j < m,
            {[p, a, k] | p ∈ δ1(r, a, $, k)}          if 1 = j < m,
            ∅                                         if j = 0;
28    for each [p, a, k] ∈ T do begin
29      if a ≠ λ then
          put (v, [p, i, j − 1 + k]) into SORT[DIST + W(λ → a)];
30      if (a ≠ λ and i < n) then
          put (v, [p, i + 1, j − 1 + k]) into SORT[DIST + W(a_{i+1} → a)];
31      if a = λ then
          put (v, [p, i, j − 1 + k]) into SORT[DIST]
32    end;
33    if i < n then
          put (v, [p, i + 1, j]) into SORT[DIST + W(a_{i+1} → λ)]
34  end
```

Further commentary on *FLOWR*:

1) Each reference to any array indexed by the vertex v is preceded by the call *REF*(v), which handles any necessary initialization in constant time.

2) The initialization of $D[q, n, 0]$ to -2 for each accessible final state q is handled by a call to *REF* at the outset. Since there are at most $t+1$ accessible states, the time required is at most proportional to t.

3) Lines 9–12 are added to cause *FLOWR* to cause termination when some vertex v with $D[v] = -2$ is first reached by an edge removed from *SORT* in line 8. (By postponing termination to the end of line 14, we can find all such closest destinations.)

4) The set T in line 27 depends only on the accessible state r and how j compares with 1 and m. All such sets can be precomputed in time proportional to t.

5) Lines 28–33 of *ENTERR* correspond to lines 13–14 of *ENTER*; they generate the edges of G only as they are about to be placed in *SORT*[i] for some i. The time spent in these lines is proportional to 1 plus the cardinality of T. Since each member of T leads to the construction of either one or two edges here, the time is proportional to 1 plus the number of edges handled.

The comments above account for all the significant differences between *FLOW* and *FLOWR*. It follows that the end of line 14, where the invariant holds, is reached in time proportional to

$$t + DIST + |\{(u, v) | D_{[q_0,0,1]}(u) \leqq DIST\}|.$$

Since *FLOWR* terminates when *DIST* reaches $C_W(x, L(M_1))$, termination occurs

within time proportional to

$$t + C_W(x, L(M_1)) + |\{(u, v) | D_{[q_0,0,1]}(u) \leqq C_W(x, L(M_1))\}|.$$

If some edit operations are disallowed by making W only a partial function, then *FLOWR* will have to discover which corresponding edges to omit from G. It can do this by testing in lines 29, 30, and 33 whether $W(\lambda \to a)$, $W(a_{i+1} \to a)$, and $W(a_{i+1} \to \lambda)$, respectively, are defined. Since time is spent discovering the omitted edges, however, the timing analysis above will apply only if we *do* count each omitted edge (u, v) with $D_{[q_0,0,1]}(u) \leqq DIST$ as being a member of $\{(u, v) | D_{[q_0,0,1]}(u) \leqq DIST\}$.

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. V. AHO AND T. G. PETERSEN, *A minimum distance error-correcting parser for context-free languages*, this Journal, 1 (1972), pp. 305–312.

[3] D. GRIES, *Compiler Construction for Digital Computers*, John Wiley, New York, 1971.

[4] G. LYON, *Syntax-directed least-errors analysis for context-free languages: a practical approach*, Comm. ACM, 17 (1974), pp. 3–14.

[5] R. A. WAGNER, *Order-n correction for regular languages*, Ibid., 17 (1974), pp. 265–268.

[6] ———, *A shortest path algorithm for edge-sparse graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 50–57.

[7] R. A. WAGNER AND M. J. FISCHER, *The string-to-string correction problem*, Ibid., 21 (1974), pp. 168–173.

SIAM J. COMPUT
Vol. 7, No. 3, August 1978

0097-5397/78/0703-0009 $1.00/0

GENERALIZED SYNTAX DIRECTED TRANSLATION, TREE TRANSDUCERS, AND LINEAR SPACE*

BRENDA S. BAKER†

Abstract. When trees are denoted by "terms" or "parenthesized expressions", which are strings, the class of top-down tree transducers (automata which map trees into trees and read their input trees from the root toward the leaves) form a subclass of a nondeterministic version of the generalized syntax directed translations of Aho and Ullman. It is shown that every nondeterministic syntax directed translation (NGSDT), and therefore every top-down tree transduction, can be carried out by a Turing machine which uses an amount of work space which is linear with respect to the size of the input and output. For every n, the family consisting of the images of recognizable sets of trees under the composition of n top-down transductions is shown to be properly contained in the family of deterministic context-sensitive languages.

Key words. tree, tree transducer, syntax directed translation, context-sensitive language

Introduction. Several models of "syntax-directed transduction" have been proposed to describe the generation of machine code from parse trees during the process of compilation of a computer program. In one common formalism, the syntax directed translation scheme (SDT) [1]–[3], a transformation is associated with each rule of a context-free grammar; the transformation permutes the order of the nonterminals in the right hand side of the rule and introduces output symbols. The SDT transforms a given parse tree by applying these transformations at the nodes labeled with the corresponding rules of the grammar; at each node, the tree is altered by deleting sons labeled by terminal symbols, reordering sons labeled by nonterminals, and introducing new sons labeled by terminal symbols. The output is the yield of the new tree (that is, the string obtained by concatenating its leaves from left to right). An SDT generates exactly one segment of the output string from each subtree of the parse tree, and the length of the output string is always linear with respect to the number of nodes in the parse tree.

However, a more powerful model, called "generalized syntax directed translation" (GSDT) has also been studied [4]. In this model, more than one segment of output string may be obtained from each subtree of the parse tree; in general, the length of the output string may be exponential with respect to the number of nodes in the parse tree. Examples have been found of common programming language statements for which the natural translation into machine code can be done by a GSDT but not by an SDT [4].

Another development has been the study of automata called "tree transducers" which map trees into trees [5]–[8]. Of interest here are "top-down" tree transducers, which read input trees by beginning at the root and working toward the leaves (think of a tree as being drawn with the root at the "top"). Certain classes of syntax directed transductions correspond to certain classes of top-down tree transductions. In particular, applying a generalized syntax directed translation is equivalent to applying a deterministic top-down tree transduction and then taking the yield of the resulting tree. In § 1, we define nondeterministic generalized syntax directed translations (NGSDT's) which correspond in the same fashion to nondeterministic top-down tree

* Received by the editors December 7, 1973, and in revised form July 25, 1977.

† Computer Science Division, University of California at Berkeley, Berkeley, California 94720. This research was done while the author was at Harvard University and was supported in part by the National Science Foundation under Grant NSF-GJ-30409. Part of the research was done while the author held a National Science Foundation Graduate Fellowship.

transducers. Denoting trees by "terms" or "parenthesized expressions" (which are strings) allows us to regard nondeterministic top-down transductions as mappings from trees into strings; by this definition, nondeterministic top-down tree transductions become a subclass of the class of nondeterministic generalized syntax directed translations. We use this relationship in order to treat generalized syntax directed translations and top-down tree transductions within the same framework; the relevant definitions are presented in § 1.

In § 2, we show that for every NGSDT (and therefore every nondeterministic top-down tree transduction) G, the set of input-output pairs of G can be recognized by a deterministic Turing machine which uses an amount of work space which is linear with respect to the size of the input and output.

In § 3, we apply this result to answer (affirmatively) a question of Thatcher [5]: whether the yield of a top down tree transduction is a context-sensitive language.[1] In fact, we show that for every n, the image T of the composition of n top-down tree transductions is always a deterministic context-sensitive language, and so is yield (T).

In § 4, we investigate the class of bottom-up tree transductions (bottom-up tree transducers read trees by starting at the leaves and reading toward the root). Using the results of §§ 2 and 3, it is shown that for every n, the image of the composition of n bottom-up tree transductions is always a deterministic context-sensitive language. The results in §§ 3 and 4 are phrased in terms of a hierarchy of families of tree languages generated by top-down and bottom-up tree transductions; this hierarchy has been studied earlier in [7], [8], [13].

1. Preliminaries. In this section, we define (finite labeled) trees, nondeterministic generalized syntax directed translations, and top-down tree transducers. Finite labeled trees are treated here as special types of strings.

Rather than labeling trees over arbitrary alphabets, we restrict the alphabets so that if a symbol b occurs at two nodes α and β, then α and β have the same number of sons.

DEFINITION. A *ranked* alphabet is a pair $\langle \Sigma, r \rangle$ where Σ is a finite set of symbols and $r: \Sigma \to \mathbb{N}$ is a "ranking" function. For $b \in \Sigma$ and $n \in \mathbb{N}$, if $r(b) = n$, we say that b is of rank n. For each $n \in \mathbb{N}$, let Σ_n denote the set $r^{-1}(n)$, which is the set of symbols of Σ which have rank n.

Usually in dealing with a ranked alphabet $\langle \Sigma, r \rangle$, we will drop the r and write only Σ, since r is implicit in the notation Σ_n.

Trees are defined in terms of ranked alphabets and the set Π, which we define to be the set containing left and right brackets and comma.

DEFINITION. Let Σ be a ranked alphabet. The set Σ_* of (finite labeled) trees over the alphabet Σ, is the least set of strings in $(\Sigma \cup \Pi)^*$ such that:

1) For $b \in A_0$, $b \in \Sigma_*$.

2) For $n \geqq 1$, $b \in \Sigma_n$, and $t_1, t_2, \ldots, t_n \in \Sigma_*$, $b[t_1, t_2, \ldots, t_n] \in \Sigma_*$.

If T is a set of trees, T is said to be a *tree language.*

Next, we define nodes of trees; nodes are denoted by "Dewey decimal" notation, i.e. by a string in $\mathscr{P}^*$, where $\mathscr{P}$ denotes the set of positive integers. Concatenation is denoted here by $\cdot$, as in $i \cdot j$.

DEFINITION. For a tree $t \in \Sigma_*$, *nodes* of t and *labels* of nodes of t are defined recursively as follows:

[1] This result was also announced by Kosaraju [12].

1) If $t \in \Sigma_0$, then e is a node of t with label t;[2]
2) If $t = b[t_1, \ldots, t_n]$, where $n > 0$, $b \in \Sigma_n$, and $t_1, \ldots, t_n \in \Sigma_*$, then e is a node of t with label b, and for $1 \leqq i \leqq n$, if α is a node of t_i with label c, then $i \cdot \alpha$ is a node of t with label c.

We assign special names to certain types of nodes and to relationships between nodes as follows.

DEFINITION. Let t be a tree in Σ_*, and let α and β be nodes of t.

1) If the label of α is in Σ_0, then α is a *leaf* of t.
2) If for some $i \in \mathscr{P}$, $\alpha = \beta \cdot i$, then β is the *father* of α and α is a *son* of β.
3) The node e is the *root* of t.

Every node α of a tree t determines a subtree t/α defined as follows.

DEFINITION. Let t be a tree in Σ_*. For a node α of t, the subtree t/α of t is defined recursively as follows:

1) If $\alpha = e$, then $t/\alpha = t$;
2) If $t = b[t_1, \ldots, t_n]$, where $n > 0$, $b \in \Sigma_n$, and $t_1, \ldots, t_n \in \Sigma_*$, and $\alpha = i \cdot \beta$, where $1 \leqq i \leqq n$ and $\beta \in \mathscr{P}^*$, then $t/\alpha = t_i/\beta$.

An important parameter of a tree is its depth, defined as follows.

DEFINITION. Define the *depth* of a tree in Σ_* inductively by:

1) For $b \in \Sigma_0$, $\text{depth}(b) = 1$.
2) For $b \in \Sigma_n$, and $t_1, \ldots, t_n \in \Sigma_*$, $\text{depth}(b[t_1, \ldots, t_n]) = 1 + \max_{1 \leqq i \leqq n} (\text{depth}(t_i))$.

The *length* of a tree or string is the number of symbols occurring in it (including symbols in Π). The length of a string w is denoted by $|w|$.

An important operation on trees is the *yield* operation, which takes the leaves of a tree and concatenates them to form a string. Since trees have been defined as a special case of strings, the yield of a tree may be defined in the following manner.

DEFINITION. Let Σ be a ranked alphabet, and let $t \in \Sigma_*$. If $t = u_1 b_1 u_2 b_2 \ldots u_m b_m u_{m+1}$, where $b_1, \ldots, b_m \in \Sigma_0$, and $u_1, \ldots, u_{m+1} \in ((\Sigma - \Sigma_0) \cup \Pi)^*$, then $\text{yield}(t) = b_1 \ldots b_m$. For a set T of trees, $\text{yield}(T) = \{\text{yield}(t) | t \in T\}$. For a family $\mathscr{L}$ of sets of trees, $\text{yield}(\mathscr{L}) = \{\text{yield}(T) | T \in \mathscr{L}\}$.

To define nondeterministic generalized syntax directed translations, we need to reserve a special set of symbols for use as "variables"; these symbols will appear in rules of a translation to mark places at which strings are to be substituted when the rule is applied to an input tree. As variables, we choose the countably infinite set $X = \{x_1, x_2, x_3, \ldots\}$. For each $n > 0$, let X_n denote the set $\{x_1, x_2, \ldots, x_n\}$, and let X_0 denote the empty set.

DEFINITION. A nondeterministic generalized syntax directed translation (NGSDT) is a 5-tuple $G = (Q, \Sigma, \Delta, R, Q_0)$, where

1) Q is a finite set of *states*,
2) Σ is a finite ranked alphabet called the *input alphabet*, such that $\Sigma \cap \Pi = \phi$,
3) Δ is a finite *output alphabet*,
4) $Q_0 \subseteq Q$ is a set of *starting* states,
5) R is a finite set of *rules*, $R \subseteq \cup_{n \geqq 0} (Q \times \Sigma_n) \times (\Delta \cup (Q \times X_n))^+$. A rule is usually written in the form $(q, b) \to w$, where $q \in Q$, $b \in \Sigma_n$, and $w \in (\Delta \cup (Q \times X_n))^+$, for some $n \geqq 0$.

The behavior of an NGSDT starting in state q on input w is defined recursively as follows.

DEFINITION. Let $G = (Q, \Sigma, \Delta, R, Q_0)$ be an NGSDT, and let $q \in Q$.

1) If $b \in \Sigma_0$, then $G(q, b) = \{w | (q, b) \to w \in R\}$.

[2] The empty string is denoted by e.

2) If $b \in \Sigma_n$, $n > 0$, and $t_1, \ldots, t_n \in \Sigma_*$, then

$$G(q, b[t_1, \ldots, t_n]) = \{u_1 z_1 u_2 z_2 \ldots u_m z_m u_{m+1} \in \Delta^* | m \geqq 0, \text{ and for some } (q_1, x_{i_1}), \ldots, (q_m, x_{i_m}) \in Q \times X_n, (q, b) \to u_1(q_1, x_{i_1}) u_2 \ldots u_m (q_m, x_{i_m}) u_{m+1} \text{ is a rule in } R \text{ and for } j = 1, \ldots, m, z_j \in G(q_j, t_{i_j})\}.$$

3) If $w \in G(q, t)$, we say that G *generates* w *from* t *starting in state* q. For a tree t, the set of all strings generated from t by G is $G(s) = \bigcup_{q \in Q_0} G(q, s)$. For a set T of trees, the set of strings generated from T by G is $G(T) = \bigcup_{t \in T} G(t)$. The *transduction* performed by G is $T(G) = \{\langle s, t \rangle \in \Sigma_* \times \Delta^* | t \in G(s)\}$.

Note that the transduction performed by G is a set of ordered pairs representing the input-output relation of G. We illustrate NGSDT's in the following example [6].

Example. Let $\Sigma = \{+, \cdot, y, c\}$ be a ranked alphabet, where $+$ and $\cdot$ have rank 2, and y and c have rank 0. We construct an NGSDT G which takes the formal derivative with respect to y of the expressions represented by input trees in Σ_*, where c represents a constant. Let $\Delta = \{+, \cdot, 1, 0, y, [,]\}$. Let $G = (\{d, I\}, \Sigma, \Delta, R, \{d\})$, where

$$\begin{aligned} R = \{&(d, +) \to [(d, x_1) + (d, x_2)], \\ &(d, \cdot) \to [[(d, x_1) \cdot (I, x_2)] + [(I, x_1) \cdot (d, x_2)]], \\ &(d, y) \to 1, (d, c) \to 0\} \cup \\ &\{(I, \sigma) \to \sigma | \sigma \in \{y, c\}\} \cup \\ &\{(I, \sigma) \to [(I, x_1) \sigma (I, x_2)] | \sigma \in \{+, \cdot\}\}. \end{aligned}$$

Notice that $G(d, +[\cdot[c, y], y]) = \{[[[0 \cdot y] + [c \cdot 1]] + 1]\}$. In general, for a tree $t \in \Sigma_*$, $G(I, t)$ is the singleton set containing the expression represented by t written in infix notation, and $G(d, t)$ is the singleton set containing the formal derivative with respect to y of the expression denoted by t.

In a rule $(q, c) \to w$, where $c \in \Sigma_n$, $n > 0$, each variable x_i, $1 \leqq i \leqq n$, may occur 0 or more times in w. If x_i does not occur in w, we say that the rule *deletes* the ith subtree of the node labeled b. For $q_1 \neq q_2$, it is possible for both (q_1, x_i) and (q_2, x_i) to appear in w. Note that w is not allowed to be the empty string. One may think of a "computation" of an NGSDT as beginning with a single rule applied to the root of the input tree; if the right side of the rule contains pairs $(q_1, x_{i_1}), \ldots, (q_m, x_{i_m})$, the jth pair begins a new "computation" on the i_jth subtree of the root, starting in state q_j, for $j = 1, \ldots, m$. If trees are drawn with the root at the "top", an NGSDT may be considered to operate by reading the tree "top-down".

Certain restrictions may be placed on NGSDT's in order to ensure that the output is always a tree; the restricted NGSDT's are called top-down tree transducers. In order to specify these restrictions, we need to define the indexing of trees by sets of symbols.

DEFINITION. If Σ is a ranked alphabet and Δ is an alphabet, then $\Sigma_*(\Delta)$, the set of trees in Σ_* indexed by Δ, is defined recursively as follows.

1) $\Sigma_0 \cup \Delta \subseteq \Sigma_*(\Delta)$.

2) If $b \in \Sigma_n$, $n > 0$, and $t_1, \ldots, t_n \in \Sigma_*(\Delta)$, then $b[t_1, \ldots, t_n] \in \Sigma_*(\Delta)$.

DEFINITION. An NGSDT $G = (Q, \Sigma, \Delta \cup \Pi, R, Q_0)$ is a *nondeterministic top-down tree transducer* if Δ is a ranked alphabet, $\Delta \cap \Pi = \varnothing$, and

$$R \subseteq \bigcup_{n \geqq 0} (Q \times \Sigma_n) \times \Delta_*(Q \times X_n).$$

We illustrate top-down transducers in the following example.

Example. Let $\Sigma = \{b, c\}$ be a ranked alphabet, in which b has rank 2 and c has rank 0. Let $\Delta = \{f, g, c\}$ be a ranked alphabet in which f and g have rank 2 and d has

rank 0. Then $M=(\{q_0\}, \Sigma, \Delta \cup \Pi, R, \{q_0\})$ is a nondeterministic top-down tree transducer, where

$$R=\{(q_0, b)\to f[(q_0, x_1), (q_0, x_1)], (q_0, b)\to g[(q_0, x_1), (q_0, x_1)], (q_0, c)\to d\}.$$

At each labeled b, M generates either an f or a g and starts two computations on the left subtree. Therefore, if s is a tree in Σ_* such that the path from the root to the leftmost leaf has exactly k nodes, then $G(s)=\{t\in\Delta_*|$ every path from the root of t to a leaf has exactly k nodes$\}$ and $\text{yield}(G(s))=\{d^{2^{k-1}}\}$. Furthermore, $\text{yield}(M(\Sigma_*))=\{d^{2^n}|n\geqq 0\}$.

DEFINITION. An NGSDT G is *deterministic* (a DGSDT) if it has exactly one starting state and for each state q and each input symbol b there is exactly one rule with left side (q, b).

Although top-down tree transducers have been defined to be a subclass of NGSDT's, they may be used to characterize the class of NGSDT's as follows.

PROPOSITION 1. *Let* $A\subseteq\Sigma_*\times\Delta^*$, $\Delta\cap\Pi=\Sigma\cap\Pi=\varnothing$. $A=T(G)$ *for some NGSDT (DGSDT)* G *if and only if* $A=\{\langle s, \text{yield}(t)\rangle|\langle s, t\rangle\in T(M)\}$ *for some nondeterministic (deterministic) top-down tree transducer* M.

Proof. Given an NGSDT $G=(Q, \Sigma, \Delta, R, Q_0)$ with $\Delta\cap\Pi=\varnothing$, let $m=\max\{|v|\,|u\to v\in R\}$. Let $\Gamma=\{d_n|1\leqq n\leqq m\}$ be a set of new symbols such that each d_i has rank i. Let $M=(Q, \Sigma, \Delta\cup\Pi, R', Q_0)$ be a top-down tree transducer, where $R'=\{u\to d_n[B_1, B_2, \ldots, B_n]|u\to B_1\ldots B_n\in R$ and each $B_i\in\Delta\cup Q\times X\}$. It is easy to see that $T(M)=\{\langle s, \text{yield}(t)\rangle|\langle s, t\rangle\in T(M)\}$.

Conversely, let $M=(Q, \Sigma, \Delta\cup\Pi, R, Q_0)$ be a top-down tree transducer, where $\Delta\cap\Pi=\varnothing$. Construct an NGSDT $G=(Q, \Sigma, \Delta, R', Q_0)$ by setting $R'=\{u\to\text{yield}(t)|u\to t\in R\}$. It is clear that $T(G)=\{\langle s, \text{yield}(t)\rangle|\langle s, t\rangle\in T(M)\}$. □

In the last example we constructed a top-down transducer M such that $\text{yield}(M(\Sigma_*))=\{d^{2^n}|n\geqq 0\}$. Since this set is not context-free, we see that top-down transducers, and therefore NGSDT's, can generate non-context-free sets from input sets which are special sets of trees called recognizable sets.

DEFINITION. A set of trees T is *recognizable* if there exists a nondeterministic top-down transducer M such that $R=\{s|M(s)\neq\varnothing\}$.

Recognizable sets have been extensively studied because their properties correspond to those of regular sets of strings [9], [10]. It is known that every recognizable set is a context-free set of strings; furthermore, a set of strings is an e-free context-free language if and only if it is the yield of a recognizable set [11].

To show that sets are context-sensitive, we use the well-known fact that a set is (deterministic) context-sensitive if and only if it is accepted by a nondeterministic (deterministic) multi-tape Turing acceptor within linear space. We assume that the reader is familiar with nondeterministic multi-tape Turing acceptors; since the constructions here are not affected by minor variations in the model of multi-tape Turing acceptor used, we do not define them formally. However, we specify that a nondeterministic multi-tape Turing acceptor M operates within linear space if there is a constant m such that for each input w, every computation of M on w visits at most $m|w|$ tape squares on any tape. In § 3, we use the well-known fact that every deterministic context-sensitive language is accepted by a deterministic one-tape Turing acceptor which operates in linear space and halts in every computation.

2. Relationship to Turing machines. In this section, we show that for any NGSDT G, there is a Turing machine M which accepts $T(G)$ using an amount of workspace which is linear with respect to the length of the input $\langle s, t\rangle$. This is not an

obvious fact, since a machine which carries out an NGSDT must be able to handle "copying": That is, if $(q, b) \to u_1(p_1, x_j)u_2 \ldots u_m(p_m, x_j)u_{m+1}$ is a rule of the NGSDT G, then m independent computations of G occur on the jth subtree of the node labeled b. One way of dealing with them is to create m copies of this subtree; but when this process is repeated at successive nodes, it may result in using an exponential amount of space.

The method used in the construction presented here is the following. To determine whether $\langle s, t\rangle$ is in $T(G)$, M simulates a computation of G as follows. If $(q, b) \to u_1(p_1, x_{i_1})u_2 \ldots u_m(p_m, x_{i_m})u_{m+1}$ is a rule of an NGSDT G, it is applied to a tree $b[s_1, \ldots, s_n]$ by checking u_1 against t, computing on the subtree s_{i_1}, checking u_2 against t, computing on the subtree s_{i_2} etc. The number of rules stored at any one time is never greater than the depth of the input tree, and the total working space is no greater than the length of the input. The actual construction is complicated by the need for M to check all possible computations of G on s if G is nondeterministic.

THEOREM 1. *If G is an NGSDT, then $T(G)$ is a deterministic context-sensitive language.*

Proof. We will construct a deterministic one-tape Turing machine M which accepts $T(G)$ and operates in linear space. Given $\langle s, t\rangle$, M checks whether s is a tree. If so, M tries all possible computations of G on s to determine whether there is one which generates t. The difficulty lies in arranging for M to encode the computations in a form which allows it to investigate them all systematically.

Obtain a new NGSDT G' from G by replacing each rule $u \to v$ by a rule $u \to \{v\}$, where { and } are new symbols. The brackets in a string such as $\{bc\{c\{d\}\}\{ef\}\}$ output by G' indicate which substrings of the output correspond to subcomputations on subtrees of s. Intuitively, M tries all such possible bracketings to find one representing a computation of G producing t. Unfortunately, M can't use this encoding directly, since the size of the bracketed string is not necessarily linear with respect to the size of t. Instead, M obtains an encoding which is linear with respect to the size of t by replacing each sequence of {'s or }'s by a single { or }, respectively. From the rules of G, M will be able to determine where there should be multiple occurrences of { or }. M uses this encoding to keep track of which computations of G it has already checked.

Given input $s \# t$, M tries all possible ways of adding brackets to t and tests each such bracketed string t' to see if it corresponds to a computation of G generating t from s. Generating all possible bracketings can be done in a straightforward manner using linear space. The only difficulty lies in showing that M can determine in linear space whether t' corresponds to a computation of G generating t from s.

M simulates a computation of G by processing each rule from left to right. At any time, certain nodes of s are marked *active*. The active nodes of s form a path leading from the root of s to the lowest active node of s. Each active node n is marked with a rule of G and a state. In addition, if a child of n is active, n has an *active index* k such that the child corresponds to a computation generated from the kth variable in the rule.

Certain brackets of t' are also marked active. If the rule at an active node n has right side $u_1(q_{i_1}, x_{i_1})u_2 \ldots u_m(q_{i_m}, x_{i_m})u_{m+1}$, then each of the m variables in the rule has a corresponding pair of active brackets. If node $n \cdot i_k$ is also active, corresponding to a computation generated from the kth variable of the rule, then the brackets corresponding to the kth variable are also said to correspond to node $n \cdot i_k$.

M can determine which active brackets correspond to which node or variable in a rule as follows. The leftmost { and rightmost } correspond to the root. Assume M has identified the { and } corresponding to an active node n, and these are at squares A and

B, respectively. Suppose the right side of the rule is $u_1(q_{i_1}, x_{i_1})u_2 \ldots u_m(q_{i_m}, x_{i_m})u_{m+1}$. The square C containing the { corresponding to the first variable is A if $u_1 = e$, and the square containing the first active { to the right of A otherwise. Similarly, the square D containing the } corresponding to the last variable is B if u_{m+1} is null, and the square containing the first active } to the left of B otherwise. If no child of n is active, there are exactly m pairs of active brackets from C to D, and each pair corresponds to one variable of the rule. Otherwise, suppose the active index of node n is k. The brackets of the first $k-1$ variables are the first $k-1$ pairs of active brackets starting at C, and the left bracket of the kth variable is the next active { to the right. The brackets of the last $m-k$ variables are the $m-k$ pairs of active brackets ending at D, and the right bracket of the kth variable is the preceding active }. There may be additional active brackets between the brackets corresponding to the kth variable of the rule.

Now, we can describe how M determines whether t' corresponds to a computation of G generating t from s starting in state q_0.

(1) Initially, M marks the root active with state q_0.

(2) (Try next rule). If there is no active node, reject t' as no computation can generate it from s. Otherwise, let n denote the active node furthest from the root. If all rules have been tried at n, mark n inactive and go to (3). Otherwise, mark n with the next rule which has not yet been tried at n. If this rule has the wrong state or label, go to (2).

(3) (Match up indices in the rule with brackets). Let n denote the active node furthest from the root. Using an appropriate algorithm for generating successive ways of matching up brackets with indices of the rule at node n, generate the next such way, making the previous matching brackets inactive and the new ones active. If the last possible way of matching brackets with indices has been tried, go to (2). See whether the output symbols in the rule are the same as the appropriate symbols between the brackets matching the indices of the rule. If not, go to (3). If the right side of the rule contains no indices, mark this node inactive and go to (4) to process the next index of its parent. Otherwise, set the active index to 1. If the first variable in the rule is x_k, mark node $n \cdot k$ active with the appropriate state from the rule, and go to (2).

(4) (Process next index). If no node is active, halt and accept. Otherwise, let n denote the active node furthest from the root and let j denote the index just processed at n. If j is the last index of the rule at n, mark n and the brackets corresponding to the variables of n's rule (but not to n itself) inactive and go to (4). Otherwise, set the active index to $j+1$. If the $(j+1)$st variable in the rule is x_k, mark node $n \cdot k$ active with the appropriate state from the rule, and go to (2).

3. Main theorem. In this section, we show that for every n, if the composition of n top-down tree transductions is applied to a recognizable set, both the resulting set of trees L and the set of strings yield(L) are context-sensitive languages; this answers a question raised by Thatcher [5]. In fact, the family of all languages L and yield(L) obtained from recognizable sets by the composition of tree transductions is properly contained in the family of context-sensitive languages. The theorems are stated in terms of a hierarchy of families of tree languages which has been studied in [7], [8].

DEFINITION. Let D_0 denote the family of recognizable sets. For $n \geqq 0$, let $D_{n+1} = \{M(T) | T \in D_n$ and M is a top-down tree transducer$\}$.

Clearly, D_n is also the family of tree languages obtained from recognizable sets by the composition of n top-down tree transductions.

We begin by stating a definition and a lemma and showing how the lemma may be used to prove the above theorems. The remainder of the section is devoted to the proof of the lemma.

DEFINITION. A top-down tree transducer $M=(Q, \Sigma, \Delta, R, Q_0)$ is *linear* if each variable occurs at most once within the right side of each rule in R.

LEMMA 1. *Let F be a family of tree languages which is closed under linear top-down transductions. If $S_0 \in F$ and M_0 is an NGSDT, then one can construct a tree language $S \in F$ and an NGSDT M such that*

1) $M(S)=M_0(S_0)$ *and*

2) *for every $t \in M(S)$, there exists $s \in S$ such that $t \in M(s)$ and $|s| \leqq 6|t|$.*

Furthermore, if M_0 is deterministic, then so is M.

Using the above lemma, we now prove the main theorem.

THEOREM 2. *For every $n \geqq 0$, D_n and* yield(D_n) *are properly contained in the family of deterministic context-sensitive languages.*

Proof. First we show that D_n and yield(D_n) are contained in the family of deterministic context-sensitive languages. The proof is by induction on n. To begin with, yield (D_0) is the family of e-free context-free languages [11], and D_0 is contained in the family of context-free languages (this is easily shown by techniques in [11]). For some $n \geqq 0$, assume that D_n is contained in the family of deterministic context-sensitive languages. Now, D_n is closed under linear top-down tree transductions [13]. Consider any language L in $D_{n+1} \cup$ yield(D_{n+1}). By Proposition 1, there exist $S \in D_n$ and an NGSDT G such that $G(S)=L$. By Lemma 1, we may assume that for every $w \in L$, there exists $s \in S$ such that $w \in G(s)$ and $|s| \leqq 6|w|$. Since $S \in D_n$, S is a deterministic context-sensitive language (by the induction hypothesis). Therefore, there exists a one-tape deterministic Turing acceptor M which accepts S and operates in linear space. Also, by Theorem 1, $T(G)$ is context-sensitive. Therefore, there is a one-tape deterministic Turing acceptor $M_{T(G)}$ which accepts $T(G)$ and operates in linear space.

Construct a one-tape Turing acceptor M_L to accept L as follows. Suppose M_L is given an input string w. By imitating M_S and $M_{T(G)}$, M_L successively tests all strings u of length at most $6|w|$ to determine whether $u \in D_n$ and $\langle u, w \rangle \in T(G)$. If M_L finds a string u such that $u \in D_n$ and $\langle u, w \rangle \in T(G)$, M_L accepts w. Otherwise, M_L rejects w.

Clearly, L is the language accepted by M_L and M_L operates within linear space. Therefore, L is a deterministic context-sensitive language.

It remains to show that for $n \geqq 0$, D_n and yield$(D)_n$ cannot be equal to the family of deterministic context-sensitive languages. In the case of D_n, this is true for the trivial reason that not every deterministic context-sensitive language is a set of trees. Now, for $n \geqq 0$, $\{\text{yield}(T), \text{yield}(T) \cup \{e\} | T \in D_n\}$ is closed under string homomorphism [13]. But the closure of the family of deterministic context-sensitive languages under string homomorphism is the family of recursively enumerable sets. Therefore, yield(D_n) is not equal to the family of deterministic context-sensitive languages. □

COROLLARY 1. $\cup_{n \geqq 0} D_n$ *and* $\cup_{n \geqq 0}$ yield (D_n) *are properly contained in the family of deterministic context-sensitive languages.*

To motivate the proof of Lemma 1, let us consider a simple example to see how an input tree s can be much bigger than the output produced from it by a GSDT.

Example. Let $\Sigma=\{a, b, c\}$ be a ranked alphabet, where a, b, and c have ranks 0, 1, and 2, respectively. Let $\Delta=\{f, g\}$ be an alphabet. Let $M=(\{p\}, \Sigma, \Delta, R, \{p\})$ be a

GSDT where

$$R = \{(p, c) \to f(p, x_2), (p, b) \to (p, x_1), (p, a) \to g\}.$$

Clearly, $fg \in M(c[c[a, a], b[b[a]]])$. The reason that fg is smaller than the input is that no output was generated from the left subtree $c[a, a]$ or from the b's. In fact, M never even "scanned" the left subtree.

We formalize the above relationships by defining computation trees for GSDT's. Intuitively, a computation tree describes which rules are applied at which nodes of an input tree in a "computation" of a GSDT. Then, given a particular input s and output t and a computation tree which describes how a GSDT obtains t from s, we can determine which nodes of s are never "scanned" and which nodes have no output produced from them in this computation. We will prove the lemma by constructing linear transducers which nondeterministically eliminate nodes which are never scanned and nodes of rank 1 from which no output is produced; enough information is preserved so that a new GSDT can imitate the original one on the modified input.

DEFINITION. Let $M = (K, \Sigma, \Delta, P, K_0)$ be an NGSDT. For a rule $u \to v$, where $v = v_1(p_1, x_{i_1})v_2 \ldots v_m(p_m, x_{i_m})v_{m+1}$ of P, and $v_1, \ldots, v_{m+1} \in \Delta^*$, we say that (p_j, x_{i_j}) is the jth *index* of v, for $1 \leqq j \leqq m$. We obtain a ranked alphabet $\langle P \rangle$ as follows. If $u \to v$ is a rule of P, and v has $m \geqq 0$ occurrences of indices, then $\langle v \rangle$ is a symbol in $\langle P \rangle$ of rank m.

For $q \in K$ and $s \in \Sigma_*$, COMP(q, s), the set of computation trees of M for (q, s) is a subset of $\langle P \rangle_*$ and is defined recursively as follows.

1) For $q \in K$ and $b \in \Sigma_0$, COMP$(q, b) = \{\langle v \rangle | v \in \Delta^*$ and $(q, b) \to v \in P\}$;
2) For $q \in K$, $n > 0$, $b \in \Sigma_n$, and $s_1 \ldots, s_n \in \Sigma_*$,

$$\begin{aligned}\text{COMP}(q, b[s_1, \ldots, s_n]) &= \{\langle v \rangle | v \in \Delta_* \text{ and } (q, b) \to v \in P\} \\ &\quad \cup \{\langle v \rangle[u_1, \ldots, u_m] | m > 0, \langle v \rangle \in \langle P \rangle_m, (q, b) \to v \in P, \text{ and for} \\ &\quad 1 \leqq j \leqq m, \text{ if the } j\text{th index in } v \text{ is } p[x_k] \text{ then } u_j \in \text{COMP}(p, s_k)\}.\end{aligned}$$

If $u \in \text{COMP}(q, s)$, we say that u *is a computation tree of* M *for* (q, s).

Output is associated with a computation tree as follows.

DEFINITION. Let $M = (K, \Sigma, \Delta, P, K_0)$ be a top-down tree transducer. If u is a computation tree of M, then OUTPUT(u) is defined recursively by:

1) For $\langle u \rangle \in \langle P \rangle_0$, OUTPUT$(\langle u \rangle) = u$;
2) For $u = \langle r \rangle[u_1, \ldots, u_m]$, where $m > 0$, $r \in \langle P \rangle_m$, and $u_1, \ldots, u_m \in \langle P \rangle_*$, if $r = v_1(p_1, x_{i_1})v_2 \ldots v_m(p_m, x_{i_m})v_{m+1}$, where $v_1, \ldots, v_{m+1} \in \Delta^*$, then

$$\text{OUTPUT}(u) = v_1\text{OUTPUT}(u_1)v_2\text{OUTPUT}(u_2) \ldots v_m\text{OUTPUT}(u_m)v_{m+1}.$$

Thus, the output from $\langle r \rangle[u_1, \ldots, u_m]$ is determined by r and the output from $u_1, \ldots, u_m$. We illustrate computation trees and their output in the following example.

Example. Let $M = (\{p_0, p_1\}, \Sigma, \Delta, P, \{p_0\})$ be an NGSDT, where $\Delta = \{f, g, h\}$, $\Sigma = \Sigma_0 \cup \Sigma_2$, $\Sigma_0 = \{d\}$, $\Sigma_2 = \{b\}$, and P contains the following rules:

$$(p_0, b) \to f(p_1, x_2)(p_0, x_2), \qquad (p_0, b) \to gh,$$

$$(p_0, d) \to h, \quad \text{and} \quad (p_1, b) \to (p_0, x_1).$$

Clearly, $\langle P \rangle_0 = \{\langle gh \rangle, \langle h \rangle\}$, $\langle P \rangle_1 = \{\langle (p_0, x_1) \rangle\}$, and $\langle P \rangle_2 = \{\langle f(p_1, x_2)(p_0, x_2) \rangle\}$. Notice that $\langle (p_0, x_1) \rangle[\langle h \rangle]$ and $\langle gh \rangle$ are computation trees for $(p_1, b[d, d])$ and $(p_0, b[d, d])$, respectively. Moreover, $\langle f(p_1, x_2)(p_0, x_2) \rangle[\langle p_0, x_1 \rangle][\langle h \rangle], \langle gh \rangle]$ is a computation tree for $(p_0, b[d, b[d, d]])$ with output $fhgh$.

Computation trees and the OUTPUT function characterize the output of an NGSDT as follows.

PROPOSITION 2. *Let* $M = (Q, \Sigma, \Delta, R, Q_0)$ *be an NGSDT. For* $u \in \Delta^*$, $q \in Q$, *and* $s \in \Sigma_*$, $u \in M(q, s)$ *if and only if there exists a computation tree* t *of* M *for* (q, s) *with* $\text{OUTPUT}(t) = u$.

The above proposition is easily proved by induction on the depth of s.

Using computation trees, we define what it means for a node of a computation tree to *scan* a node of a tree s or to *produce output* from a node of s.

DEFINITION. Let $M = (Q, \Sigma, \Delta, R, Q_0)$ be an NGSDT. Let $p \in Q$ and $s \in \Sigma_*$, and let u be a computation tree of M for (p, s). For a node α of s, and a node γ of u, we say that node γ of u *scans* node α of s if either $\alpha = \gamma = e$ or all of the following three conditions hold:

1) for some $k \in \mathscr{P}$ and $\beta \in \mathscr{P}^*$, $\alpha = k \cdot \beta$,
2) for some $j \in \mathscr{P}$ and $\eta \in \mathscr{P}^*$, $\gamma = j \cdot \eta$,
3) if $\langle r \rangle$ is the label of the root of u, then the jth variable in r is x_k and node η of u/j scans node β of s/k.

If node γ of u scans node α of s, the label of γ is $\langle r \rangle$, and r contains at least one symbol of Δ, then node γ of u *generates output* from node α of s.

These definitions are illustrated in the following example.

Example. Let $M = (\{p_0, p_1\}, \Sigma, \Delta, P, \{p_0\})$ be as in the previous example. In the previous example, it was shown that $t = \langle f(p_1, x_2)(p_0, x_2)\rangle[\langle(p_0, x_1)\rangle[\langle h\rangle], \langle gh\rangle]$ is a computation tree for (p_0, s) where $s = b[d, b[d, d]]$. Observe that node e of t scans node e of s and generates output f from node e of s. Node 1 of t scans node 2 of s (but generates no output). Node $1 \cdot 1$ of t scans node $2 \cdot 1$ of s and produces output h. Node 2 of t scans node 2 of s and produces output gh. Observe that node 1 of s is not scanned by any node of t.

If t is a computation tree for (q, s), α is a leaf of s, and node γ of t generates output from α, then at least one symbol of Δ is contributed to $\text{OUTPUT}(t)$ by γ. Furthermore, if nodes γ_1 and γ_2 of t generate output from nodes α_1 and α_2 of s, respectively, and $\alpha_1 \neq \alpha_2$, then $\gamma_1 \neq \gamma_2$. Therefore, if output is produced in u from every leaf of s, the number of leaves in s is at most the length of $\text{OUTPUT}(t)$. Furthermore, if s has m leaves, then s has at most $m - 1$ nodes of rank greater than 1. Thus, if the number of nodes of rank 1 in s is at most $|\text{OUTPUT}(t)|$, and every leaf of s has output produced from it in t, then the number of nodes of s is at most three times $|\text{OUTPUT}(t)|$.

Therefore, our strategy in proving Lemma 1 has two parts. Given $S_0 \in F$ and an NGSDT M_0 such that $M_0(S_0) = T$, we first modify S_0 and M_0 to produce $S_1 \in F$ and NGSDT M_1 such that $M_1(S_1) = T$ and for every $t \in T$, there exists $s \in S_1$ and a computation tree u for $q_0[s]$ such that output is produced in u from every leaf of s and $\text{OUTPUT}(u) = t$. Next, we modify S_1 by deleting certain nodes of rank 1 to obtain $S \in F$ and a top-down transducer M which satisfy Lemma 1.

LEMMA 2. *Let* F *be a family of tree languages closed under linear top-down transductions. Let* $S_0 \in F$, *let* M_0 *be an NGSDT (DGSDT), and let* $T = M_0(S_0)$. *There exist* $S \in F$ *and an NGSDT (DGSDT)* M *with starting state* q_0 *such that*

1) $M(S) = T$ *and*

2) *for every* $t \in T$, *there exist* $s \in S$ *and a computation tree* u *of* M *for* (q_0, s) *such that* $\text{OUTPUT}(u) = t$ *and output is produced in* u *from every leaf of* s.

Proof. Since M_0 is either nondeterministic or is deterministic and has a single starting state, we lose no generality by assuming that M_0 has a single starting state q_0. The new set S is obtained from S_0 by means of a linear top-down transducer which

nondeterministically deletes subtrees from each tree s of S_0. The result is that for each computation tree t of M for $(q, s) \in Q \times S$, there is a tree $s' \in S$ which is obtained by deleting precisely those nodes of s which are not scanned in t.

For each $m > 0$, let

$$\Sigma_m = \{(b, i_1, i_2, \ldots, i_m) | \text{ for some } n > 0,\ 1 \leqq m \leqq n,\ b \in \Gamma_n, \text{ and } 1 \leqq i_1 < i_2 < \cdots < i_m \leqq n\}$$

be a set of new symbols of rank m. Let $\Sigma_0 = \{(b) | b \in \Gamma\}$ be a set of new symbols of rank 0. Thus, $\Sigma = \bigcup_{i=0}^{\infty} \Sigma_i$ is a finite ranked alphabet.

Let $N = (\{q\}, \Gamma, \Sigma \cup \Pi, R_N, \{q\})$ be a nondeterministic linear top-down transducer, where

$$\begin{aligned} R_N = {} & \{(q, b) \to (b) | b \in \Gamma_n,\ n \geqq 0\} \\ & \cup \{(q, b) \to (b, i_1, \ldots, i_m)[(q, x_1), \ldots, (q, x_m)] | b \in \Gamma_n, \\ & \quad n > 0,\ 1 \leqq m \leqq n, \text{ and } 1 \leqq i_1 < i_2 < \ldots < i_m \leqq n\}. \end{aligned}$$

As N reads a tree $s \in \Gamma_*$, it nondeterministically deletes subtrees. If N deletes all but subtrees $i_1, i_2, \ldots, i_m$ of a particular node labeled $b \in \Gamma_n$, then it encodes the numbers of the undeleted subtrees in its output $(b, i_1, i_2, \ldots, i_m)$ and continues to process the undeleted subtrees. Similarly, if N deletes all the subtrees of a node labeled $b \in \Gamma_n$, $n \geqq 0$, then N encodes the deletions in its output (b).

Now, we define another NGSDT M such that $M(N(S_0)) = T$. Let $M = (Q, \Sigma, \Delta, R, \{q_0\})$, where

$$\begin{aligned} R = {} & \{(p, (b)) \to v | v \in \Delta_* \text{ and } (p, b) \to v \in P\} \\ & \cup \{(p, (b, i_1, \ldots, i_m)) \to \\ & u_1(p_1, h(x_{k_1}))u_2 \ldots u_n(p_n, h(x_{k_n}))u_{n+1} | \\ & n \geqq 0,\ m > 0,\ (b, i_1, \ldots, i_m) \in \Sigma,\ u_1, \ldots, u_{n+1} \in \Delta^*, \\ & (p, b) \to u_1(p_1, x_{k_1})u_2 \ldots u_n(p_n, x_{k_n})u_{n+1} \in P, \\ & \text{each } x_{k_j} \text{ is in the set } \{x_{i_1}, \ldots, x_{i_m}\}, \\ & \text{and for } 1 \leqq j \leqq m,\ h(x_{i_j}) = x_j\}. \end{aligned}$$

Note that if M_0 is deterministic, then so is M.

Let $S = N(S_0)$. We assert that M and S satisfy the conditions of the lemma. That $M(S) \subseteq T$ follows from the following claim.

CLAIM. *Let $p \in Q$, $s \in \Gamma_*$, $t \in \Sigma_*$, and $u \in \Delta_*$. If $t \in N(s)$ and $u \in M(p, t)$, then $u \in M_0(p, s)$.*

The proof of the above claim is a straightforward induction on the depth of t and is left to the reader.

Next, we show that M and S satisfy condition 2) of Lemma 2. Since condition 2) implies that $T \subseteq M(S)$, this will complete the proof of the lemma. Condition 2) follows from the following claim by setting $c = 1$.

CLAIM. *Let $c > 0$, $p_1, \ldots, p_c \in Q$, and $s \in \Gamma_*$. If $u_1, \ldots, u_c$ are computation trees of M_0 for $(p_1, s), \ldots, (p_c, s)$, respectively, then there exist $t \in \Sigma_*$ and $v_1, \ldots, v_c \in \langle R \rangle_*$ such that*

1) *$t \in N(q, s)$,*
2) *for $i = 1, 2, \ldots, c$, v_i is a computation tree of M for (p_i, t) with* OUTPUT(v_i) = OUTPUT(u_i), *and*

3) *output is generated from every leaf of t by at least one of $v_1, \ldots, v_c$.*

Proof. The claim is proved by induction on the depth of s. If the depth of s (and therefore of $u_1, \ldots, u_c$) is 1, the statement is easy once one notices that for every rule $(p_i, b) \to z$ of M_0, with $b \in \Gamma_0$ and $z \in \Delta_*$, $(p_i, (b)) \to z$ is a rule of M, and $(q, b) \to (b)$ is a rule of N.

For some $k > 0$, assume that the claim holds for all trees $s \in \Gamma_*$ of depth at most k. Let $s = b[s_1, \ldots, s_n]$ be a tree of depth $k+1$, where $n > 0$, $b \in \Gamma_n$, and $s_1, \ldots, s_n \in \Gamma_*$. Suppose that $u_1, \ldots, u_c$ are computation trees for $(p_1, s), \ldots, (p_c, s)$, respectively.

If the root of s is the only node of s which is scanned in $u_1, \ldots, u_c$, then $u_1, \ldots, u_c \in \langle P_0 \rangle$, and the construction is similar to that of the basis. Otherwise, for $i = 1, 2, \ldots, n$, define

$$D_i = \{(j, \alpha) | 1 \leqq j \leqq c, \alpha \in \mathscr{P}, \text{ and for some } p \in Q,$$
$$u_j/\alpha \text{ is a computation tree for } (p, s_i)\}.$$

For some m, $1 \leqq m \leqq n$, and $1 \leqq i_1 < i_2 < \ldots < i_m \leqq n$, $D_{i_1}, \ldots, D_{i_m}$ are nonempty but $D_j = \varnothing$ for $j \notin \{i_1, \ldots, i_m\}$. We obtain the desired t and $v_1, \ldots, v_c$ by applying the induction hypothesis to each D_{i_j}, deleting the subtrees s_i of s which are not scanned, and modifying the roots of $u_1, \ldots, u_c$ appropriately.

For $1 \leqq j \leqq m$, s_{i_j} has depth at most k. By the induction hypothesis, there exist $t_j \in \Sigma_*$ and a set of computation trees $E_j = \{w_{(d,\alpha)} | (d, \alpha) \in D_{i_j}\} \subseteq \langle R \rangle_*$ such that

1) $t_j \in N(q, s_{i_j})$,
2) if u_i/α is a computation tree for $p[s_{i_j}]$, then $w_{(i,\alpha)}$ is a computation tree for $p[t_j]$, and
3) output is generated from every leaf of t_j by at least one computation tree in E_j.

First, we produce t. Note that N has a rule

$$(q, b) \to (b, i_1, \ldots, i_m)[(q, x_{i_1}), \ldots, (q, x_{i_m})].$$

For $t = (b, i_1, \ldots, i_m)[t_1, \ldots, t_m]$, t is clearly in $N(q, b[s_1, \ldots, s_n]) = N(q, s)$.

Now, we produce $v_1, \ldots, v_c$ from the computation trees in $E_1, \ldots, E_m$ and rules of M. For $i = 1, 2, \ldots, c$, if the root of u_i is labeled $\langle z_i \rangle \in \langle P \rangle_f$, $f \geqq 0$, then M has a rule $(p_i, \langle b, i_1, \ldots, i_m \rangle) \to h_{i_1,\ldots,i_m}(z_i) = z_i'$. If $f = 0$, then $v_i = \langle z_i' \rangle$ is a computation tree for $(p_i, (b, i_1, \ldots, i_m)[t_1, \ldots, t_m]) = (p_i, t)$. If $f > 0$, then $v_i = \langle z_i' \rangle [w_{(i,1)}, w_{(i,2)}, \ldots, w_{(i,f)}]$ is a computation tree for (p_i, t). In each case, $\mathrm{OUTPUT}(v_i) = \mathrm{OUTPUT}(u_i)$.

Each tree in E_j, $1 \leqq j \leqq m$, is used in one of $v_1, \ldots, v_c$. Since output is generated from every leaf of t_j by some tree of E_j, for $1 \leqq j \leqq m$, output is also generated from every leaf of t by at least one of $v_1, \ldots, v_c$. Therefore, t and $v_1, \ldots, v_c$ satisfy the conditions of the claim. □

Now, we use Lemma 2 to obtain our objective, a proof of Lemma 1. First, we restate Lemma 1.

LEMMA 1. *Let F be a family of tree languages which is closed under linear top-down tree transductions. Let $S_0 \in F$ and let $M_0 = (Q, \Sigma, \Delta, R, Q_0)$ be an NGSDT. There exist $S \in F$ and an NGSDT M such that*

1) $M(S) = M_0(S_0)$ *and*
2) *For every $t \in M(S)$, there exists $s \in S$ such that $t \in M(s)$ and $|s| \leqq 6|t|$.*

Furthermore, if M_0 is deterministic, so is M.

Proof. By Lemma 2, we may assume that Q_0 contains a single state and that for every $t \in T$, there exist $s \in S$ and a computation tree u of M_0 for (q_0, s) such that $\mathrm{OUTPUT}(u) = t$ and output is generated in u from every leaf of s.

The new set S is obtained from S_0 by means of a linear transducer N. Given input s, N nondeterministically shortens paths consisting of nodes of rank 1. For each computation tree t of M for (q, s), there is a tree $s' \in S$ such that s' is obtained from s

by deleting exactly those nodes of rank 1 from which no output is produced in t. The transducer N encodes enough information about the deletions for a new transducer M to be able to generate T from S by imitating M_0.

The encoding is motivated by the following observation. Suppose that a tree s has a sequence of nodes $\alpha, \alpha\cdot 1, \ldots, \alpha\cdot 1^{n+1}$, all of rank 1 except possibly $\alpha\cdot 1^{n+1}$. Consider all computations of M_0 which produce no output from $\alpha, \alpha\cdot 1, \ldots, \alpha\cdot 1^n$. For each $p\in Q$, the finite state control of M_0 determines a set C_p such that if M scans node α in state p and produces no output from $\alpha, \alpha\cdot 1, \ldots, \alpha\cdot 1^n$, then it can reach node $\alpha\cdot 1^{n+1}$ in any state in C_p. The linear transducer N nondeterministically computes $A=\{\langle p, q\rangle | p\in Q, q\in C_p\}$ as it reads and deletes $\alpha, \alpha\cdot 1, \ldots, \alpha\cdot 1^n$, and it writes A at the first undeleted node $\alpha\cdot 1^{n+1}$. From A, the new transducer M knows that when it reaches this node in state p, it should apply a rule of M_0 with some state q such that $(p, q)\in A$.

For a set A, let $\mathcal{P}(A)$ denote the power set of A. Let $\Gamma=\Sigma\times\mathcal{P}(Q\times Q)$ be a new ranked alphabet, where for $n\geqq 0$, $\Gamma_n=\Sigma_n\times\mathcal{P}(Q\times Q)$.

The new linear transducer N is defined by $N=(\mathcal{P}(Q\times Q), \Sigma, \Gamma\cup\Pi, P_N, K_0)$, where $K_0=\{(p, p)|p\in Q\}$, and P_N is given as follows.

1) For $B\subseteq Q\times Q$, $n>0$, and $b\in\Sigma_n$, the rule $(B, b)\to\langle b, B\rangle[(K_0, x_1), \ldots, (K_0, x_n)]$ is in P_N;
2) For $B\subseteq Q\times Q$, and $b\in\Sigma_0$, the rule $(B, b)\to\langle b, B\rangle$ is in P_N;
3) For $B\subseteq Q\times Q$, and $b\in\Sigma_1$, the rule $(B, b)\to(C, x_1)$ is in P_N, where $C=\{(p, r)|$ for some $q\in Q$, $(p, q)\in B$ and there is a rule $(q, b)\to(r, x_1)$ in $M_0\}$.

Now, we define a new NGSDT $M=(Q, \Gamma, \Delta, P, \{q_0\})$ by setting $P=\{(p, \langle b, B\rangle)\to v|\langle b, B\rangle\in\Gamma, p\in Q$, and for some $q\in Q$, $(p, q)\in B$ and $(q, b)\to v$ is a rule of $M_0\}$.

Let $S=N(S_0)$. We claim that S and M satisfy conditions 1) and 2) of lemma 1. First, we prove that $M(S)\subseteq T$ by proving the following claim.

CLAIM. *Let* $s\in\Sigma_*$, $t\in\Gamma_*$, $w\in\Delta^*$, $B\subseteq Q\times Q$, *and* $p\in Q$. *If* $t\in N(B, s)$ *and* $w\in M(p, t)$, *then* $\exists q\in Q$ *such that* $(p, q)\in B$ *and* $w\in M_0(q, s)$.

Proof. We use induction on the depth of s. If $s\in\Sigma_0$ and $t\in N(B, s)$, then $t=\langle s, B\rangle$. If $w\in M(p, t)$, then M has a rule $(p, \langle s, B\rangle)\to w$. Therefore, for some q such that $(p, q)\in B$, M_0 has a rule $(q, s)\to w$ and $w\in M_0(q, s)$.

For some $k\geqq 1$, assume that the claim holds for all trees s of depth at most k. Let $s=b[s_1, \ldots, s_n]$ be a tree of depth $k+1$, where $n\geqq 1$, $b\in\Sigma_n$, and $s_1, \ldots, s_n\in\Sigma_*$. Suppose that $t\in N(B, s)$ and $w\in M(p, t)$. We consider two cases, according to how t is obtained from s.

Case 1. $n=1$ and $t\in N(C, s_1)$, where $C=\{(p, r)|\exists q\in Q$ such that $(p, q)\in B$ and $(q, b)\to(r, x_1)$ is a rule of $M_0\}$.

In this case, we apply the induction hypothesis to s_1 to obtain a state $q\in Q$ such that $(p, q)\in C$ and $w\in M_0(q, s_1)$. By definition of C, there exists $r\in Q$ such that $(p, r)\in B$ and $(r, b)\to(q, x_1)$ is a rule of M_0. Thus, $w\in M_0(r, s)$.

Case 2. $t=(b, B)[t_1, \ldots, t_n]$ and for $1\leqq i\leqq n$, $t_i\in N(K_0, s_i)$.

In this case, there exist $w_1, \ldots, w_m\in\Delta_*$, $p_1, \ldots, p_m\in Q$, $i_1, \ldots, i_m\in\mathcal{P}$, and $z_1, \ldots, z_{m+1}\in(\Delta\cup\Pi)^*$ such that $w=z_1w_1z_2w_2\ldots z_mw_mz_{m+1}$, $(p, \langle b, B\rangle)\to z=z_1(p_1, x_{i_1})z_2(p_2, x_{i_2})\ldots z_m(p_m, x_{i_m})z_{m+1}$ is a rule of M, and for $1\leqq j\leqq m$, $w_j\in M(p_j, t_{i_j})$. For some q such that $(p, q)\in B$, M_0 has a rule $(q, b)\to z$. By the induction hypothesis, for $1\leqq j\leqq m$, $w_j\in M_0(p_j, s_{i_j})$. Therefore, $w\in M(q, b[s_1, \ldots, s_n])=M(q, s)$.

Next, we prove that $T\subseteq M(S)$ and that condition 2) of Lemma 1 holds. For any tree s, if s has n nodes of rank greater than 1, m nodes of rank 1, and p leaves, then s has

at least $2n+m$ nodes which are sons of other nodes. Thus, $2n+m \leqq n+m+p$, and $n \leqq p$. Moreover, for each node of s there are at most two symbols of Π. Therefore, if u is a tree, and $m+p \leqq |u|$ then $|s| \leqq 6|u|$. Now, for every tree $t \in T$, there are a tree $s \in S_0$ and a computation tree u of M_0 for $q_0[s]$ such that $\mathrm{OUTPUT}(u) = t$ and output is generated in u from every leaf of s. Thus, we obtain condition 2) by proving the following claim and setting $c = 1$.

CLAIM. *Let $c > 0$, $s \in \Sigma_*$, and $B \subseteq Q \times Q$. If $u_1, \ldots, u_c$ are computation trees of M_0 for $(p_1, s), \ldots, (p_c, s)$, respectively, output is generated from every leaf of s in at least one of $u_1, \ldots, u_c$, and for $1 \leqq i \leqq c$, $(q_i, p_i) \in B$, then there exists $t \in N(B, s)$ such that the number of nodes of rank 1 or 0 in t is at most $\sum_{i=1}^{c} |\mathrm{OUTPUT}(u_i)|$ and for $1 \leqq j \leqq c$, $\mathrm{OUTPUT}(u_i) \in M(q_i, t)$.*

Proof. We use induction on the depth of s. For $s \in \Sigma_0$, the claim is easy, since $(s, B) \in N(B, s)$ and for any rule $(p_i, s) \to z_i$ of M_0, M has a rule $(q_i, \langle s, B \rangle) \to z_i$.

For some $k \geqq 1$, assume that the claim holds for all trees s of depth at most k. Let $B \subseteq Q \times Q$ and $(q_1, p_1), \ldots, (q_c, p_c) \in B$. Let $s = b[s_1, \ldots, s_n] \in \Sigma_*$ be a tree of depth $k+1$, where $n > 0$, $b \in \Sigma_n$, and $s_1, \ldots, s_n \in \Sigma_*$. Suppose that $u_1, \ldots, u_c$ are computation trees of M_0 for $(p_1, s), \ldots, (p_c, s)$, respectively, and output is generated from every leaf of s in at least one of $u_1, \ldots, u_c$. We consider two cases, according to whether or not $n = 1$ and output is produced from the root of s in $u_1, \ldots, u_c$.

Case 1. $n = 1$ and no output is produced from the root of s in $u_1, \ldots, u_c$.

In this case, for $i = 1, 2, \ldots, c$, there exist $z_i \in \langle R \rangle_1$ and $u_1' \in \langle R \rangle_*$ such that $u_i = \langle z_i \rangle [u_i']$. Moreover, for some $r_i \in Q$, $z_i = (r_i, x_1)$, u_i' is a computation tree of M_0 for $r_i[s_1]$, and $\mathrm{OUTPUT}(u_i') = \mathrm{OUTPUT}(u_i)$.

Let $C = \{(p, r) | \exists q \in Q$ such that $(p, q) \in B$ and $(q, b) \to (r, x_1)$ is a rule of $M_0\}$. Clearly, for $1 \leqq i \leqq c$, $(q_i, r_i) \in C$. Since the depth of s_i is k, we apply the induction hypothesis to obtain a tree $t \in N(C, s_1)$ such that the number of nodes of rank 1 or 0 in t is at most $\sum_{i=1}^{c} |\mathrm{OUTPUT}(u_i')| = \sum_{i=1}^{c} |\mathrm{OUTPUT}(u_i)|$ and for $1 \leqq i \leqq c$, $\mathrm{OUTPUT}(u_i) \in M(q_i, t)$. Since N has a rule $(B, b) \to (C, x_1)$, $t \in N(B, s)$, and the claim is satisfied.

Case 2. Either $n > 1$, or $n = 1$ and output is produced from the root of s in at least one of $u_1, \ldots, u_c$.

For $i = 1, 2, \ldots, n$, define $D_i = \{(j, \alpha) | 1 \leqq j \leqq c, \alpha \in \mathcal{P}$, and for some $p \in Q$, u_j/α is a computation tree of M_0 for $(p, s_i)\}$. Since output is produced from every leaf of s in at least one of $u_1, \ldots, u_c$, $D_i \neq \varnothing$ for $1 \leqq i \leqq c$.

For each i, $1 \leqq i \leqq c$, consider D_i. By the induction hypothesis, there exists a tree $t_i \in N(K_0, s_i)$ such that the number of nodes of rank 1 or 0 in t_i is at most $\sum_{(j,\alpha) \in D_i} |\mathrm{OUTPUT}(u_j/\alpha)|$, and for $p \in Q$ and $(j, \alpha) \in D_i$, if u_j/α is a computation tree for (p, s_i), then $\mathrm{OUTPUT}(u_j/\alpha) \in M(p, t_i)$.

Let $t = (b, B)[t_1, \ldots, t_n]$. Since N has a rule $(B, b) \to (b, B)[(K_0, x_1), \ldots, (K_0, x_n)]$ and each $t_i \in N(K_0, s_i)$, $t \in N(B, b[s_1, \ldots, s_n]) = N(B, s)$.

For $i = 1, 2, \ldots, c$, if the root of u_i is labeled $\langle z \rangle$, then M_0 has a rule $(p_i, b) \to z$ and M has a rule $(q_i, \langle b, B \rangle) \to z$. Suppose that $z = z_1(r_1, x_{i_1}) z_2(r_2, x_{i_2}) \ldots z_m(r_m, x_{i_m}) z_{m+1}$, where $r_1, \ldots, r_m \in Q$, $i_1, \ldots, i_m \in \mathcal{P}$, and $z_1, \ldots, z_{m+1} \in (\Delta \cup \Pi)^*$. For $1 \leqq j \leqq m$, let $w_j = \mathrm{OUTPUT}(u_i/j)$. Then $\mathrm{OUTPUT}(u_i) = z_1 w_1 z_2 w_2 \ldots z_m w_m z_{m+1}$ and for $1 \leqq j \leqq m$, $w_j \in M_0(r_j, s_{i_j})$. But, for $1 \leqq j \leqq m$, $(i, j) \in D_{i_j}$ and $w_j = \mathrm{OUTPUT}(u_i/j) \in M(r_j, t_{i_j})$. Therefore, $\mathrm{OUTPUT}(u_i) \in M(q_i, t)$.

For any tree y, let $f(y)$ denote the number of nodes of rank 1 or 0 in y. If $n = 1$, then at least one output symbol is contributed by the root of u_i to $\mathrm{OUTPUT}(u_i)$, for

some i. Thus

$$f(t) = 1 + \sum_{i=1}^{n} f(t_i)$$

$$\leqq 1 + \sum_{i=1}^{n} \sum_{(j,\alpha)\in D_i} |\text{OUTPUT}(u_j/\alpha)| \leqq \sum_{i=1}^{c} |\text{OUTPUT}(u_i)|.$$

If $n > 1$, then

$$f(t) = \sum_{i=1}^{n} f(t_i)$$

$$\leqq \sum_{i=1}^{n} \sum_{(j,\alpha)\in D_i} |\text{OUTPUT}(u_j/\alpha)| \leqq \sum_{i=1}^{c} |\text{OUTPUT}(u_i)|.$$

We conclude by showing that if M_0 is deterministic, then we can omit unnecessary rules of M to obtain a DGSDT M' such that $M'(S) = M(S) = M_0(S_0)$ and M' satisfies condition 2). If M_0 is deterministic, then N never generates an output symbol (b, B) which is not in the set $\Gamma' = \{(b, B) \in \Gamma |$ for each $p \in Q$, there exists at most one $q \in Q$ such that $(p, q) \in B\}$. Therefore, if we obtain M' from M by restricting the input alphabet to Γ' and omitting all rules containing input symbols in $\Gamma - \Gamma'$, we obtain a DGSDT M' such that $M'(S) = M(S) = M_0(S_0)$ and M' satisfies condition 2) of Lemma 1. □

4. The class of bottom-up tree transductions. In this section, we define bottom-up tree transducers and show that for any tree language T obtained from a recognizable set by the application of $n \geqq 0$ bottom-up transductions, both T and yield(T) are context-sensitive.

DEFINITION. A nondeterministic bottom-up tree transducer is a five-tuple $M = (Q, \Sigma, \Delta, R, F)$ where

1) Q is a finite set of *states*,
2) Σ and Δ are finite *input* and *output* ranked alphabets, respectively,
3) $F \subseteq Q$ is a set of *final* or *accepting* states,
4) $\Sigma \cap \Pi = \Delta \cap \Pi = \Delta \cap X = \phi$, and
5) R is a finite set of *transition rules* such that every rule in R is either of the form

$$(b) \to (q, t), \quad \text{where } b \in \Sigma_0,\ q \in Q, \text{ and } t \in \Delta_*,$$

or of the form

$$(b, q_1, \ldots, q_n) \to (q, t), \quad \text{where } n > 0,\ b \in \Sigma_n,\ q, q_1, \ldots, q_n \in Q, \text{ and } t \in \Delta_*(X_n).$$

As in the case of top-down transducers, note that a variable may occur more than once or not at all in the right side of a rule. The output produced by a bottom-up transducer from an input tree is defined as follows.

DEFINITION. Let $M = (Q, \Sigma, \Delta, R, F)$ be a bottom-up transducer. For $q \in Q$, and $t \in \Sigma_*$, $M(q, t)$ is the smallest set of trees in Δ_* such that

1) if $t \in \Sigma_0$, and $(t) \to (q, u) \in R$, then $u \in M(q, t)$;
2) if $t = b[t_1, \ldots, t_n]$, where $n > 0$, $b \in \Sigma_n$, and $t_1, \ldots, t_n \in \Sigma_*$, and (i) $(b, q_1, \ldots, q_n) \to (q, z_1 x_{i_1} z_2 x_{i_2} \ldots z_m x_{i_m} z_{m+1})$ is a rule of R, where $q_1, \ldots, q_n \in Q$, $m \geqq 0$, and $z_1, \ldots, z_{m+1} \in (\Delta \cup \Pi)^*$, and (ii) for $i = 1, 2, \ldots, n$, $u_i \in M(q_i, t_i)$, then $z_1 u_{i_1} z_2 u_{i_2} \ldots z_m u_{i_m} z_{m+1} \in M(q, t)$.

DEFINITION. Let $M = (Q, \Sigma, \Delta, R, F)$ be a bottom-up tree transducer. For $s \in \Sigma_*$, $M(s) = \bigcup_{q \in F} M(q, s)$. For $S \subseteq \Sigma_*$, $M(S) = \bigcup_{s \in S} M(s)$.

Finally, we define a hierarchy of families of tree languages analagous to the top-down hierarchy studied earlier.

DEFINITION. Let U_0 denote the family of recognizable sets. For $n > 0$, define

$$U_n = \{M(T) | T \in U_{n-1} \text{ and } M \text{ is a bottom-up tree transducer}\}.$$

Thus, for $n \geqq 0$, U_n is precisely the family of tree languages which can be obtained from the recognizable sets by the application of n bottom-up tree transductions. Now, in [8], it was shown that for every n, $U_n \subseteq D_n$ and $\text{yield}(U_n) \subseteq \text{yield}(D_n)$. Therefore, from Theorem 2, we obtain the following corollary.

COROLLARY 2. *For every $n \geqq 0$, U_n and* yield(U_n) *are properly contained in the family of deterministic context-sensitive languages.*

Therefore, we have shown that if a set T is obtained from a recognizable set by the application of $n > 0$ tree transductions, either top-down or bottom-up, then both T and yield(T) are deterministic context-sensitive. Moreover, not every deterministic context-sensitive language can be obtained from the recognizable sets in this manner.

Acknowledgment. The author would like to thank R. V. Book for many helpful comments concerning the results of this paper and the two referees for suggesting that Theorem 2 could be strengthened from context-sensitive to deterministic context-sensitive.

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *Properties of syntax-directed translations*, J. Comput. System Sci., 3 (1969), pp. 319–334.

[2] ———, *Syntax directed translations and the push-down assembler*, Ibid., 3 (1969), pp. 37–56.

[3] ———, *Characterizations and extensions of push-down translations*, Math. Systems Theory, 5 (1971), pp. 95–96.

[4] ———, *Translations on a context-free grammar*, Information and Control, 19 (1971), pp. 439–475.

[5] J. THATCHER, *Generalized^2 sequential machines*, J. Comput. System Sci., 4 (1970), pp. 339–367.

[6] W. ROUNDS, *Mappings and grammars on trees*, Math. Systems Theory, 4 (1970), pp. 257–287.

[7] W. OGDEN AND W. ROUNDS, *Compositions of n tree transducers*, Proc. Fourth Annual ACM Symposium on Theory of Computing (1972), pp. 198–206.

[8] B. S. BAKER, *Tree transductions and families of tree languages*, Ph.D. dissertation, Harvard Univ., Cambridge, MA, 1973.

[9] J. W. THATCHER AND J. B. WRIGHT, *Generalized finite automata theory with an application to a decision problem of second order logic*, Math. Systems Theory, 2 (1968), pp. 57–81.

[10] J. DONER, *Tree acceptors and some of their applications*, J. Comput. System Sci., 4 (1970), pp. 406–451.

[11] W. BRAINERD, *Tree generating regular systems*, Information and Control, 14 (1969), pp. 217–231.

[12] S. KOSARAJU, *Context-sensitiveness of translational languages*, Proc. Seventh Annual Princeton Conference on Information Sciences and Systems (1973).

[13] B. BAKER, *Tree transducers and tree languages*, Information and Control, to appear.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0001 $01.00/0

COMPLEXITY OF TASK SEQUENCING WITH DEADLINES, SET-UP TIMES AND CHANGEOVER COSTS*

JOHN BRUNO† AND PETER DOWNEY‡

Abstract. In this paper we consider the problem of sequencing classes of tasks with deadlines in which there is a set-up time or a changeover cost associated with switching from tasks in one class to another. We consider the case of a single machine and our results delineate the borderline between polynomial-solvable and *NP*-complete versions of the problem. This is accomplished by giving polynomial time reductions, pseudo-polynomial time algorithms and polynomial time algorithms for various restricted cases of these problems.

Key words. set-up time, changeover cost, sequencing, scheduling, *NP*-complete, pseudo-polynomial, polynomial time, algorithms

1. Introduction. Suppose that a computer is presented with a collection of tasks (source code programs), each with a known processing time (compilation plus execution), and each with a fixed completion deadline. Each task requires the presence of a particular compiler in memory. If the proper compiler for a task is resident, the task may be instantly started; otherwise the contents of memory are abandoned and a *set-up time* is suffered to bring the proper compiler to memory. This activity is called a *changeover.* Only one compiler can be resident in memory at a time. Thus it may be advisable to contiguously schedule several tasks requiring the same compiler's presence.

The question is: can the given collection of tasks, each demanding a compiler, a processing time and a deadline, be sequenced so as to meet the given deadlines? We examine the complexity of any algorithm which answers this question for arbitrary collections of tasks. We will show that the problem of deciding feasibility on one processor is *NP*-complete. It is *NP*-complete even for some very restricted special cases.

We assume that the reader is familiar with *NP*-completeness; the reference [1] gives a definition and examples, and [2] relates complexity to sequencing problems.

A closely related set of problems involves a single production line manufacturing various sized lots of several different products. Each lot has a deadline for completion, and there is a *changeover cost* associated with switching the line from the production of one product to another, even though there is no time lost in the changeover. The problem is to minimize the total changeover cost (feasibility is trivial to decide).

We demonstrate below that this problem, even for unit changeover costs, is *NP*-complete.

Our results delineate the borderline between polynomial-solvable and *NP*-complete problems. We do this by giving polynomial time reductions, pseudo-polynomial time algorithms and polynomial time algorithms for various restricted cases of the above problems. Reductions are used to show *NP*-completeness and thus give evidence that certain problems cannot be solved in polynomial time; pseudo-polynomial time algorithms and polynomial time algorithms are provided to give evidence that certain *NP*-completeness results cannot be strengthened. The results are summarized in Fig. 5 and Table 1.

* Received by the editors November 10, 1976, and in final revised form January 6, 1978. This work was supported by the National Science Foundation under Grant MCS 75-22557.

† Department of Electrical Engineering and Computer Science and the Computer Systems Laboratory, University of California, Santa Barbara, California 93106.

‡ Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802.

2. The problems. Suppose we are given r pairwise disjoint classes of tasks $C_1, \cdots, C_r$. Each class C_i has associated with it a special *set-up task* S_i with *set-up time* $\tau(S_i)$ and *set-up cost* (*changeover cost*) $c(S_i)$.[1] By convention S_i is not itself an element of C_i.

Every task T in $C = \bigcup_i C_i$ has a given integer *processing time* $\tau(T)$ and a deadline $d(T)$. These tasks are to be scheduled on m identical processors, though only the case $m = 1$ will be dealt with here.

In this paper we shall assume that all schedules are nonpreemptive.

On any processor, every schedule must satisfy the following *sequencing constraint*: every task must immediately follow another task of its class or the set-up task for its class. A set-up task may be scheduled at any time a processor is idle, and may be scheduled as often as necessary.

We shall assume that "schedule" means "schedule satisfying the sequencing constraints."

We now define the two problems whose complexity we wish to explore.

DEFINITION 1. The *Feasibility Problem*[2] is: given a collection of classes $(C_1, \cdots, C_r)$, does there exist a schedule for all the tasks in C (and all the necessary set-up tasks) such that all the non-set-up tasks finish before their deadlines?

DEFINITION 2. The *Schedule Cost Problem* is: given a collection of classes $(C_1, \cdots, C_r)$ and a nonnegative integer K, does there exist a schedule for all the tasks in C such that all the non-set-up tasks finish before their deadlines and the total changeover cost is less than or equal to K?

Example. Let $r = 2$ where the tasks in each class are described as follows:

$$
\begin{aligned}
&C_1 = \{T_{11}, T_{12}, T_{13}\}, && \tau(S_1) = 2, && c(S_1) = 1,\\
&d(T_{11}) = 3, && \tau(T_{11}) = 1,\\
&d(T_{12}) = 15, && \tau(T_{12}) = 2,\\
&d(T_{13}) = 12, && \tau(T_{13}) = 2,\\
&C_2 = \{T_{21}, T_{22}\}, && \tau(S_2) = 1, && c(S_2) = 1,\\
&d(T_{21}) = 10, && \tau(T_{21}) = 3,\\
&d(T_{22}) = 15, && \tau(T_{22}) = 2.
\end{aligned}
$$

We depict a feasible schedule in Fig. 1 (which is also minimum in cost, since only one feasible schedule exists). Note that Class 1 tasks must be split into two segments in order for T_{21} to meet its deadline.

Given a schedule S, we define a *segment* of S to be a maximal consecutive sequence of tasks all from the same class beginning with a set-up task and followed by non-set-up tasks. The schedule of Figure 1 has three segments.

In this paper we confine our attention to the above problems for a single processor ($m = 1$). For two processors ($m = 2$) all these problems are hard; it is trivial to reduce the Partition Problem [7] to even degenerate cases of these problems.

---

[1] We shall not be simultaneously interested in set-up time and set-up cost; this formulation is given for generality.

[2] Closely related to this problem is the *Schedule Length Problem*: given a collection of classes $(C_1, \cdots, C_r)$, and a nonnegative integer K, does there exist a schedule such that all non-set-up tasks make their deadlines and the maximum finishing time is $\leqq K$? This problem is easily seen to be polynomially equivalent to the Feasibility Problem, and so will not be discussed further.

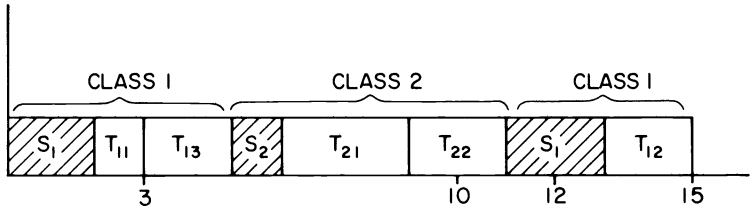


FIG. 1. *A schedule.*

We have also assumed that changeover costs and set-up times depend only upon the class scheduled after each changeover. If costs or times depend upon both the classes processed before and after the changeover, the costs or times are called *sequence-dependent* [5]. When sequence-dependent costs (times) are allowed, it is trivial to reduce the Traveling Salesman Problem to even special cases of our problems.

In § 4 below we address ourselves to the case of $m = 1$ and non-sequence-dependent costs and times. Depending upon the number of distinct deadlines involved, these problems can be easy (polynomial) or hard (*NP*-complete).

3. History. Several authors have examined the case of changeover *costs* with deadlines, attempting to design efficient algorithms for minimum cost schedules. Glassey [8] considered the special case of unit changeover costs, i.e., the problem of minimizing the number of changeovers within deadlines.

Mitsumori [10] improved upon Glassey's branch-and-bound algorithm and extended it to handle non-sequence-dependent changeover costs. Driscoll and Emmons [5], [4] have given dynamic programming algorithms for the most general case of sequence-dependent changeover costs with deadlines.

Turning to the problem of minimizing total set-up *time* within deadlines, this is seen to be identical to the problem of minimizing schedule length within deadlines. Driscoll [3] gives a branch-and-bound algorithm for the general case of sequence-dependent set-up times. If all set-up times are identical and nonzero, this is the same as the problem of minimizing the number of changeovers.

4. *NP*-Complete sequencing problems: Reductions. In this section we reduce the Knapsack Problem to the Feasibility Problem, showing it to be *NP*-hard (that it is in *NP* is trivial). For each instance of the Knapsack Problem, we construct a sequencing problem instance with uniform set-up times, three tasks per class and three distinct deadlines, and such that the instance is feasible if and only if the Knapsack instance has a solution.

In this section we also consider the Schedule Cost Problem. Reductions are provided showing various versions of these problems to be *NP*-complete. These results are summarized in Table 1.

The following problem, called the Knapsack Problem, is known to be complete in *NP* [7].

Knapsack Problem. Let $\tau_1, \cdots, \tau_n, b$ be positive integers. Does there exist a subset $I \subseteq \{1, \cdots, n\}$ such that

$$\sum_I \tau_i = b?$$

We denote an instance of the problem by Kn $(\{\tau_1, \cdots, \tau_n\}, b)$.

TABLE 1
NP-complete problems.

Problem	Set-up times	Changeover costs	Processing times	Tasks per class	Number of distinct deadlines	Results
Feasibility	unit	zero	—	3	3	Cor. 1
Schedule Cost	zero	unit	—	3	2†	Cor. 2
Feasibility	—	zero	—	2	2	Thm. 3
Schedule Cost	zero	—	—	2	1†	Thm. 4

† Number of noninfinite deadlines.

Next we exhibit a linear-time construction of an instance P of the Feasibility Problem from an instance Kn of the Knapsack Problem.

Construction. Let Kn $(\{\tau_1, \cdots, \tau_n\}, b)$ with $\tau_0 = \sum_{i=1}^{n} \tau_i > b$ be an instance of the Knapsack Problem.

Construct an instance P of the sequencing problem with $n+2$ classes $C_0, C_1, \cdots, C_n, C_{n+1}$. Each class contains three tasks: T_{i1} with deadline d_1, T_{i2} with deadline d_2 and T_{i3} with deadline d_3.

class C_i	$\tau(S_i)$	$\tau(T_{i1})$	$\tau(T_{i2})$	$\tau(T_{i3})$
C_0	1	1	τ_0	$H\tau_0$
C_1	1	1	τ_1	$H\tau_1$
⋮	⋮	⋮	⋮	⋮
C_n	1	1	τ_n	$H\tau_n$
C_{n+1}	1	1	τ_0	$H\tau_0$

Let

$$d_1 = 2(n+2)+b, \quad d_2 = 3n+5+3\tau_0+2H\tau_0-Hb \quad \text{and} \quad d_3 = 3n+5+3\tau_0+3H\tau_0$$

where $H = \max(\tau_0, n+2)$. Clearly $d_3 > d_2 > d_1$.

Call a task with a deadline d_i a *d_i-task.*

Consider any feasible schedule for P. It consists of a succession of segments. We classify segments according to the tasks they contain. For example, a segment which consists of a d_1-task and a d_3-task is called a *d_1d_3-segment.*

LEMMA 1. *Let S be a feasible schedule for P. Then no d_3-task finishes at or before deadline d_1 in S. Moreover, neither of the tasks T_{02} or T_{n+12} finishes at or before d_1.*

Proof. Assume that some d_3-task, say T, finishes on or before deadline d_1. By the choice of H we have $\tau(T) \geqq \tau_0$. Also appearing before deadline d_1 we must have $n+2$ d_1-tasks and $n+2$ corresponding set-up tasks. Thus the total processing requirement before deadline d_1 is as great as $2(n+2)+\tau_0 > 2(n+2)+b = d_1$ (since $\tau_0 > b$), a contradiction.

Similarly, tasks T_{02} and T_{n+12} have processing times equal to τ_0 and cannot finish before d_1 by the same reasoning. □

LEMMA 2. *Let S be a feasible schedule for P. Then S cannot contain more than $2n+3$ segments.*

Proof. Suppose S contains more than $2n+3$ segments and consequently at least $2n+4$ set-up tasks appearing before d_3. Also, before d_3 we must find all d_1-tasks,

d_2-tasks and d_3-tasks. Thus the total processing requirement before d_3 (including the set-up tasks) is at least $(2n+4)+(n+2)+3\tau_0+3H\tau_0 = d_3+1$, a contradiction. □

LEMMA 3. *Let S be a feasible schedule for P. Then in S there is exactly one $d_1d_2d_3$-segment; there are no d_1d_3-segments and no d_2-segments.*

Proof. It is easy to show, using Lemma 1, that there can be at most one d_1d_3-segment or $d_1d_2d_3$-segment in S. Assume there is a d_1d_3-segment. There are $n+2$ segments containing d_1-tasks. There must be an additional $n+1$ segments containing d_3-tasks. In addition, there must be a d_2-segment which contains the d_2-task missing from the d_1d_3-segment. This is a total of $2n+4$ segments, in contradiction to Lemma 2. Consequently, there are no d_1d_3-segments in S.

A similar argument shows that the existence of a d_2-segment contradicts Lemma 2 and therefore S cannot contain any d_2-segments.

Assume there is no $d_1d_2d_3$-segment in S. As before, there must be $n+2$ segments containing d_1-tasks and an additional $n+2$ segments containing d_3-tasks, a contradiction. Accordingly, S contains exactly one $d_1d_2d_3$-segment. □

LEMMA 4. *Given a feasible schedule S, there exists a feasible schedule S' such that all d_1-segments appear before all d_1d_2-segments, all d_1d_2-segments appear before the $d_1d_2d_3$-segment, the $d_1d_2d_3$-segment appears before all the d_2d_3-segments and all the d_2d_3-segments appear before all the d_3-segments.* (cf. Fig. 2.)

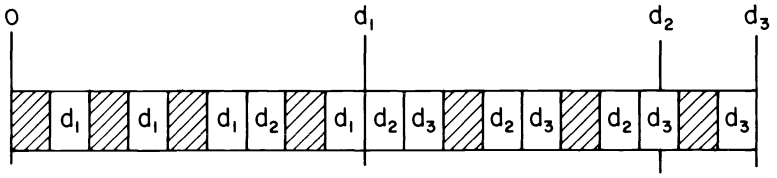


FIG. 2. *A normal schedule.*

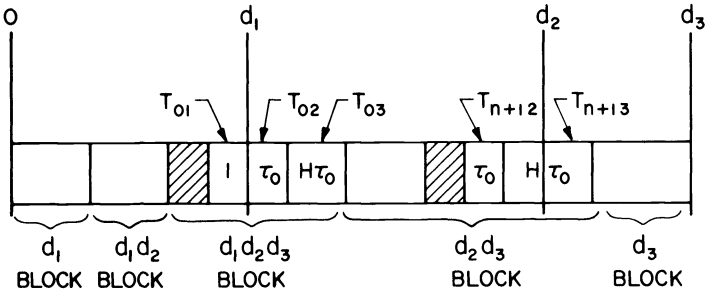


FIG. 3. *A feasible schedule.*

Proof. Starting with the schedule S, we move all the d_1-segments to the beginning of the schedule. It is easy to see that this is possible. Then we move all the d_1d_2-segments to positions immediately following the d_1-segments, etc. Each separate interchange of segments leaves feasibility invariant. □

When a schedule is in the normal form of Lemma 4, we will refer to the contiguous group of d_1-segments as the *d_1-block*, the contiguous group of d_1d_2-segments as the *d_1d_2-block*, etc.

THEOREM 1. *The sequencing problem instance P is feasible if and only if* Kn $(\{\tau_1, \cdots, \tau_n\}, b)$, *with $\tau_0 = \sum_{i=1}^{n} \tau_i > b$, has a solution.*

Proof. ($\Leftarrow$) Suppose Kn $(\{\tau_1, \cdots, \tau_n\}, b)$ has a solution $I \subseteq \{1, \cdots, n\}$, that is, $\sum_{i \in I} \tau_i = b$. Define $\bar{I} = \{1, \cdots, n\} - I$. Consider the schedule illustrated in Fig. 3. The d_2-tasks in the d_1d_2-block are the T_{i2}'s with $i \in I$. The $d_1d_2d_3$-segment contains $T_{01}T_{02}$ and T_{03}. The d_3-tasks in the d_2d_3-block are the T_{i3}'s with $i \in \bar{I}$ and T_{n+13}. The last d_2d_3-segment in the d_2d_3-block consists of tasks T_{n+12} and T_{n+13}. The tasks T_{i3} for $i \in I$ are in the rear d_3-block. This accounts for all the tasks. We consider each deadline in turn showing feasibility of the schedule illustrated in Fig. 3.

Deadline d_1. All the d_1-tasks, the d_2-tasks in the d_1d_2-block, and $n+2$ set-up tasks must be processed before d_1. The total processing time is

$$n+2+\sum_I \tau_i + n+2 = d_1.$$

Deadline d_2. All the d_1-tasks, all the d_2-tasks, no more than $2n+3$ set-up tasks (cf. Lemma 2) and all d_3-tasks T_{i3} with $i \in \bar{I}$ and T_{03} must be processed before deadline d_2. The total processing time does not exceed $(n+2)+3\tau_0+2n+3+2H\tau_0 - Hb = d_2$.

Deadline d_3. The total processing time to d_3 consists of $2n+3$ set-up tasks and all the other tasks. This total is exactly d_3.

It follows that the illustrated schedule is feasible.

($\Rightarrow$) Suppose there is a feasible schedule S for P and that S is in normal form (Lemma 4 and Fig. 2). Let us call the d_2-tasks in the d_1d_2-block the *chosen* tasks: I enumerates their indices.

Consider the sum of the chosen task times $\sum_I \tau_i = L$. If $L = b$ then the indices in I provide a solution to Kn $(\{\tau_1, \cdots, \tau_n\}. b)$ since, by Lemma 1, the indices 0 and $n+1$ cannot be in I.

It remains to show that $L = b$ is the only possibility. We treat two cases.

Case 1. $L > b$. Summing the total processing prior to d_1 we get $2(n+2)+L > d_1$, a contradiction. The first term on the left-hand side is the contribution from the set-up tasks and the d_1-tasks, the second is due to the d_2-tasks.

Case 2. $L = b - \varepsilon$ where ε is a positive integer. Summing the processing required prior to d_2 we get a lower bound D where

$$D = (n+2)+(n+2)+3\tau_0+H(3\tau_0-b+\varepsilon)-H\tau_0.$$

The first three terms correspond to set-up tasks, d_1-tasks and d_2-tasks, respectively. The last two terms correspond to d_3-tasks which appear in segments with a d_2-task less the maximum such d_3-task which can extend beyond d_2. Computing $D - d_2$ we get

$$D - d_2 = H\varepsilon - (n+1) > 0$$

since $H = \max(\tau_0, n+2)$. Thus the deadline d_2 is not met, a contradiction. □

COROLLARY 1. *The Feasibility Problem with unit set-up times, three tasks per class and three distinct deadlines is NP-complete.*

Next we study the complexity of minimizing the *number* of changeovers when all set-up times are zero. A construction similar to the one used above proves that the Schedule Cost Problem for unit changeover costs, two noninfinite deadlines and zero set-up times is *NP*-complete.

Construction. Let Kn $(\{\tau_1, \cdots, \tau_n\}, b)$ with $\tau_0 = \sum_{i=1}^n \tau_i > b$ be an instance of the Knapsack Problem. Construct an instance of P of the sequencing problem with $n+2$ classes $C_0, C_1, \cdots, C_n, C_{n+1}$. Each class contains three tasks: T_{i1} with deadline d_1, T_{i2} with deadline d_2 and T_{i3} without a deadline, i.e., $d_3 = \infty$.

class C_i	$\tau(S_i)$	$c(S_i)$	$\tau(T_{i1})$	$\tau(T_{i2})$	$\tau(T_{i3})$
C_0	0	1	1	τ_0	$H\tau_0$
C_1	0	1	1	τ_1	$H\tau_1$
⋮	⋮	⋮	⋮	⋮	⋮
C_n	0	1	1	τ_n	$H\tau_n$
C_{n+1}	0	1	1	τ_0	$H\tau_0$

Let

$$d_1 = n+2+b, \quad d_2 = n+2+3\tau_0+2H\tau_0-Hb \quad \text{and} \quad d_3 = \infty$$

where $H = \tau_0$. K, the upper bound on the number of changeovers, is equal to $2n+3$.

THEOREM 2. *The sequencing problem instance P has a feasible solution with total changeover cost less than or equal to K if and only if* Kn $(\{\tau_1, \cdots, \tau_n\}, b)$, *with* $\tau_0 = \sum_{i=1}^{n} \tau_i > b$, *has a solution.*

We omit the proof of Theorem 2 since it follows the same pattern as Theorem 1 with the same series of supporting lemmas being true if the total changeover cost does not exceed K. The role of d_3 in the previous construction is now played by K.

COROLLARY 2. *The Schedule Cost Problem for unit changeover costs, three tasks per class, two noninfinite deadlines and zero set-up times is NP-complete.*

Corollaries 1 and 2 account for the first two rows of Table 1. In the next two rows of Table 1 we allow unrestricted set-up times and changeover costs, respectively. We find that we can place more severe restrictions on the number of tasks and still show that the Feasibility and Schedule Cost problems are *NP*-complete.

THEOREM 3. *The Feasibility Problem for two tasks per class and two distinct deadlines is NP-complete.*

Proof. We reduce the Knapsack Problem to this sequencing problem. The technique is similar to the one used in the proof of Theorem 1.

Let Kn $(\{\tau_1, \cdots, \tau_n\}, b)$ be an instance of the Knapsack Problem, with $\tau_0 = \sum_{i=1}^{n} \tau_i > b$. Construct a sequencing problem P with $n+1$ classes $C_0, C_1, \cdots, C_n$. Each class contains two tasks: one with deadline d_1 and the other with deadline d_2.

class C_i	$\tau(S_i)$	$\tau(T_{i1})$	$\tau(T_{i2})$
C_0	τ_0	1	τ_0
C_1	τ_1	1	τ_1
⋮	⋮	⋮	⋮
C_n	τ_n	1	τ_n

Let

$$d_1 = (n+1)+2\tau_0+b \quad \text{and} \quad d_2 = (n+1)+5\tau_0-b.$$

Clearly $d_1 < d_2$.

We will merely sketch the remainder of the proof, leaving the details to the reader. We need two easy lemmas.

LEMMA 5. *If S is a feasible schedule, then there is a* normalized *feasible schedule S' starting with a d_1-block, followed by a d_1d_2-block and concluding with a d_2-block.*

LEMMA 6. *In any normalized feasible schedule S, T_{02} must be the last task scheduled in the d_1d_2-block.*

We now make the

CLAIM. *The problem instance P is feasible if and only if* Kn $(\{\tau_1, \cdots, \tau_n\}, b)$ *has a solution.*

Proof of claim. If Kn has a solution I, then P has a feasible schedule with the d_2-tasks indexed by $I \cup \{0\}$ chosen for the d_1d_2-block.

Conversely, suppose P has a normalized feasible schedule (Lemma 5). Let I be the indices of the d_2-tasks chosen for the d_1d_2-block. Then $0 \in I$ by Lemma 6. We reason that $I' = I - \{0\}$ solves the Knapsack instance Kn. For if $\sum_{I'} \tau_i > b$ then the finishing time of the d_1-tasks is greater than d_1, a contradiction. In the case that $\sum_{I'} \tau_i < b$, then the finishing time of all the d_2-tasks is greater than d_2, a contradiction. □

Finally, we turn to the Schedule Cost Problem with the restriction to unit changeover costs removed.

THEOREM 4. *The Schedule Cost Problem for two tasks per class, one noninfinite deadline and zero set-up times is NP-complete.*

We shall not include a proof of Theorem 4 as it bears the same relationship to Theorem 3 as Corollary 2 bears to Corollary 1. However, we give the underlying construction, viz.,

class C_i	$\tau(S_i)$	$c(S_i)$	$\tau(T_{i1})$	$\tau(T_{i2})$
C_0	0	τ_0	1	τ_0
C_1	0	τ_1	1	τ_1
⋮	⋮	⋮	⋮	⋮
C_n	0	τ_n	1	τ_n

Let $d_1 = n + 1 + b$, $d_2 = \infty$ and K, the upper bound on the changeover cost, equals $3\tau_0 - b$.

5. Polynomial time algorithms. We have demonstrated that even very restrictive cases of sequencing problems with deadlines and set-up times/costs are *NP*-complete. If we restrict these problems in any way we can obtain polynomial time algorithms.

For example, the Feasibility Problem for equal set-up times and two deadlines and the Schedule Cost Problem with equal changeover costs and one noninfinite deadline can be solved efficiently. In these cases a greedy algorithm [9] will work. The basic idea is to form as many d_1d_2-segments as possible thereby causing the fewest additional d_2-segments.

The following theorem gives a class of problems which have polynomial time algorithms. The theorem covers restricted versions of the problems in Table 1.

THEOREM 5. *Let d be a given positive integer. Then the Feasibility Problem for equal set-up times, no more than two tasks per class and d distinct deadlines and the Schedule Cost Problem with equal changeover costs, no more than two tasks per class and d noninfinite deadlines can be solved in polynomial time.*

Proof. We can assume, without loss of generality, that each task within a class has a distinct deadline. This follows from the observation that if in a feasible schedule the d_i-tasks of a particular class (d_i is a deadline) appear in different segments, then they may be moved to the latest segment containing one of these d_i-tasks and merged into a single task without affecting feasibility and without increasing the schedule cost.

The observation behind our algorithm goes as follows. We call a class a d_id_j-*class* (where $d_i < d_j$) if it contains a d_i-task and a d_j-task. Order the d_id_j-classes in nondecreasing order of the processing times of the d_j-tasks. Let $C_1, C_2, \cdots, C_k$ be the

ordered list of d_id_j-classes. We claim that we need not consider schedules in which C_i uses two set-ups (changeovers), C_j uses one set-up (changeover) and $j > i$. An interchange argument can be used to show this. Consequently, we need only determine the *index* i^* such that each $C_1, \cdots, C_{i^*}$ uses one setup (changeover) and each $C_{i^*+1}, \cdots, C_k$ uses two set-ups (changeovers). Since there is a fixed number (d) of deadlines we only have to consider a fixed number $\left(\text{less than or equal to } \binom{d}{2}\right)$ of d_id_j-classes. Once we have decided on the segments we can arrange to place all the same type of sements into a single block (this is easy to show) and choose the sement which goes last in each block. It remains to pick an order for the blocks and test feasibility. This leads to a polynomial-time algorithm, because the number of distinct deadlines is d, the number of blocks bounded by $d + \binom{d}{2}$ and the number of arrangements of these blocks is bounded by $\left[d + \binom{d}{2}\right]!$. The choices which depend on the number of classes are the choice of i^* and the choice of the segment which goes last in a block; but the number of these choices is bounded by a polynomial in r whose degree is easily no larger than $\binom{d}{2} + d$.

By suitably arranging the i^* choices this algorithm solves both the Feasibility and the Schedule Cost Problems. □

6. Pseudo-polynomial time algorithms. Let I be an instance of a problem and let MAX[I] denote the magnitude of the largest number occurring in I. Let LENGTH[I] denote the length of the string which encodes I.

The restriction we place on the encoding of problem instances requires that numbers be represented in binary. A *pseudo-polynomial time algorithm* is an algorithm that runs in time bounded by a polynomial in the two variables LENGTH[I] and MAX[I].

Our aim in this section is to show that there exist pseudo-polynomial time algorithms for the problems presented in Table 1 thereby giving evidence that none of these problems are strongly *NP*-complete [6]. The existence of pseudo-polynomial time algorithms means that if a polynomial time reduction exists in which processing times, set-up times and changeover costs are encoded in unary then *deterministic polynomial time* equals *nondeterministic polynomial time*, an unlikely situation! The reader is referred to [6] for a thorough discussion of these issues.

THEOREM 6. *Let d be a given positive integer. There exist pseudo-polynomial time algorithms for the Feasibility Problem and the Schedule Cost Problem with d deadlines.*

Note that Theorem 6 covers a much wider class of problems than those presented in Table 1.

Proof. For simplicity we restrict ourselves to the Feasibility Problem with three distinct deadlines, $d_1 < d_2 < d_3$. It is conceptually easy to extend the proof to handle any fixed number of deadlines. Without loss of generality, we assume that each task within a class has a distinct deadline and that within a class the d_1 task precedes the d_2 task and the d_2 task precedes the d_3 task.

We specify an algorithm which has some nondeterministic steps. Our goal is to show that if the size of the processing times and the set-up times is bounded above by a polynomial in the number of classes then the algorithm need only consider the feasibility of a number of cases bounded by a polynomial in the number of classes. This conclusion relies on the fact that there is a fixed number of deadlines.

Step 1. Pick the class C which forms the latest segment to contain a d_1-task. In addition, choose from the remaining tasks in C those which will accompany the d_1-task in this segment. Consider the remaining tasks (if any) to form a new class.

Step 2. Pick the class C' which forms the latest segment to contain a d_2-task. If C' has a d_3-task then it should accompany the d_2-task in this segment. (It should be recognized that one possible choice is to have class C form the single segment which contains both the last d_1-task *and* the last d_2-task!)

Comment. We refer to Fig. 4, which shows the structure of the partial schedule we have up to this point. Interval t_1 may contain any type of segment, interval t_2 contains d_2-, d_3-, or d_2d_3-segments, and interval t_3 contains d_3-segments only. The rest of our algorithm decides whether the remaining tasks can be added to the partial schedule, determined by the choices in Steps 1 and 2, so as to construct a feasible schedule. Let $C_1, \cdots, C_s$ denote the classes of tasks which remain after Steps 1 and 2. Note that $s \leqq r$, the original number of classes.

Step 3. $L_0 \leftarrow \{\langle 0, 0, 0\rangle\}$; $k \leftarrow 1$.

Comment. L_k denotes a set of triples where each triple $\langle t_1, t_2, t_3\rangle$ represents a possible set of durations (see Fig. 4) using all the tasks in classes $C_1, \cdots, C_k$. We require that $t_1 + a \leqq d_1$; $t_1 + a + b + t_2 + a' \leqq d_2$; and $t_1 + a + b + t_2 + a' + b' + t_3 \leqq d_3$.

The set L_k is constructed from set L_{k-1} and class C_k. The information regarding all possible segmentations of the tasks in classes $C_1, \cdots, C_{k-1}$ is contained in the triples belonging to L_{k-1}. This economy leads to a pseudo-polynomial time algorithm.

Step 4. Consider class C_k. For all divisions of C_k into segments and for all $\langle t_1, t_2, t_3\rangle$ in L_{k-1}, add the duration of these new segments to t_1, t_2 and t_3 in all possible ways to form new triples $\langle t_1', t_2', t_3'\rangle$ which belong to list L_k.

If $k = s$ then go to Step 5. Otherwise $k \leftarrow k + 1$ and repeat Step 4.

Step 5. If $L_k \neq \phi$ then there is a feasible schedule. *Stop.*

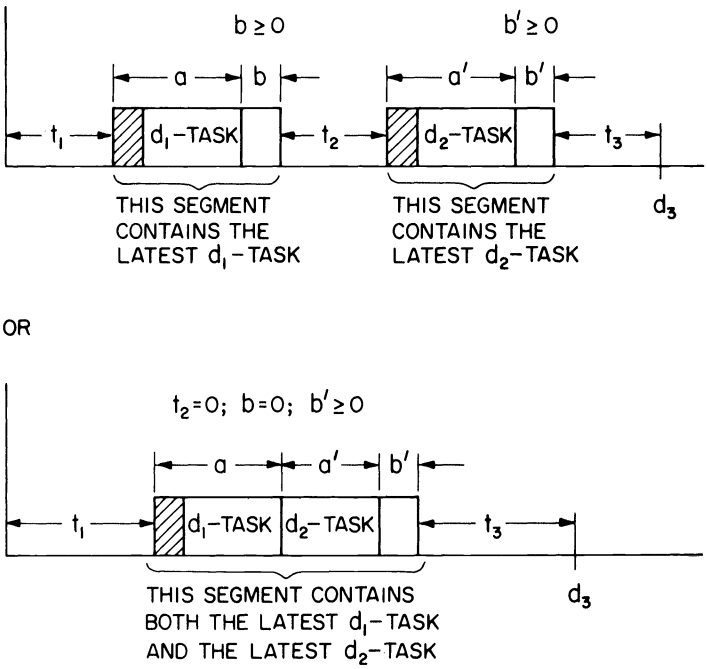
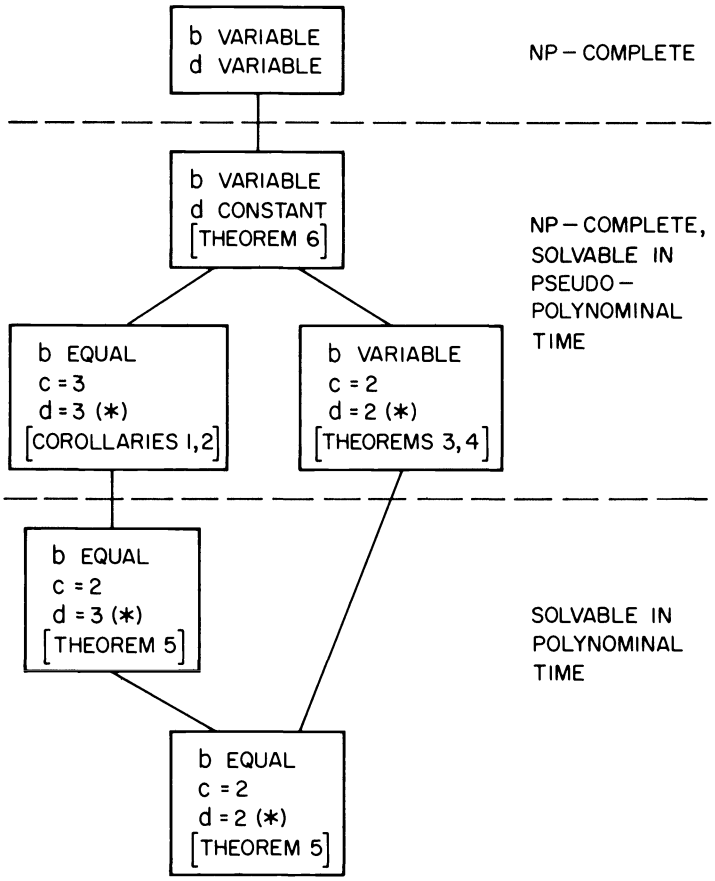


FIG. 4. *Partial schedules.*

To analyze the time complexity of the above algorithm we observe that the nondeterminism in Steps 1 and 2 results in a multiplicative factor of $O(r^2)$. Step 4 is repeated no more than r times. Let M be the sum of all the processing times and set-up times. The list L_k can have no more than $O(M^3)$ entries and we can arrange (if we do not allow duplicate entries) to construct list L_k from list L_{k-1} in $O(M^3)$ time. This leads to a time complexity of $O(r^3M^3)$. Thus we have shown the existence of a pseudo-polynomial time algorithm for the Feasibility Problem with three deadlines. Extension to any fixed number of deadlines is easy.

A similar construction works for the Schedule Cost Problem. □

7. Conclusions. We have demonstrated that even very restrictive cases of sequencing problems with deadlines and set-up times/costs are *NP*-complete. The



b - SET-UP TIMES (CHANGEOVER COSTS)
c - NUMBER OF TASKS PER CLASS
d - NUMBER OF DISTINCT DEADLINES
(*)- INCLUDING AN INFINITE DEADLINE IN THE CASE OF THE SCHEDULE COST PROBLEM

ASSUMPTIONS:
- PROCESSING TIMES ARE VARIABLE
- $c \leq d$, WITHOUT LOSS OF GENERALITY

FIG. 5. *Summary of results for feasibility (schedule cost) problem.*

various completeness results are summarized in Table 1. We have also shown that pseudo-polynomial time algorithms exist for the problems in this table and some of their generalizations, thereby giving evidence that stronger results are not possible. Thus it is unlikely that any of these problems is *NP*-complete when task lengths are constrained to be polynomially bounded in the number of tasks. Moreover the problems listed in Table 1 are likely to be the most restricted cases of the general problem which are *NP*-complete—in each case reducing either the number of tasks per class or the number of distinct deadlines yields a problem which can be solved in polynomial time. We summarize these results in Fig. 5.

One issue that we have not been able to resolve is whether the general problem is *NP*-complete when the task lengths, set-up times and/or change-over costs are not allowed to be exponentially large with respect to the number of tasks.

Acknowledgment. The results presented concern a fixed number d of deadlines and a single *release date* (at time zero). We are indebted to Teofilo Gonzalez for the observation that the same set of results goes through for one deadline and a fixed number of release dates.

REFERENCES

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

[2] E. G. COFFMAN, *Computer and Job-Shop Scheduling Theory*, John Wiley, New York, 1976.

[3] W. C. DRISCOLL, *Scheduling production on one machine with changeover times*, Bull. ORSA 23 (1975), Supp. 23, p. B-418.

[4] ———, *A branch-bound algorithm for scheduling production on one machine with changeover costs*, Dept. of Industrial Engng., Youngstown State Univ., Youngstown, OH, 1976.

[5] W. C. DRISCOLL AND H. EMMONS, *Scheduling production on one machine with changeover costs*, Trans. Amer. Inst. Industrial Engineers, to appear.

[6] M. R. GAREY AND D. S. JOHNSON, "*Strong*" *NP-completeness results: Motivation, examples and implications*, J. Assoc. Comput. Mach., to appear.

[7] R. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[8] C. R. GLASSEY, *Minimum changeover scheduling of several products on one machine*, Operations Res. 16 (1968), pp. 342–352.

[9] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[10] S. MITSUMORI, *Optimal production scheduling of multi-commodity in flow line*, IEEE Trans. Systems, Man and Cybernetics, SMC-2 (1972), pp. 486–493.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0002 $01.00/0

POLYNOMIAL ALGORITHMS FOR DETERMINISTIC PUSHDOWN AUTOMATA*

DANIEL J. ROSENKRANTZ† AND HARRY B. HUNT III‡

Abstract. An algorithm is presented for converting a deterministic pushdown automaton (dpda) of size n into an equivalent dpda that always halts. The dpda produced is of size $O(n)$. The algorithm operates in linear time on a random access machine (but may require the allocation of $O(n^2)$ storage), and in time $O(n^2)$ on a multi-tape Turing machine. Related results on polynomial time algorithms for dpda equivalence problems and for two-way pushdown automata language recognition problems are discussed.

Key words. pushdown automata, cycle removal, halting, two-way pushdown automata

1. Introduction. Deterministic pushdown automata (dpda) are of both practical and theoretical interest. A dpda may have the defect that it cycles (does not halt) for some input sequences. However an arbitrary dpda can be redesigned so that it never cycles. The redesigned dpda can in turn easily be converted into a dpda that accepts the complement of the language accepted by the original dpda. Previous algorithms for the redesign to eliminate cycles either emphasize the form of the redesigned machine and are not specific enough to have an analyzable time bound ([14], [4]), or take an amount of time that is exponential in the size of the machine description ([6], [2], [12]). However, the redesign can be done in polynomial time, as is noted in [9] and [16]. In this paper, we present a redesign algorithm that operates in linear time on a unit-cost random access machine, and in time $O(n^2)$ on a multi-tape Turing machine. Our linear time redesign algorithm provides additional motivation for studying the complexity of decision procedures for dpda's in cycle-free normal form, as required by the algorithms in [17].

We also discuss several results concerning polynomial time algorithms for related equivalence problems, and show how our linear time redesign algorithm relates to some other polynomial time problems. A new proof that each two-way dpda language is recognizable in linear time on a unit-cost random access machine is presented.

2. Definitions and notation. We use ε to denote the empty string and ϕ to denote the empty set. For a set Σ, we let $|\Sigma|$ denote the cardinality of Σ, and for a string ξ, we let $|\xi|$ denote the length of ξ. For a set of symbols Σ, we let Σ_ε denote $\Sigma \cup \{\varepsilon\}$.

A *deterministic pushdown automaton* (*dpda*) is a seven-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where

1. Q is a finite set of states;
2. Σ is a finite set of input symbols;
3. Γ is a finite set of stack symbols;
4. q_0 in Q is the initial state;
5. Z_0 in Γ is the initial stack symbol;
6. $F \subseteq Q$ is the set of accepting states; and
7. δ, the transition function, is a mapping of $Q \times \Sigma_\varepsilon \times \Gamma$ into $(Q \times \Gamma^*) \cup \{\varnothing\}$ subject to the constraint that if $\delta(q, \varepsilon, A) \neq \varnothing$ then $\delta(q, a, A) = \varnothing$ for each a in Σ.

* Received by the editors June 14, 1977, and in revised form January 16, 1978.

† Department of Computer Science, State University of New York at Albany, Albany, New York 12222. This work was done while this author was at the General Electric Co. Research and Development Center, Schenectady, New York.

‡ Department of Electrical Engineering and Computer Science, Columbia University, New York, New York 10027. The work of this author was supported by the National Science Foundation under Grant No. DCR 75-22505.

We define a *configuration* as an element of $Q \times \Sigma^* \times \Gamma^*$. We define the relation $\vdash$ on configurations by

$$(p, ax, A\gamma) \vdash (q, x, \xi\gamma)$$

if $\delta(p, a, A) = (q, \xi)$. If $a = \varepsilon$, we say that the second configuration is obtained from the first by an *ε-move*.

Let $\vdash^*$ be the reflective transitive close of $\vdash$, and let $\vdash^+$ be the transitive closure of $\vdash$. The *language accepted* by M is

$$L(M) = \{x \mid (q_0, x, Z_0) \vdash^* (p, \varepsilon, \gamma) \text{ for some } p \text{ in } F \text{ and } \gamma \text{ in } \Gamma^*\}.$$

Thus to accept an input sequence, the dpda must read the entire input sequence and then (perhaps during a series of ε-moves) enter an accepting state. Two machines are said to be *equivalent* if they accept the same language.

A dpda halts for input sequence x if there is a configuration α such that

$$(q_0, x, Z_0) \vdash^* \alpha$$

and there is no configuration β such that $\alpha \vdash \beta$. If a dpda does not halt for a particular input sequence, we say that it *cycles* for that input sequence.

Define the symbol count of a pair (q, ξ), where q is in Q and ξ is in Γ^* as $|\xi| + 1$, and the symbol count of $\varnothing$ as zero. We define the symbol count of δ, denoted $|\delta|$, as the sum over all q in Q, a in Σ_ε, and Z in Γ, of the symbol count of $\delta(q, a, Z)$. Note that if $\delta(q, a, Z)$ is null, then (q, a, Z) does not contribute to $|\delta|$. We define the *symbol count* of a dpda as $|Q| + |\Sigma| + |\Gamma| + |\delta|$. If the symbol count is m, we define the *size* of the dpda to be $m \cdot \log(|Q| + |\Sigma| + |\Gamma|)$. The number of symbols used to describe the dpda is proportional to the symbol count, and the number of bits required is proportional to the size.

In analyzing the complexity of algorithms, we consider three computing devices: a multi-tape Turing machine, a unit-cost random access machine (unit-cost RAM), and a log-cost RAM. A unit-cost RAM is assumed capable of manipulating a symbol or following a pointer in a single unit of time. A log-cost RAM is assumed to have to separately process each bit in the binary encoding of a symbol, and thus for polynomial space algorithms processes a symbol or follows a pointer in time bounded by the logarithm of the dpda size. For practical computations on dpda's, the unit-cost RAM is the most realistic of the three models. Algorithms taking time O (symbol count) on a unit-cost RAM take time O ((symbol count) $\cdot$ log (size)) on a log-cost RAM. For the class of dpda's that push at most one symbol at a time, log (size) is $O(\log(|Q| + |\Sigma| + |\Gamma|))$, and so algorithms taking time O (symbol count) on a unit-cost RAM take time O (size) on a log-cost RAM.

For a unit-cost RAM, a well-known technique for implementing sparse arrays can be used to access array elements in unit time, without having to initialize the entire array. For instance, suppose we want an array of size $|Q| \cdot |\Gamma|$, but with valid entries for a designated subset of $Q \times \Gamma$ (say $\{(q, A) \mid \delta(q, \varepsilon, A) \neq \varnothing\}$). Given a (q, A) in the designated subset, the corresponding entry in an array of size $|Q| \cdot |\Gamma|$ can be found in unit time by a unit-cost RAM. Given a list of members of the designated subset and an array of size $|Q| \cdot |\Gamma|$, the array can be initialized by a unit-cost RAM in time bounded by the cardinality of the designated subset. For each item in the list, the initialization consists of storing a pointer to the list item in the corresponding array entry and placing a pointer to the array entry in the list item. Any array entry can subsequently be checked for validity by determining whether it points into the list and if so, whether the list item also points back at the array entry.

3. Eliminating cycles.

THEOREM 3.1. *There is an algorithm for converting an arbitrary dpda of size n into an equivalent dpda of size $O(n)$ that always halts. The algorithm takes time O* (symbol count) *on a unit-cost RAM and $O(n^2)$ on a multi-tape Turing machine.*

Proof. We describe an algorithm that takes time O (symbol count) on a unit-cost RAM. It should be clear that each unit step of this algorithm can be done within time n on a multi-tape Turing machine. The algorithm consists of a sequence of transformations of the dpda. Each transformation maintains the time and size bounds of the theorem. In fact, the symbol count of the dpda after each transformation is bounded by twice the symbol count of the dpda before the transformation.

If a dpda enters a nonterminating loop of ε-moves, then either the pushdown stack oscillates in size, or no symbol pushed on the stack during the loop is subsequently popped off. Our second transformation prevents stack oscillations by assuring that no symbol placed on the stack during a series of ε-moves is ever popped off by a continuation of that series. Our third transformation cuts off loops that do not involve popping. Our first transformation is necessary to preserve the language accepted by the dpda when sequences of ε-moves are bypassed by the second transformation.

The first transformation changes the dpda so that if the machine is in an accepting state, all subsequent configurations obtained solely by ε-moves also have accepting states. For each state p, we introduce a new accepting state $\hat{p}$. For each a in Σ_ε and A in Γ, we define $\delta(\hat{p}, a, A)$ as follows—

$$\delta(\hat{p}, a, A) = \begin{cases} \varnothing & \text{if } \delta(p, a, A) = \varnothing, \\ (r, \gamma) & \text{if } \delta(p, a, A) = (r, \gamma) \quad \text{and } a \neq \varepsilon, \\ (\hat{r}, \gamma) & \text{if } \delta(p, a, A) = (r, \gamma) \quad \text{and } a = \varepsilon. \end{cases}$$

Next, whenever $\delta(q, \varepsilon, A) = (r, \gamma)$ for some accepting state q, δ is changed so that

$$\delta(q, \varepsilon, A) = (\hat{r}, \gamma).$$

For p in Q and A in Γ, we say that a pair (p, A) is *reducible* if $\delta(p, \varepsilon, A) = (q, B\xi)$ where B is in Γ, ξ is in Γ^*, and $(q, \varepsilon, B) \vdash^* (r, \varepsilon, \varepsilon)$. If $\delta(q, \varepsilon, B) = (r, \varepsilon)$, we say that (p, A) is *directly reducible*. The second transformation changes the dpda so that no pair (p, A) is reducible. Thus when an ε-move of the transformed dpda places a symbol on the stack, that symbol is not popped off the stack by a sequence of moves consisting only of ε-moves. In particular if for the current dpda

$$(s, \varepsilon, B) \vdash^* (t, \varepsilon, \varepsilon),$$

then for the transformed dpda

$$\delta(s, \varepsilon, B) = (t, \varepsilon).$$

The transformation consists of repeatedly simplifying transitions for directly reducible pairs, until there are no directly reducible pairs. Once there are no directly reducible pairs, there are no reducible pairs. Therefore after the transformation, it is impossible to pop a symbol stacked earlier during the same sequence of ε-moves.

The transformation is an efficient implementation of the following loop.

while δ contains a move of the form
$\delta(p, \varepsilon, A) = (q, B\xi)$ with
$\delta(q, \varepsilon, B) = (r, \varepsilon)$
do change $\delta(p, \varepsilon, A)$ to (r, ξ)

This loop must terminate after O(symbol count) iterations because each time around the loop the value of $|\delta|$ decreases.

The transformation makes use of the following data structures:

1. *δ-table* is a sparse array of size $|Q| \cdot |\Gamma|$ with an entry for each (p, A) having an ε-move. The entry contains $\delta(p, \varepsilon, A)$, where if $\delta(p, \varepsilon, A) = (q, \gamma)$, then γ is stored as a linked list.
2. *predecessor-table* is a sparse array of size $|Q| \cdot |\Gamma|$ with an entry for each (p, A) having an ε-move. The entry for (p, A) contains a linked list of pairs

$$\{(s, C) | \delta(s, \varepsilon, C) = (p, A\gamma) \text{ for some } \gamma \in \Gamma^*\}.$$

3. *DR-list* of pairs (p, A) in $Q \times \Gamma$. The DR-list is initialized to $\{(p, A) | (p, A)$ is directly reducible$\}$.

These data structures can be constructed or initialized in time O(symbol count) by a unit-cost RAM.

The transformation consists of constructing or initializing the three data structures and then performing the loop until the DR-list is empty. Each time around the loop, the following steps are taken:

1. Delete a member of the DR-list, say (p, A). Suppose $\delta(p, \varepsilon, A) = (q, B\gamma)$ and $\delta(q, \varepsilon, B) = (r, \varepsilon)$.
2. Change $\delta(p, \varepsilon, A)$ to (r, γ). This involves changing the δ-table entry for (p, A).
3. If $\gamma = \varepsilon$, then add to the DR-list each (s, C) on the linked list pointed to by the predecessor-table entry for (p, A).
4. If $\gamma = C\xi$, add (p, A) to the predecessor-table entry for (r, C). Furthermore, if (p, A) is directly reducible, i.e., if $\delta(r, \varepsilon, C) = (s, \varepsilon)$, then add (p, A) to the DR-list.

Each time around the loop steps 1, 2, and 4 take a fixed amount of time on a unit-cost RAM. Each item added to the DR-list during step 3 is subsequently deleted from the DR-list during some execution of the loop. Since there are at most $|\delta|$ iterations of the loop, the number of additions to the DR-list occurring during all executions of step 3 is bounded by $|\delta|$. Thus the entire second transformation takes time O(symbol count).

The third transformation involves the construction of a directed graph whose nodes are a subset of $Q \times \Gamma$. Each node has at most one edge leaving it. The graph contains an edge for each non-popping ε-move. Specifically, suppose that $\delta(p, \varepsilon, A) = (q, B\xi)$. Then the graph contains an edge from (p, A) to (q, B).

Note that there is a path in the graph from (p, Z) to (r, Y) if and only if for some γ in Γ^*

$$(p, \varepsilon, Z) \vdash^{+} (r, \varepsilon, Y\gamma).$$

If a node is a sink of the graph, or the path leaving the node ends in a sink, we say that the node is *halting*. All other nodes (i.e., the nodes that are part of a cycle or lead to a cycle) we call *nonhalting*.

The transition function is now changed based on the graph. If (p, Z) is a nonhalting node and p is an accepting state, then $\delta(p, \varepsilon, Z)$ is changed to $\varnothing$. If (p, Z) is nonhalting, p is not an accepting state, and there is no path in the graph from (p, Z) to a node with an accepting state, then $\delta(p, \varepsilon, Z)$ is also changed to $\varnothing$. Otherwise, the transition function is unchanged.

The computation required for the third transformation involves finding all nodes having a path to a sink, and finding all nodes having a path to a node with an accepting

state. Since the number of edges in the graph is bounded by the symbol count, the third transformation can be done in the stated time.

The new machine never cycles. Any cycle in the previous machine must reach some smallest stack and then pass through some configuration corresponding to a node whose transition was changed to $\varnothing$. □

THEOREM 3.2. *Determining whether an arbitrary dpda of size n accepts a string x can be done in time* $O(\text{symbol count} \cdot (|x|+1))$ *on a unit-cost RAM and time* $O(n^2 \cdot (|x|+1))$ *on a multi-tape Turing machine.*

Proof. Apply the algorithm of Theorem 3.1 to the given dpda. The converted machine, presented with input string x, starts with an initial stack consisting of a single symbol. Let δ be the transition function of the converted machine. Let $|\delta_\varepsilon|$ be the portion of $|\delta|$ due to ε-moves, and $|\delta_\Sigma|$ be the portion of $|\delta|$ due to non-ε-moves. Each of the $|x|$ possible non-ε-moves can push at most $|\delta_\Sigma|$ symbols on the stack. Each series of ε-moves consists of a popping phase, followed by a pushing phase in which no symbols are popped off the stack. For each pushing phase, $|\delta_\varepsilon|$ is a bound on the number of transitions and a bound on the number of symbols pushed. The total number of symbols pushed while processing x is a bound on the number of symbols popped. If the converted machine pops a symbol off the stack via ε-moves, it does so in a single transition. Therefore the number of transitions in all the popping phases is bounded by (symbol count) $\cdot$ $(|x|+1)$. The total sequence of moves in processing x can thus be simulated within the time bounds of the theorem. □

COROLLARY 3.3. *Determining whether an arbitrary dpda of size n accepts* ε *can be done in time* $O(\text{symbol count})$ *on a unit-cost RAM.*

4. Polynomial equivalence problems for deterministic languages. The fact that cycles can be eliminated from dpda's in polynomial time directly implies that several equivalence and containment problems are decidable in polynomial time.

THEOREM 4.1. *Given a dpda M accepting some language over* Σ, *a dpda* M' *can be obtained in polynomial time such that*

$$L(M') = \Sigma^* - L(M).$$

Proof. First modify the dpda if necessary, so that the initial stack symbol serves as a "bottommarker" that is never popped off the stack. Then use a method similar to that in [4] to obtain M', but using the method of Theorem 3.1 to eliminate cycles. □

THEOREM 4.2. (1) *The set* $\{(M, R) | M$ *is a dpda,* R *is a deterministic finite state automaton, and* $L(M) = L(R)$ $[L(M) \supseteq L(R)$, *or* $L(M) \subseteq L(R)]\}$ *is polynomial time recognizable.*

(2) *The set* $\{(M, R) | M$ *in a dpda,* R *is a regular expression, and* $L(M) \supseteq L(R)\}$ *is polynomial time recognizable.*

Proof. $L(M) \supseteq L(R)$ if and only if $\overline{L(M)} \cap L(R)$ is empty, and $L(M) \subseteq L(R)$ if and only if $L(M) \cap \overline{L(R)}$ is empty. In polynomial time a dpda and a finite state automaton (or regular expression) can be converted into a nondeterministic pushdown automaton recognizing the intersection of their languages, and this automaton can be tested for emptiness. □

An immediate corollary of Theorem 4.2 is the following.

COROLLARY 4.3. *For each regular set* R_0, *there is a polynomial time algorithm to decide, for a dpda M, if* $L(M) = R_0$, $L(M) \supseteq R_0$, *or* $L(M) \subseteq R_0$.

One approach that has been used to obtain lower bounds on the complexity of equivalence and containment problems is to establish the lower bound for testing equivalence to the language $\{0, 1\}^*$([15], [7], [10]). Corollary 4.3 shows that a

straightforward application of this approach cannot provide a nonpolynomial lower time bound for the dpda equivalence problem. Furthermore, the dpda equivalence results above also apply to classes of context-free grammars for which an equivalent dpda can be constructed in polynomial time. These classes include the LL(1) grammars, the strict deterministic grammars [18], the uniquely invertible weak precedence grammars, the uniquely invertible operator precedence grammars, and the simple mixed strategy precedence grammars. Thus a straightforward application of the techniques used in [15], [7], and [10] cannot be used to derive a nonpolynomial lower time bound for the equivalence problem for LL(1) grammars, etc.

Theorem 4.2 presents a subclass of the dpda equivalence problem that is provably decidable in polynomial time. A related equivalence problem, that is also decidable in polynomial time, is the equivalence problem for linear s-grammars [11].

The linear time unit-cost RAM algorithm of Corollary 3.3 implies the existence of several other linear time unit-cost RAM algorithms as well. First, consider the emptiness problem for dpda's with a singleton input alphabet.

THEOREM 4.4. *The emptiness problem for dpda's with a singleton input alphabet is decidable in linear time on a unit-cost RAM.*

Proof. Let $M = (Q, \{a\}, \Gamma, \delta, q_0, Z_0, F)$ be the given dpda. The algorithm for emptiness first modifies M by replacing each transition of the form $\delta(p, a, A) = (q, \gamma)$ with the new transition $\delta(p, \varepsilon, A) = (q, \gamma)$. Note that the ε-transitions of M are retained. The new dpda, M', has the property that $L(M) \neq \varnothing$ if and only if ε is accepted by $L(M')$. From Corollary 3.3, this recognition problem is decidable in linear time on a unit-cost RAM. □

Corollary 3.3 also provides an alternate way of obtaining the result in [3] that every two-way dpda language is recognizable in linear time on a unit-cost RAM.

THEOREM 4.5. *Every two-way dpda language is recognizable in linear time by a unit-cost RAM.*

Proof. Let M be a two-way dpda, and let $x = a_1 a_2 \cdots a_n$ be an input string to M. Using an encoding technique from [5] and [8], x can be transformed into a dpda M_x such that x is in $L(M)$ if and only if ε is in $L(M_x)$. A state of M_x is a pair consisting of a state of M plus a position in x (i.e., an integer between 1 and n). Intuitively, M' simulates the operation of M on input x. Let δ and δ_x be the transition functions of M and M_x, respectively. If $\delta(p, a_i, A)$ or $\delta(p, \varepsilon, A)$ is nonnull (at most one is nonnull), then $\delta_x((p, i), \varepsilon, A)$ is nonnull and encodes the corresponding move of M.

The number of states and nonnull transitions of M_x is proportional to n (with the constant of proportionality depending on M). Therefore the symbol count of M_x is proportional to n. From Collorary 3.3, a unit-cost RAM can test membership of ε in $L(M_x)$ in time proportional to the symbol count of M_x. □

Conversely, Theorem 4.5 implies Corollary 3.3 and Theorem 4.4 since the languages $\mathscr{L}_1 = \{M | M$ is a dpda accepting $\varepsilon\}$ and $\mathscr{L}_2 = \{M | M$ is a dpda with a singleton input alphabet and $L(M) \neq \varnothing\}$ are both 2-way dpda languages [8].

Testing dpda for equivalence to any fixed deterministic language is at least as hard as the membership problem for two-way nondeterministic pushdown automata (*2npda*), as indicated by the following.

PROPOSITION 4.6. *For any fixed deterministic context-free language L_0, any 2npda language L_1 is reducible to $\{M | M$ is a dpda and $L(M) = L_0\}$ in linear time by a unit-cost RAM and in time and space n $(\log n)$ by a multi-tape Turing machine.*

Proof. The proof uses encoding techniques from [5] and [8]. Let N be a 2npda recognizing L_1 and M_0 be a dpda that recognizes L_0 and always halts. We assume without loss of generality, that the transition function of N permits at most two choices for each move.

Let $x = a_1 a_2 \cdots a_n$ be a string to be tested for membership in L_1. The reducibility is based on transforming x into a dpda M_x such that x is in L_1 if and only if $L(M_x) \neq L_0$.

Let Σ_0 be the input set of M_0 and $\{c, d\}$ be two new symbols. The input alphabet of M_x is $\Sigma_0 \cup \{c, d\}$. Machine M_x simulates M_0 until it encounters a c or d. It then empties its stack and starts to simulate N. In the simulation of N, the state of M_x is a pair consisting of a state of N and a position in x. Let δ_x and δ_N be the transition function of M_x and N, respectively. Then $\delta_x((p, i), c, A)$ and $\delta_x((p, i), d, A)$ correspond to the choices allowed by $\delta_N(p, a_i, A)$. Once M_x starts simulating N, it enters an accepting state if it reads an input sequence corresponding to a sequence of nondeterministic choices of N leading to acceptance of x. Note that M_x accepts an input string in Σ^* if and only if M_0 does. Thus $L(M) = L_0 \cup L_2$ where $L_2 \subseteq \Sigma_0^* \{c, d\}^+$ and L_2 is empty if and only if x is not in L_1. □

Again as shown in [8], the language $\mathcal{L}_3 = \{M | M$ is a 2-way dpda (2npda) and $L(M) \neq \varnothing\}$ is a 2npda language. Thus, the uniform "lower bound" of Proposition 4.6 is fairly tight.

The best known algorithm for the recognition of 2npda languages takes time $O(n^3)$ on a unit-cost RAM [1]. An alternate algorithm with the same time bound can be obtained using Prop. 4.6 with $L_0 = \varnothing$. The alternate algorithm, outlined in [8], consists of converting x into M_x, converting M_x into an equivalent context-free grammar, and testing the grammar for emptiness. The grammar is nonempty if and only if x is in L_1. The conversion of M_x into the grammar can be done in time $O(n^3)$ on a unit-cost RAM [6], and, as is well known, the grammar can be tested for emptiness in linear time by a unit-cost RAM.

Acknowledgment. We wish to thank L. G. Valiant for suggesting the RAM implementation of the algorithm of Theorem 3.1, and for Theorem 4.4. In addition, we wish to thank R. E. Stearns and a referee for helpful comments.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *Time and tape complexity of pushdown automaton languages*, Information and Control, 13 (1968), pp. 186–206.

[2] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation, and Compiling*, vol. 1, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[3] S. A. COOK, *Linear time simulation of deterministic two-way pushdown automata*, Proc. IFIP Congress 71, TA-2, North-Holland, Amsterdam, 1971, pp. 172–179.

[4] S. GINSBURG AND S. GREIBACH, *Deterministic context-free languages*, Information and Control, 9 (1966), pp. 620–648.

[5] J. E. HOPCROFT AND J. D. ULLMAN, *Decidable and undecidable questions about automata*, J. Assoc. Comput. Mach., 15 (1968), pp. 317–324.

[6] ———, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[7] H. B. HUNT III, *On the time and tape complexity of languages*, Ph.D. thesis, Cornell Univ., Ithaca, New York, 1973.

[8] ———, *On the complexity of finite, pushdown, and stack automata*, Math. Systems Theory, 10 (1976), pp. 33–52.

[9] H. B. HUNT III AND D. J. ROSENKRANTZ, *Computational parallels between the regular and context-free languages*, Proc. Sixth Annual ACM Symp. on Theory of Computing (1974), pp. 64–74.

[10] H. B. HUNT III, D. J. ROSENKRANTZ AND T. G. SZYMANSKI, *On the equivalence, containment and covering problems for the regular and context-free languages*, J. Computer and System Sciences, 12 (1976), pp. 222–268.

[11] ———, *The covering problem for linear context-free grammars*, Theoret. Comput. Sci., 2 (1976), pp. 361–382.

[12] J. VAN LEEUWEN AND C. H. SMITH, *An improved bound for detecting looping configurations in deterministic dpa's*, Information Processing Lett., 3 (1974), 22–24.

[13] A. R. MEYER AND L. J. STOCKMEYER, *The equivalence problem for regular expressions requires exponential space*, Proc. 13th Annual IEEE Symp. on Switching and Automata Theory (1972), pp. 125–129.

[14] M. P. SCHUTZENBERGER, *On context-free languages and push-down automata*, Information and Control, 6 (1973), pp. 246–264.

[15] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time*, Proc. Fifth Annual ACM Symp. on Theory of Computing (1973), pp. 1–9.

[16] L. G. VALIANT, *Decision procedures for families of deterministic pushdown automata*, Ph.D. thesis, University of Warwick, Coventry, England, 1973.

[17] ———, *Regularity and related problems for deterministic pushdown automata*, J. Assoc. Comput. Mach., 22 (1975), pp. 1–10.

[18] M. A. HARRISON AND I. HAVEL, *Real-time strict deterministic languages*, this Journal, 1 (1972), pp. 333–349.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0003 $01.00/0

FINDING A MINIMUM CIRCUIT IN A GRAPH*

ALON ITAI† AND MICHAEL RODEH‡

Abstract. Finding minimum circuits in graphs and digraphs is discussed. An almost minimum circuit is a circuit which may have only one edge more than the minimum. To find an almost minimum circuit an $O(n^2)$ algorithm is presented. A direct algorithm for finding a minimum circuit has an $O(ne)$ behavior. It is refined to yield an $O(n^2)$ average time algorithm. An alternative method is to reduce the problem of finding a minimum circuit to that of finding a triangle in an auxiliary graph. Three methods for finding a triangle in a graph are given. The first has an $O(e^{3/2})$ worst case bound ($O(n)$ for planar graphs); the second takes $O(n^{5/3})$ time on the average; the third has an $O(n^{\log 7})$ worst case behavior. For digraphs, results of Bloniarz, Fisher and Meyer are used to obtain an algorithm with $O(n^2 \log n)$ average behavior.

Key words. graph, digraph, triangle, circuit, shortest path, matrix multiplication, analysis of an algorithm, computational complexity, worst-case, average-case, random graph

1. Introduction. In this paper we discuss finding short circuits in graphs and digraphs. The problem of digraphs arose when we tried to define the distance between two perfect matchings in a bipartite graph [4]. We assume that the reader is familiar with the standard definitions of graph theory [9]. Let $G = (V, E)$ be a graph with n vertices and e edges. In this paper the edges of a path (circuit) are all distinct. The length of a path (circuit) is the number of its edges. We assume that the vertices are numbered and we shall not distinguish between a vertex and its number. A minimum circuit is a circuit whose length is minimum. Harary [6] defines the girth of a graph to be the length of its minimum circuit. Several theorems relate to this notion [5], [7]. An almost minimum circuit is a circuit whose length is greater than that of a minimum circuit by at most one. We present an $O(n^2)$ algorithm for finding an almost minimum circuit. To find a minimum circuit we develop an $O(n^2)$ average time algorithm. The straightforward algorithm for finding a minimum circuit has an $O(ne)$ behavior. We also show an $O(n^2)$ reduction from the problem of finding a minimum circuit to that of finding a triangle (a circuit of length 3). Three methods for finding triangles are presented:

(i) Using rooted trees. The algorithm takes $O(e^{3/2})$ time in the worst case and $O(n)$ for planar graphs.

(ii) Check directly whether an edge is contained in a triangle. $O(ne)$ worst case and $O(n^{5/3})$ average time.

(iii) By Boolean matrix multiplication, in $O(n^{\log 7})$ time [10] (all logarithms are taken to base 2).

Algorithms for finding a shortest path in digraphs can be adapted to finding a minimum directed circuit (dicircuit). In particular, Friedman's $O(n^3(\log\log n/\log n)^{1/3})$ algorithm for weighted graphs [8] and directed breadth first search. The latter requires $O(ne)$ time in the worst case. However, it is proven, using the methods of [2], that on the average $O(n^2 \log n)$ time suffices. Using Boolean matrix multiplication we show how to find a shortest dicircuit in at most $O(n^{\log 7} \log n)$ time.

We use three representations of labeled graphs:

(i) The adjacency lists: $A(v)$ is the set of vertices adjacent to v. In this paper it is assumed that all graphs are given in this representation.

* Received by the editors November 24, 1976, and in revised form on October 18, 1977.

† Computer Science Department, Technion, Israel Institute of Technology, Haifa, Israel.
‡ IBM, Israel Scientific Center, Haifa, Israel.

(ii) The upper adjacency vectors: $UA(v)$ is a sorted vector which contains those vertices $w > v$ adjacent to v. This representation depends on the labeling of the vertices. Each edge is represented in exactly one vector. The vectors may be obtained from the adjacency lists in $O(e)$ time (using bucket sort).

(iii) The adjacency matrix: $(M)_{u,v} = 1$ if and only if u and v are connected by an edge. The adjacency matrix may be constructed from the adjacency lists in $O(e)$ time [1, p. 71, Ex. 2.12], even though this representation requires $O(n^2)$ space. Hence n^2 is a lower bound to the space requirements of all algorithms which use this representation.

2. Finding an almost minimum circuit. Let $G = (V, E)$ be an undirected graph with n vertices and e edges which has neither parallel edges nor self loops. Let *lmc* denote the length of a minimum circuit (if none exists then $lmc = \infty$). A circuit is *an almost minimum circuit* if its length is less than or equal to $lmc + 1$. We present an $O(n^2)$ algorithm for finding an almost minimum circuit.

First we present the algorithm *FRONT*. Given a vertex $v \in V$ this algorithm finds a lower bound for the length of the shortest circuit through v. The algorithm assigns values to two global variables: the vector k of length n and the $n \times n$ matrix *level*. These values are used in the sequel. *FRONT*(v) conducts a partial breadth first search (BFS) from v. When defined the value of $level(v, u)$ is the level of u in the search. If the connected component which contains v is circuit-free then the algorithm terminates with $k(v) = \infty$. Otherwise, it stops when the first circuit is closed; this circuit does not necessarily pass through v; $k(v)$ is defined to be the last level from which the search was conducted; $2k(v)+1$ is a lower bound for the length of the minimum circuit through v.

The algorithm *FRONT* uses a first-in, first-out queue which is initially empty. The queue operations are *enqueue*(u) which inserts u at the rear of the queue, and *dequeue* which removes and takes the value of the first element of the queue.

```
   procedure FRONT(v);
   begin for u ∈ V do level(v, u) := nil;
      enqueue(v); level(v, v) := 0;
      while the queue is not empty do
      begin comment: if the graph contains a circuit in the connected component of v
                     then the queue is never empty at this point;
         u := dequeue;
         for w ∈ A(u) do
            begin if level(v, w) = nil
                  then begin level(v, w) := level(v, u)+1;
                             enqueue(w) end
1.                else if level(v, u) ≦ level(v, w)
                        then begin k(v) := level(v, u);
                                   return end
            end
      end;
      comment the connected component of v is circuit-free;
2.    k(v) := ∞
   end
```

FRONT builds a partial BFS tree. When a nontree edge is encountered (line 1) the algorithm terminates. Otherwise $k(v) = \infty$ (line 2). Each tree edge is scanned at

most twice. Thus the algorithm takes $O(n)$ time. In the queue each vertex may appear at most once. Therefore, the algorithm requires $O(n)$ space for local variables, the vector *level* of length n, and the queue, in which each vertex can appear at most once. Hence, the algorithm requires $O(n)$ space in addition to the input. Observe that a minimum circuit through v could be found by scanning all the edges. In the worst case this takes $O(e)$ time. In the next section we present a method of scanning which takes $O(n)$ time on the average.

Let us apply *FRONT* to every vertex $v \in V$, and let *kmin* be the minimum value of $k(v)$.

LEMMA 1. *Let x be a vertex for which $k(x) = kmin < \infty$, then x is contained in an almost minimum circuit.*

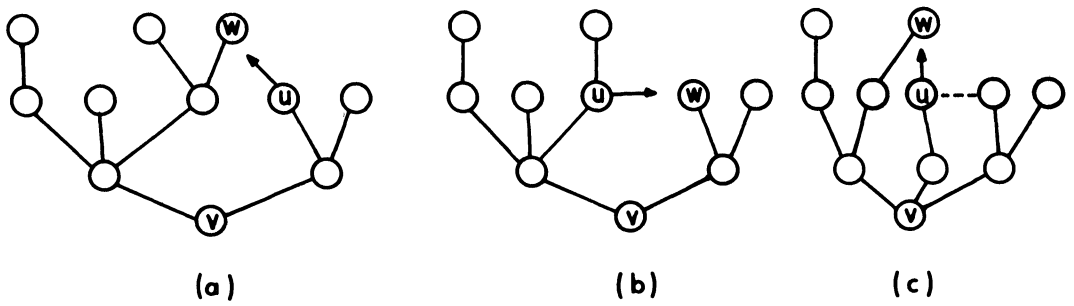


FIG. 1

Proof. Let v be a vertex which belongs to a minimum circuit C. If *lmc* is even, *FRONT*(v) stops when encountering a vertex w as in Fig. 1a; $k(v) = lmc/2 - 1$. If *lmc* is odd the algorithm stops as in Fig. 1b or Fig. 1c; $k(v) = (lmc - 1)/2$.

$$2k(v) + 1 \leqq lmc \leqq 2k(v) + 2.$$

Since $k(v) \geqq kmin$, $2kmin + 1 \leqq lmc$. The circuit found when applying *FRONT* to x is not longer than $2kmin + 2$. Therefore, it is not longer than $lmc + 1$ and is an almost minimum circuit. This circuit contains x, since otherwise its length would have been at most $2(kmin - 1) + 2 = 2kmin < lmc$, a contradiction. Q.E.D.

Note that if *lmc* is even then for a vertex x on a minimum circuit the algorithm stops as in Fig. 1a and finds a minimum circuit. In particular, in bipartite graphs the length of all circuits is even and the algorithm finds a minimum circuit.

Since *FRONT* is applied n times at most $O(n^2)$ time is required to find an almost minimum circuit. If the algorithm is applied to the full bipartite graph to which we add zero or more edges the algorithm might find only circuits of length four, even though the graph may contain triangles. In this case the algorithm requires $O(n^2)$ time, hence the bound is tight for the algorithm.

The space requirements can be lowered to $O(n)$. As noted, the queue requires only $O(n)$ space. The matrix *level* can be replaced by a vector in which for all v *level*(v, u) share the same location. The algorithm for finding an almost minimum circuit can be optimized by keeping the value of *kmin* and terminating *FRONT*(v) whenever *level*$(v, w) = kmin$.

3. Finding a minimum circuit. We have shown how to find a minimum circuit for the special case in which the length is known *a priori* to be even. In this section we use

by-products of *FRONT* to develop an $O(n^2)$ average time algorithm to find a minimum circuit for the general case.

Assume that *FRONT* has been applied to a vertex v for which k is minimum and let us look at the values of *level*. If the connected component of v is circuit-free then the entire graph is circuit-free. Otherwise, a circuit is detected. Using the notation of *FRONT*, this circuit passes through u and w. If $level(v, u) = level(v, w)$ then the circuit is odd and thus minimum. Otherwise, the circuit is even and may not be minimum. It remains to check for the existence of an edge (x, y) such that $level(v, x) = level(v, y) = level(v, u)$. The vertex x must be either a vertex still in the queue or u itself. Thus, when $FRONT(v)$ terminates, define

$$F(v) = \{u\} \cup \{x \mid x \in V, x \text{ is in the queue, } level(v, x) = level(v, u)\}.$$

In $O(n)$ time we may sort $F(v)$ (bucket sort) and prepare a bit vector representing $F(v)$ and a linked list of its nonzero elements. The procedure *EDGE* below, when applied to $F(v)$ searches for an edge (x, y) in $F(v)$.

Let S be an ordered list of distinct vertices with the additional property that membership can be determined in constant time. (Observe that $F(v)$ satisfies these requirements.) $(x, y) \in E$ is an *S-edge* if $x, y \in S$. $EDGE(S)$ searches for vertices $u < w$ such that (u, w) is an S-edge. First it searches (lines 1–4) for (u, w) such that u is not among the last $n^{1/3}$ vertices of S. If unsuccessful, it searches exhaustively for an edge, the endpoints of which belong to the last $n^{1/3}$ portion of S (lines 5–6). If both searches fail then there exists no S-edge.

EDGE uses *UA* in a destructive mode. Since needed later, it can either be copied before use or reconstructed using a stack to undo all destructive operations. The latter solution is preferred since it enables a sublinear algorithm ($o(n)$). However, the details are omitted.

```
    procedure EDGE(S);
1.  begin for i := 1 step 1 until |S| - n^(1/3) do
      begin u := S(i);
            while UA(u) is not empty do
2.          begin choose at random a vertex w in UA(u);
3.                if w ∈ S then return ((u, w));
                  delete w from UA(u)
            end
4.    end;
5.    for i := max(1, |S| - n^(1/3) + 1) step 1 until |S| do
      begin u := S(i);
            for j := i + 1 step 1 until |S| do
            begin w := S(j);
                  if (u, w) ∈ E then return ((u, w))
            end
      end;
6.    return(nil)
    end
```

EDGE may require $O(n^2)$ time. However, its average behavior is better.

Let *ud* be *the upper degree vector* $(ud(v) = |UA(v)|)$ and G_{ud} be the class of all labeled graphs with a given *ud* vector. Observe that the class of all labeled graphs is a disjoint union of all the G_{ud} classes.

Let P be a probability measure on labeled graphs, such that any two graphs in G_{ud} are equiprobable. The following probability measures are special cases of P [3]:

(i) The existence of each edge is an independent random variable with equal probabilities.

(ii) All graphs with a given number of edges are equiprobable.

For $S \subseteq V$, let E_S be a subset of $S \times (V-S)$ and e_S the cardinality of E_S.

LEMMA 2. *Let* $G_{E_S} = \{G = (V, E) | E \supseteq E_S\}$. *Then the average behavior of EDGE on* G_{E_S} *is bounded by* $O(e_S + n^{2/3})$.

Proof. If (u, w) belongs to E_S then the check $w \in S$ (line 3) necessarily fails. *EDGE* might waste at most $O(e_S)$ time on such edges. Therefore, it suffices to prove that the other edges require $O(n^{2/3})$ time on the average.

Using the linked list representation of S and the adjacency matrix, lines 5–6 require at most $O(n^{2/3})$ time. Thus, it remains to show that lines 1–4 require $O(n^{2/3})$ average time.

Under P, all graphs in $G_{E_S} \cap G_{ud}$ are equiprobable. We now wish to estimate the probability that an edge (u, w) chosen at random in line 2 is an S-edge. By assumption (u, w) does not belong to E_S. Let there be l_1 edges in $UA(u) \cap E_S$. Denote by l_2 the number of edges in $UA(u) - E_S$ checked before (u, w). The vertex w may be any of $n - u - (l_1 + l_2)$ remaining vertices, with equal probabilities. Since $w > u$, if $w \in S$ then it can be any one of the vertices of $S \cap \{u+1, \cdots, n\}$. The probability that $w \in S$ is therefore:

$$\frac{|S \cap \{u+1, \cdots, n\}|}{n - u - (l_1 + l_2)} \geqq \frac{n^{1/3}}{n}.$$

By decreasing the probability of success, the average number of trials until the first success increases. Hence, the average execution time of lines 1–4 is bounded by

$$O\left(\sum_{i=1}^{\infty} i(1-n^{-2/3})^{i-1} n^{-2/3}\right) = O(n^{2/3}). \qquad \text{Q.E.D.}$$

The following procedure *MIN_CIRCUIT* finds a minimum circuit of length *lmc*. If *lmc* is finite the circuit passes through v. If *lmc* is odd then the circuit also passes through the edge a.

```
    procedure MIN_CIRCUIT(lmc, v, a);
1.  begin for v ∈ V do FRONT(v);
2.      find kmin;
        if kmin = ∞ then begin lmc := ∞;
                                return end;
3.      for v ∈ V and k(v) = kmin do
4.      begin find F(v);
                prepare a representation of F(v) as a sorted linked list;
5.              prepare a bit vector representation of F(v);
6.              a := EDGE(F(v));
                if a ≠ nil then begin lmc := 2kmin + 1;
                                      return end
7.      end;
        lmc := 2kmin + 2;
        v := any vertex for which k is minimum
    end
```

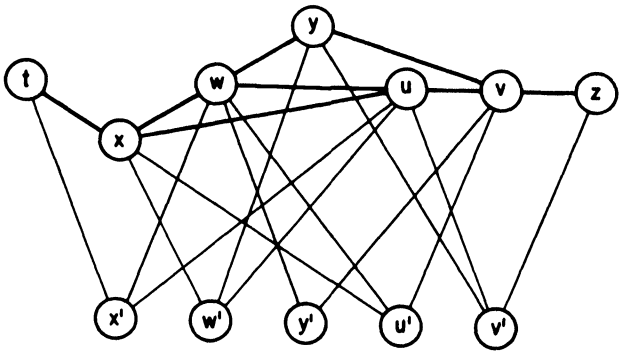
THEOREM 1. *The average execution time of MIN_CIRCUIT is bounded by* $O(n^2)$.

Proof. Line 1 requires at most $O(n^2)$ time; line 2, $O(n)$ time. In each iteration, lines 4–5 require $O(n)$ time. In line 6 *EDGE* is called with $S = F(v)$ and E_S is the set of edges incident with S which were scanned by *FRONT*(v). Hence, $e_S \leqq n$ and each iteration of line 6 costs $O(e_S + n^{2/3}) = O(n)$ time on the average. Since the loop (lines 4–7) may be executed at most n times, *MIN_CIRCUIT* requires $O(n^2)$ time on the average. (The average of a sum is equal to the sum of the averages.) Q.E.D.

Steps 1 and 2 of *MIN_CIRCUIT* can be done in one pass as explained in the end of the previous section. The space requirements can be lowered to $O(n)$ since we can keep a vector instead of the matrix *level*. In step 4, the values of *level*$(v, \cdot)$ are required to find $F(v)$, however *FRONT*(v) can be called again to obtain these values. This optimization increases the running time by at most a constant factor, while decreasing the space by a factor of n.

4. A reduction to finding triangles. Now we turn to show a reduction of the problem of finding a minimum circuit to that of finding a triangle in an auxiliary graph. A disadvantage of this method is that the number of edges might grow considerably. However, the number of vertices may only be doubled. Thereby, an upper bound for the complexity of the problem is found.

To this end we construct the graph $G' = (V' \cup V, E')$. V' consists of a copy of those vertices of G for which k is minimum. A vertex v' (v' denotes the vertex corresponding to v) is connected by an edge to all the vertices in $F(v)$. Fig. 2 contains an example of an auxiliary graph G'. The original graph G appears in boldface.



v	$F(v)$	$k(v)$
t	w, u	2
x	t, w, u	1
w	x, y, u	1
y	w, v	1
u	x, w, v	1
v	y, u, z	1
z	y, u	2

FIG. 2

LEMMA 3. *G' contains a triangle through v' if and only if v is contained in a minimum circuit in G and lmc is odd (i.e. $lmc = 2kmin + 1$).*

Proof. Let G' contain a triangle (v', x, y). By the construction, v' is connected only to vertices of $F(v)$. Therefore, $x, y \in F(v) \subseteq V$. The vertices x and y are at distance *kmin* from v. *FRONT* traces minimum length paths v–x, v–y. The length of these paths is *kmin* and they are vertex disjoint (i.e. they intersect only at v), because an additional intersection would entail a shorter circuit. (v', x, y) is a triangle in G' and $x, y \in V$. Thus, (x, y) belongs to E. This edge and the two paths form a circuit of length $2kmin + 1$. Since $lmc \geqq 2kmin + 1$ the circuit is minimum.

In the other direction, assume *lmc* is odd and a minimum circuit passes through v. Therefore, $lmc = 2kmin + 1$, $k(v) = kmin$ and $v' \in V'$. Let C be a minimum circuit through v. There are exactly two vertices x, y in C whose distance from v is $kmin = \lfloor lmc/2 \rfloor$. Thus, $x, y \in F(v)$ and $(x, y) \in E'$. Therefore, (v', x, y) is a triangle in G'. Q.E.D.

COROLLARY. *If a triangle in G' passes through a vertex $x \in V$ then there exists a minimum circuit of G through x.*

Proof. If the triangle consists solely of vertices of V then the triangle is contained in G and is a minimum circuit (because parallel edges and self loops have been excluded). If the triangle contains a vertex of V' then this follows from the proof of Lemma 3. Q.E.D.

Finding a triangle in G' provides us with an edge $(x, y) \in E$ which is contained in a minimum circuit of G. The circuit itself may be found in $O(n)$ time by an algorithm similar to *FRONT*.

5. Algorithms for finding triangles. We study several algorithms for finding triangles.

5.1. Search by rooted spanning trees. Let T be a rooted spanning tree of a connected graph. Using the following lemma we may construct an algorithm to check whether the graph contains a triangle.

LEMMA 4. *There exists a triangle which contains a tree edge if and only if there exists a nontree edge (x, y) for which $(father(x), y) \in E$.* (Every edge is checked in both directions.)

Proof. If $(father(x), y) \in E$ then obviously $(x, y, father(x))$ is a triangle.

In the other direction, assume that (x, y, z) is a triangle and (x, z) is a tree edge (without loss of generality $x = father(z)$). Two cases arise: If $(z, y) \notin T$ then the condition is met for this edge since $(father(z), y) = (x, y) \in E$. Otherwise, $(z, y) \in T$. In this case $z = father(y)$ (each vertex has at most one father). The condition is met for the nontree edge (y, z) since $(father(y), x) = (z, x) \in E$. Q.E.D.

For each nontree edge (x, y) we can check whether $(father(x), y) \in E$ in constant time using the adjacency matrix. Consequently, in time $O(e)$ we may check whether there exists a tree edge which belongs to a triangle.

Let us call a connected component *trivial* if it is an isolated vertex. We may now describe the procedure *TREE*:

procedure *TREE*;

1. Find a rooted spanning tree for each nontrivial connected component of G;
2. If any tree edge is contained in a triangle the algorithm terminates;
3. Delete the tree edges from G.

Each iteration of *TREE* requires at most $O(e)$ time.

procedure *TRIANGLE*;

Repeat *TREE* until all edges of G are deleted.

THEOREM 2. *For planar graphs TRIANGLE requires at most $O(n)$ time.*

Proof. *TRIANGLE* deletes edges from the graph. We first show that each iteration of *TREE* deletes at least a third of the remaining edges. At first $e \leqq 3n - 6$ and we delete $n - 1$ edges; $(n - 1) \geqq e/3$. At subsequent iterations a third of the edges of each connected component are deleted. Therefore, a third of the remaining edges are deleted. Consequently, the number of edges at the ith iteration is at most $(\frac{2}{3})^{i-1}e$. The work in the ith stage is proportional to the number of remaining edges. Therefore, the total work is proportional to $\sum_{i=1}^{\infty} e(\frac{2}{3})^{i-1} = 3e = O(n)$. Q.E.D.

THEOREM 3. *For any graph TRIANGLE requires at most* $O(e^{3/2})$ *time.*

Proof. Let c denote the number of connected components. During the execution of *TRIANGLE* the value of c increases. Initially $c = 1$. At first we estimate the time required by *TRIANGLE* while $c \leqq n - e^{1/2}$. Then we estimate the time while $c > n - e^{1/2}$:

(a) $$c \leqq n - e^{1/2}.$$

Each iteration of *TREE* causes the deletion of $n - c \geqq n - (n - e^{1/2}) = e^{1/2}$ edges. Since there are e edges there may be at most $e/e^{1/2} = e^{1/2}$ such iterations.

(b) $$c > n - e^{1/2}.$$

The degree of each vertex is at most $n - c \leqq n - (n - e^{1/2}) = e^{1/2}$. Since each iteration of *TREE* decreases the degree of each nonisolated vertex, there may be at most $e^{1/2}$ such iterations.

Therefore, we have at most $2e^{1/2}$ iterations in the entire process. Each iteration takes $O(e)$ time. Thus, *TRIANGLE* takes $O(e^{1/2})O(e) = O(e^{3/2})$ time. Q.E.D.

For $K_{n,n}$ (the full bipartite graph with $2n$ vertices) the algorithm may take $O(e^{3/2})$ time while $c \leqq n - e^{1/2}$. For the graph obtained by adding m vertices all connected to a single vertex of $K_{m,m}$ $O(e^{3/2})$ time is required while $c > n - e^{1/2}$.

5.2. Search by vertices. G contains a triangle if there exists a vertex v and an edge a between two vertices u, $w (u < w)$ of $UA(v)$.

```
procedure VERTEX;
for v ∈ V do
  begin a := EDGE(UA(v));
        if a ≠ nil then return(v)
  end
```

EDGE requires that $UA(v)$ be represented by an ordered linked list; moreover, membership in $UA(v)$ can be determined in constant time using the adjacency matrix.

THEOREM 4. *VERTEX finds a triangle in* $O(n^{5/3})$ *on the average.*

Proof. The proof is based on Lemma 2. When calling $EDGE(UA(v))$, E_S is empty. Therefore, $EDGE(UA(v))$ requires at most $O(n^{2/3})$ time on the average. The result follows since *EDGE* is called at most n times. Q.E.D.

Note, that if the upper adjacency vectors or the adjacency matrix has to be prepared then by the note in the Introduction, the algorithm requires additional $O(e)$ time. In any case, $O(n^2)$ space is required.

5.3. Matrix multiplication. Let M be the adjacency matrix (i.e. $(M)_{u,v} = 1$ if and only if $(u, v) \in E$). Let M^2 be the Boolean multiplication of M with itself. $(M^2)_{u,v} = 1$ if and only if there exists a vertex w such that $(M)_{u,w} = (M)_{w,v} = 1$ (i.e. $(u, w), (w, v) \in E$). If also $(M)_{u,v} = 1$, then (u, v, w) forms a triangle. Let $B = M^2$ **and** M (**and** denotes element-by-element logical and). $(B)_{u,v} = 1$ if and only if a triangle passes through the edge (u, v). Using Strassen's algorithm [10] we may multiply Boolean matrices in $O(n^{\log 7})$ time, thus obtaining an $O(n^{\log 7})$ algorithm.

Combining this algorithm with the reduction of § 4 we obtain an algorithm for finding a minimum circuit that takes at most $O(n^{\log 7})$ time.

6. Finding a minimum dicircuit. In the sequel digraphs, dicircuits and dipaths denote directed graphs, circuits and paths respectively.

The techniques for (undirected) graphs described in the previous sections are not applicable to the problem of finding minimum dicircuits in digraphs. Dicircuits may be

found by n applications of the procedure *DICIRCUIT* described below. This method has worst case behavior $O(ne)$ but $O(n^2 \log n)$ on the average. Another method using Boolean matrix multiplication requires $O(n^{\log 7} \log n)$ time.

6.1. The procedure *DICIRCUIT*. *DICIRCUIT*(v) finds a shortest dicircuit through v. We conduct a directed BFS from v. The queue has the same role as in *FRONT*; *level*(v, u) denotes the length of the shortest dipath from v to u if one exists and **nil** otherwise; *scan* denotes the number of scanned vertices.

```
    procedure DICIRCUIT(v);
    begin for u ∈ V do level(v, u) := nil;
          enqueue(v); level(v, v) := 0; scan := 1;
          while scan < n do
          begin if queue is empty then begin
                                          k(v) := nil;
1.                                        return end;
                u := dequeue;
                for w ∈ A(u) do
                   if w = v then begin k(v) := level(v, u)+1;
2.                                    return end
                   else if level(v, w) = nil then
3.                        begin level(v, w) := level(v, u)+1;
                                enqueue(w);
                                scan := scan + 1 end

          end;
          enqueue(u);
4.        while queue is not empty do
                begin u := dequeue;
                   if (u, v) ∈ E then begin k(v) := level(v, u)+1;
5.                                        return end
                end;
6.        k(v) := nil
    end
```

The procedure may terminate at four points in the program:

(a) Line 1. The queue has become empty. In this case there is no dicircuit through v, so we return with $k(v) =$ **nil**.

(b) Line 2. We have returned to vertex v. In this case we have closed a shortest dicircuit through v whose length is $k(v)$.

(c) Line 5. We have reached all the vertices. In this case we look for the first vertex in the queue which closes a dicircuit. This is done via the adjacency matrix. The vertices of the queue are ordered by nondecreasing value of *level*. Therefore, a dicircuit closed in this stage is indeed a shortest dicircuit through v.

(d) Line 6. There is no edge from the scanned vertices to v. Therefore, there is no dicircuit through v, so we return with $k(v) =$ **nil**.

Even though *DICIRCUIT* may require $O(e+n)$ time, the average performance is somewhat better.

THEOREM 5. *Suppose P is a probability measure on labeled digraphs with n vertices such that digraphs with the same outdegrees are equiprobable. Then DICIRCUIT takes* $O(n \log n)$ *time on the average.*

Proof. *DICIRCUIT* takes most time if it scans all vertices. We may consider only

the time needed to reach all vertices since the additional time (lines 4–5) is $O(n)$. Procedure R of [2] also scans a digraph until all vertices have been reached. The main difference is that R uses a stack while *DICIRCUIT* uses a queue. However, R does not take advantage of any property of the stack not shared by a queue. R is proven to take $O(n \log n)$ time on the average. Thus *DICIRCUIT* has an $O(n \log n)$ average behavior too. Q.E.D.

A shortest dicircuit through v can be found by inserting $father(w) := u$ after line 3. The dicircuit is found by backtracking from the vertex u which closed the dicircuit (lines 2 and 5).

By applying *DICIRCUIT* to all vertices of the digraph a shortest dicircuit may be found in $O(n^2 \log n)$ average time.

6.2. Binary search using matrix multiplication. Let *lmdc* be the length of the minimum dicircuit in G; M the adjacency matrix; D_j the matrix of dipaths of length less than or equal to j. ($(D_j)_{u,v} = 1$ if and only if there exists a dipath of length $1 \leqq l \leqq j$ from u to v.) The matrix D_j has a nonzero element on the main diagonal if and only if $lmdc \leqq j$. (i.e. *lmdc* is the smallest j for which D_j contains a nonzero element on its main diagonal.)

Let $j = i + k$ then $D_j = (D_i D_k)$ **or** M where $D_i D_k$ is Boolean matrix multiplication and **or** is an element-by-element logical or, since $(D_i D_k)_{u,v} = 1$ if and only if there exists a dipath of length l, $2 \leqq l \leqq i + k$. The **or** operation adds the dipaths of length 1.

We compute D_j by the following method:

$$D_1 = M$$

$$D_{2l} = D_l^2 \textbf{ or } M.$$

We compute D_{2^i} until there is a nonzero element on the main diagonal. This happens when $i = \lceil \log lmdc \rceil$. The value of *lmdc* is found by a binary search on j in the region $2^{i-1} < j \leqq 2^i$: First we compute $D_{(2^{i-1}+2^i)/2} = D_{2^{i-2}+2^{i-1}} = D_{2^{i-2}} D_{2^{i-1}}$ **or** M. If the diagonal is all zeros we continue the search in the region $2^{i-1} + 2^{i-2} < j \leqq 2^i$. Otherwise, we continue in $2^{i-1} < j \leqq 2^{i-1} + 2^{i-2}$.

The process requires $2 \log lmdc$ matrix multiplications (i.e. $O(n^{\log 7} \log lmdc) = O(n^{\log 7} \log n)$ time).

The space requirements are $O(n^2 \log lmdc) = O(n^2 \log n)$ since we store $\log lmdc$ matrices.

The minimum dicircuit itself may be found in additional $O(e)$ time by a directed BFS from a vertex v for which $(D_{lmdc})_{v,v} = 1$.

7. Conclusions. Using *FRONT* we have an $O(n^2)$ reduction from the problem of finding a minimum circuit to that of finding a triangle. We have shown a method to find a triangle in $O(n^{5/3})$ average time. However, this itself does not yield an $O(n^2)$ average time algorithm to find a minimum circuit since the graphs obtained by the reduction might have a special structure and do not necessarily satisfy the probabilistic assumptions which led to the $O(n^{5/3})$ average time bound. Fortunately, we can solve the problem directly in $O(n^2)$ time on the average. However, any algorithm which finds a triangle in time greater or equal to $O(n^2)$ implies an algorithm to find a minimum circuit within the same time bound. Consequently, finding triangles by Boolean matrix multiplication leads to an $O(n^{\log 7})$ worst case algorithm to find a minimum circuit.

We have seen several algorithms for finding a triangle. *TRIANGLE* is efficient for sparse graphs (especially for planar graphs). *VERTEX* appears better on the

average but has $O(n^3)$ worst case behavior. Better worst case performance can be achieved by using Boolean matrix multiplication.

A related problem is finding a minimum weighted circuit in a weighted graph. It is unclear to us whether our methods can be modified to answer this problem too.

Acknowledgment. The authors wish to thank Shmuel Katz for making valuable 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] P. A. BLONIARZ, M. J. FISHER AND A. R. MEYER, *A note on the average time to compute transitive closures*, Proc. of the 3rd Int. Colloquium on Automata, Languages and Programming (July 1976), S. Michelson and R. Milner, eds.

[3] P. ERDOS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[4] A. ITAI AND M. RODEH, *Some matching problems*, Proc. of the 4th Int. Colloquium on Automata, Languages and Programming (July 1977), A. Salomaa, ed.

[5] P. ERDÖS, *Graph theory and probability* II, Canad. J. Math., 13 (1961), pp. 346–352.

[6] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[7] L. LOVÁSZ, *On chromatic number of finite set systems*, Acta Math. Acad. Sci. Hungar., 19 (1968), pp. 59–67.

[8] M. L. FRIEDMAN, *New bounds on the complexity of the shortest path problem*, this Journal, 5 (1976), pp. 83–89.

[9] C. L. LIU, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

[10] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13, pp. 354–356.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0004 $01.00/0

GENERATING t-ARY TREES LEXICOGRAPHICALLY*

FRANK RUSKEY†

Abstract. This paper extends the results of Ruskey and Hu (1977) from binary trees to t-ary trees. A t-ary tree on N leaves is represented by the sequence of N level numbers of the leaves of that tree. An algorithm is presented for generating these sequences lexicographically as a list. This algorithm is shown to have an average running time of $O(t)$ per sequence generated. $O(N)$ algorithms are developed for determining the position (ranking) of a sequence in the list, and for producing the sequence corresponding to a given place in the list (unranking). Also, a one-to-one correspondence between t-ary trees and walks on a certain lattice is demonstrated; as well as a close relationship between these walks and the lexicographic ranking.

Key words. t-ary trees, lexicographic order, ranking algorithms, lattice walks

1. Introduction. Recently there has appeared a number of papers dealing with the ordering and ranking of binary trees. Typically, the ordering involves showing the existence of a one-to-one correspondence between binary trees and some other combinatorial object. In [2] G. D. Knott shows a correspondence between binary trees and "tree permutations." The ordering he defines on binary trees corresponds to a lexicographic ordering of tree permutations. He then produces algorithms for the ranking and unranking of tree permutations. A. E. Trojanowski in [7] extends the ordering of [2] from binary trees to k-ary trees and then develops another different ordering of binary and k-ary trees. His second ordering involves a correspondence between binary trees and what may be called "stack permutations" (in the sense of exercises 2.2.1–5 and 2.3.1–6 of [3]). The ordering is the lexicographic ordering of stack permutations. Another correspondence, this time between binary trees and "2-permutations," is demonstrated by D. Rotem in [5]. Reportedly, D. Rotem has also developed ranking and unranking algorithms for binary trees in his Ph.D. thesis.

The purpose of this paper is to extend the result of Ruskey and Hu [6] from binary trees to t-ary trees. The correspondence in [6] is between binary trees and "feasible sequences." A feasible sequence is just the sequence of level numbers of the leaves of a binary tree when read from left to right. In the earlier paper [6] we developed algorithms for generating all feasible sequences lexicographically as a list, gave algorithms for ranking and unranking of feasible sequences, and proved that the average running time of the generating algorithm was $O(1)$ per sequence generated. Here we generalize these results to t-ary trees and in addition, analyze the running time of the ranking and unranking algorithms and demonstrate a one-to-one correspondence between t-ary trees and walks on a certain lattice. In [5] a correspondence is given between binary trees and this same lattice. Note that all of the orderings considered so far are essentially lexicographic.

We begin by introducing the relevant terminology. A *t-ary tree*, J, can be defined recursively as being either a *leaf node*, L, or an *internal node*, I, together with a sequence $J_1, J_2, \cdots, J_t$ of t-ary trees. J_i is referred to as the ith *subtree* of I. In the literature t-ary trees are also known as extended t-ary trees [3] or $(t+1)$-valent planted plane trees [1]. Hereafter, tree will mean a t-ary tree as defined above. From the definition it is clear that if a t-ary tree has N leaves (leaf nodes) then $N = n(t-1)+1$ where n is the number of internal nodes. In the computer a t-ary tree is

* Received by the editors May 9, 1977, and in revised form February 10, 1978.

† Department of Applied Physics and Information Science, University of California, San Diego, La Jolla, California 92093. This research was supported by the National Science Foundation under Grant DCR75-06270 and U.S. Army Research Office under Contract DAAG29-76-G-0031.

represented by regarding each node as a record containing t ordered links; if the node is a leaf then all links are set to nil, and if the node is an internal node then the ith link points to the ith subtree. A special pointer, *root*, points to the first internal node created when applying the recursive part of the t-ary tree definition (this node is the *root node*). We draw t-ary trees in the usual way. For example, Fig. 1 shows the 12 trinary (3-ary) trees on 7 leaves.

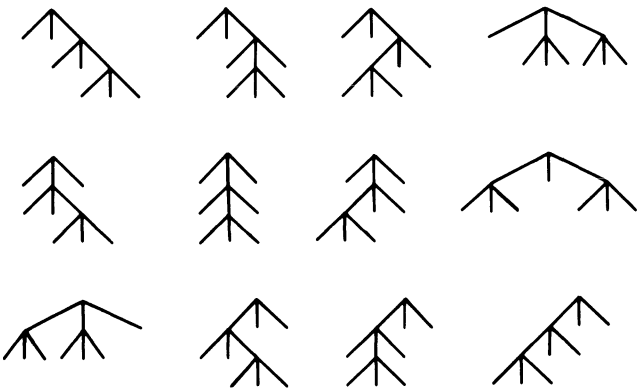


FIG. 1. *The trinary trees with* 3 *internal nodes.*

The number of internal nodes on the path from the root to a leaf is the *level number* of that leaf. If we traverse a t-ary tree from left to right and record the level numbers of the leaves, then we get a sequence of integers $a_1, a_2, \cdots, a_N$. This sequence characterizes the tree it came from. There exist $O(N)$ algorithms for going from the level number sequence to the computer representation of the tree and vice versa. These are easy modifications of the algorithms ComputeDepths and MakeTree for the case $t = 2$ of the earlier paper [6].

The number of t-ary trees on $N = n(t-1)+1$ leaves is well-known [1], [3]. It is

$$T_n = \frac{1}{(t-1)n+1}\binom{tn}{n} = \frac{1}{tn+1}\binom{tn+1}{n} = \sum_{\substack{\mu_1+\mu_2+\cdots+\mu_t=n-1 \\ \mu_i \geq 0}} T_{\mu_1} \cdots T_{\mu_t}.$$

This paper develops an algorithm for generating all of the T_n t-ary trees on N leaves. We represent a t-ary tree by its sequence of level numbers, and generate the sequences lexicographically. The trees in Fig. 1 are listed lexicographically. The algorithm is easily modified to generate not only the sequences but also the computer representations of the trees. This algorithm has an average running time of $O(t)$ per sequence generated. Also, $O(N)$ algorithms are presented for the ranking and unranking of t-ary trees. The running time of Trojanowski's [7] ranking and unranking algorithms is $O(N^2)$. In § 4 we show the close relation between our algorithms and walks on a certain lattice.

2. The algorithm. Many of the results of this section are simple modifications of the corresponding results for $t = 2$ contained in [6]. The ideas behind their proofs are essentially identical and the proofs will therefore be omitted.

Not every sequence of $N = n(t-1)+1$ positive integers represents the level numbers of a t-ary tree with N leaves. Those which do are called *feasible sequences.*

We also refer to $a_1, a_2, \cdots, a_M$ as a *feasible initial sequence for* N if there exist integers $a_{M+1}, \cdots, a_N$ such that $a_1, a_2, \cdots, a_N$ is a feasible sequence. Now suppose there is a smallest integer k $(k \leqq N)$ such that $a_{k-t+1} = a_{k-t+2} = \cdots = a_k = q$. The process of replacing $a_{k-t+1}, a_{k-t+2}, \cdots, a_k$ by $q-1$ to get a new sequence $a_1, a_2, \cdots, a_{k-t}, q-1, a_{k+1}, \cdots, a_N$ is called a *reduction from the left.* On a t-ary tree what this process amounts to is finding the leftmost internal node whose sons are all leaves, deleting those t leaves and replacing that internal node by a leaf. Continuing this reducing process until no further reductions from the left are possible, we get a final sequence called the *left reduced sequence* for $a_1, a_2, \cdots, a_N$. *Reduction from the right* and *right reduced sequence* are defined analogously, except that we take the largest integer k such that $a_{k-t+1} = a_{k-t+2} = \cdots = a_k$.

LEMMA 0. *A necessary condition for a sequence* $a_1, a_2, \cdots, a_N$ *to be feasible is that*

$$\sum_{i=1}^{N} t^{-a_i} = 1.$$

LEMMA 1. *A sequence* $a_1, a_2, \cdots, a_N$ *is feasible if and only if* N *reductions from the right or left (in any order) reduce the original sequence to the single integer* 0.

Let $a_1, a_2, \cdots, a_N$ be a feasible sequence. If $r_1, r_2, \cdots, r_L$ is the left reduced sequence for $a_1, a_2, \cdots, a_{M-1}$ then the procedure REDUCE given below returns the left reduced sequence for $a_1, a_2, \cdots, a_M$ (again as a sequence $r_1, r_2, \cdots, r_L$). The variables L and l are global to REDUCE and will be used in later procedures which call REDUCE as a subroutine. Initially, if $M = m(t-1)+j$ $(1 \leqq j \leqq t-1)$ then $L = l(t-1)+j-1$ for some integer $l \leqq m$ where $m-l$ is the number of reductions used to get from $a_1, a_2, \cdots, a_M$ to $r_1, r_2, \cdots, r_L$.

```
procedure REDUCE (r_1, r_2, ..., r_L, a_M)
    begin
    L ← L+1; r_L ← a_M;
    while r_{L-t+1} = r_L do
        begin
        L ← L-t+1
        l ← l-1;
        r_L ← r_L-1;
        end;
    end of REDUCE
```

This procedure will be used later in the ranking and unranking algorithms. Let $a_1, a_2, \cdots, a_N$ be a sequence of integers; we then define $E(b) = |\{1 \leqq i \leqq N: a_i = b\}|$. In other words, $E(b)$ is the number of integers in the sequence that are equal to b. We say that a sequence of positive integers $a_1, a_2, \cdots, a_N$ is *t-increasing* if $a_1 \leqq a_2 \leqq \cdots \leqq a_N$ and $E(i) < t$ for all $i \geqq 1$. *t-Decreasing* is defined similarly. For example, 1, 1, 3, 3, 3, 6, 6 is 4-increasing but is not 3-increasing.

LEMMA 2. *Let* $a_1, a_2, \cdots, a_N$ *be a sequence of positive integers and suppose that there is a* k *such that* $a_1, a_2, \cdots, a_{k-1}$ *is a t-increasing sequence and* $a_k, a_{k+1}, \cdots, a_N$ *is a t-decreasing sequence; then* $a_1, a_2, \cdots, a_N$ *is a feasible sequence if and only if*

(i) $E(n) = t$ *and*

(ii) $E(1) = E(2) = \cdots = E(n-1) = t-1$.

Proof. See [6].

Note that the tree represented by the feasible sequence of the above lemma is one of maximum height (n) among all t-ary trees with N leaves. In the next lemma we are replacing the rightmost leaf by an internal node, all of whose sons are leaves.

LEMMA 3. *If $a_1, a_2, \cdots, a_N$ is a feasible sequence then so is*

$$a_1, a_2, \cdots, a_{N-1}, \underbrace{a_N+1, a_N+1, \cdots, a_N+1}_{t}.$$

The following tells us how many nodes we need to complete a t-ary tree, given the level number of the first M nodes.

THEOREM 1. *A sequence $a_1, a_2, \cdots, a_M$ $(M<N)$ is a feasible initial sequence for N if and only if its left reduced sequence $r_1, r_2, \cdots, r_L$ satisfies the following conditions (where $m-l$ is the number of reductions used to get the left reduced sequence):*

(i) $1 \leqq r_1$ *and* $r_1, r_2, \cdots, r_L$ *is t-increasing*

(ii) $r_L \leqq n-m+l$.

Proof. See [6].

COROLLARY. *Let $a_1, a_2, \cdots, a_{M-1}$ be a feasible initial sequence for N $(M<N)$ with left reduced sequence $r_1, r_2, \cdots, r_L$; then $a_1, a_2, \cdots, a_{M-1}, a_M$ is a feasible initial sequence for N if and only if*

Case 1. $r_L = l$: $\quad l+1 \leqq a_M \leqq n-m+l$.

Case 2. $r_L > l$: $\quad r_L \leqq a_M \leqq n-m+l$.

(Again, where $M = m(t-1)+j$ and $L = l(t-1)+j-1$ $(1 \leqq j \leqq t-1)$).

Proof. If the upper bound is not satisfied then the condition (ii) of Theorem 1 is violated. If the lower bound is not satisfied then condition (i) of Theorem 1 is violated. If the a_M lies in the indicated ranges then both conditions of Theorem 1 are satisfied. Q.E.D.

Note that if $M = N$ in the above corollary then $a_M = a_N = r_L$.

Given a feasible sequence $a_1, a_2, \cdots, a_N$ we will develop a procedure for determining the next feasible sequence in our lexicographic listing. The first sequence is

$$\underbrace{1, 1, \cdots, 1}_{t-1}, \underbrace{2, \cdots, 2}_{t-1}, \cdots, \underbrace{n-1, \cdots, n-1}_{t-1}, \underbrace{n, \cdots, n, n}_{t}$$

and the last is the first written in reverse. We can test for the last sequence by noting that it is the only feasible sequence such that $a_1 = n$.

Suppose that we had a t-ary tree and wanted to produce the next one in our lexicographic order. Intuitively, we would wish to leave as much of the left part of the tree as possible unchanged, increase the level number of some leaf by replacing it with an internal node, and finally to readjust the remaining nodes in the right part of the tree to make their sequence of level numbers as lexicographically small as possible while maintaining feasibility. What actually happens is described next.

If we had a t-ary tree and had chosen a leaf to be replaced by an internal node, then to make the tree as lexicographically small as possible the sons of the internal node would all be leaves, as shown in Fig. 2. We call the tree shown in Fig. 2 a *one-tree*. A replacement of the kind mentioned above could only happen if there was a subtree that was a one-tree to the right of the leaf that the one-tree is replacing. To do this replacement in a lexicographically minimal way, we could scan our sequence of level numbers $a_1, a_2, \cdots, a_N$ from the right to left until t equal level numbers $a_{k-t+1} = a_{k-t+2} = \cdots = a_k = q$ were found. We would then do the replacement of the leaf with level number $r = a_{k-t}$ to get a sequence of level numbers that started off

$$a_1, a_2, \cdots, a_{k-t-1}, \underbrace{r+1, r+1, \cdots, r+1}_{t}.$$

Now we have to decide if it is possible to rearrange the remaining leaves and, if so, to do the rearrangement in a lexicographically minimal way. Rearrangement is possible only if $a_{k+1} = a_{k+2} = \cdots = a_{k+(t-1)} = q-1$ and $k+(t-1) \neq N$. For example, consider the 3-ary tree with level numbers 1, 1, 3, 3, 6, 6, 6, 5, 5, 4, 4, 2, 2 shown in Fig. 3(a). The rightmost one-tree has leaves at level $a_5 = a_6 = a_7 = 6$. We move these leaves to become sons of a_4 to get a tree like that shown in Fig. 3(b). The encircled subtree can be rearranged; the lexicographically minimal way is shown in Fig. 3(c). In general let r be the largest integer such at $a_{k+1}, a_{k+2}, \cdots, a_{k+r(t-1)}$ are the leaves of a lexicographically maximal subtree. We then turn the leaves around to make a lexicographically minimal subtree and attach it to the leaf a_N (hence the requirement $k+r(t-1) \neq N$). We now present an ALGOL-like procedure NEXTTREE which, from a feasible sequence $a_1, a_2, \cdots, a_N$, produces the next feasible sequence in the lexicographic ordering. It follows the ideas laid out above.

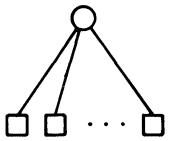


FIG. 2. *A one-tree.*

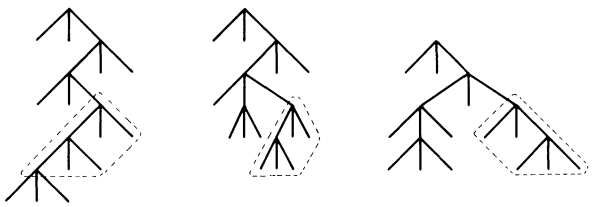


FIG. 3

procedure NEXTTREE ($a_1, a_2, \cdots, a_N$: **sequence**);
begin
$k \leftarrow N$; $R \leftarrow 0$; $r \leftarrow 0$; $I \leftarrow 0$;
while $a_{k-t+1} \neq a_k$ **do** $k \leftarrow k-1$;
comment Now $a_{k-t+1} = a_{k+t-2} = \cdots = a_k$ are the rightmost t equal level numbers;
while $a_{k+R+t-1} = a_k - r - 1$ and $k+R+t-1 \neq N$ **do**
begin
$R \leftarrow R+t-1$;
$r \leftarrow r+1$;
end;
comment $R = r(t-1)$ is now as defined earlier, next we rearrange the level numbers;
for $i \leftarrow 1$ **to** t **do** $a_{k-i} \leftarrow a_{k-t}+1$;
$a_k \leftarrow a_k - r - 1$;
for $i \leftarrow k+1$ **to** $N-R-1$ **do** $a_i \leftarrow a_{i+R}$;
for $i \leftarrow 0$ **to** $r-1$ **do**

begin
for $j \leftarrow 0$ **to** $t-2$ **do** $a_{N-R+I+j} \leftarrow a_N + i + 1$;
$I \leftarrow I + t - 1$;
end;
$a_N \leftarrow a_N + r$;
end of NEXTTREE.

This algorithm will be analyzed in § 6. Note that NEXTTREE uses only additions (no multiplications), a property it shares with the later ranking and unranking algorithms.

3. The ranking algorithm. In this section we will develop an algorithm for determining the rank of any feasible sequence. To do this, a class of numbers will be defined and some of its properties discussed.

In general, a ranking function, f, for an algorithm generating the elements of some set, S, is a bijection $f: S \to \{0, 1, \cdots, |S|-1\}$ such that $f(s) = i$ if and only if the ith element (counting from 0) generated by the algorithm is s. We refer to s as the *rank* i element of S. We wish to find an efficient algorithm for computing f.

If S is a subset of N-tuples of positive integers and the N-tuples are generated in lexicographic order, then one method of determining the rank of an element, $a_1, a_2, \cdots, a_N$, is as follows. Determine for each $k = 1, 2, \cdots, N$ the number, A_k, of elements of S whose first $k-1$ components are $a_1, a_2, \cdots, a_{k-1}$ and whose kth component is one of $1, 2, \cdots, a_k - 1$. The rank of $a_1, a_2, \cdots, a_N$ is then $A_1 + A_2 + \cdots + A_N$. This is the strategy employed below.

The above considerations lead us to ask: How many t-ary trees on $N = n(t-1)+1$ leaves are there whose first $M = m(t-1)+j$ $(1 \leqq j < t)$ level numbers are $a_1, a_2, \cdots, a_M$? Suppose that $a_1, a_2, \cdots, a_M$, after s reductions from the left, left reduces to $r_1, r_2, \cdots, r_L$ where $L = l(t-1)+j = (m-s)(t-1)+j$. The number of t-ary trees on N leaves whose first M level numbers are $a_1, a_2, \cdots, a_M$ is equal to the number of t-ary trees on $(n-s)(t-1)+1$ leaves whose first L level numbers are $r_1, r_2, \cdots, r_L$. Now consider a general t-ary trees on $(n-s)(t-1)+1$ leaves whose first L leaves have level numbers $r_1, r_2, \cdots, r_L$. Such a tree is shown in Fig. 4. The squares represent the first L leaves; the circles are the internal nodes along the path from the root to the Lth leaf; and the triangles represent subtrees whose exact structure is unspecified. There are $r_L(t-1)-L+1$ of these subtrees. They must account for $(n-s)(t-1)-L+1$ leaves and $n-r_L-s$ internal nodes. Thus, there are

$$(1) \qquad \sum_{\substack{\nu_1+\nu_2+\cdots+\nu_{r_L(t-1)-L+1}=n-r_L-s \\ \nu_i=0}} T_{\nu_1} T_{\nu_2} \cdots T_{\nu_{r_L(t-1)-L+1}}$$

t-ary trees on N leaves whose first M level numbers are $a_1, a_2, \cdots, a_M$. We can rewrite (1) as

$$(2) \qquad \sum_{\substack{\nu_1+\nu_2+\cdots+\nu_{(r_L-l)(t-1)-j+1}=(n-m)-(r_L-l) \\ \nu_i \geqq 0}} T_{\nu_1} T_{\nu_2} \cdots T_{\nu_{(r_L-l)(t-1)-j+1}}.$$

Note that (1) and thus (2) depends only on r_L, L and s and not otherwise on the sequence $a_1, a_2, \cdots, a_M$. Introduce the notation

$$(3) \qquad T(n, k, p) = \sum_{\substack{\nu_1+\nu_2+\cdots+\nu_{k(t-1)-p}=n-k \\ \nu_i \geqq 0}} T_{\nu_1} T_{\nu_2} \cdots T_{\nu_{k(t-1)-p}};$$

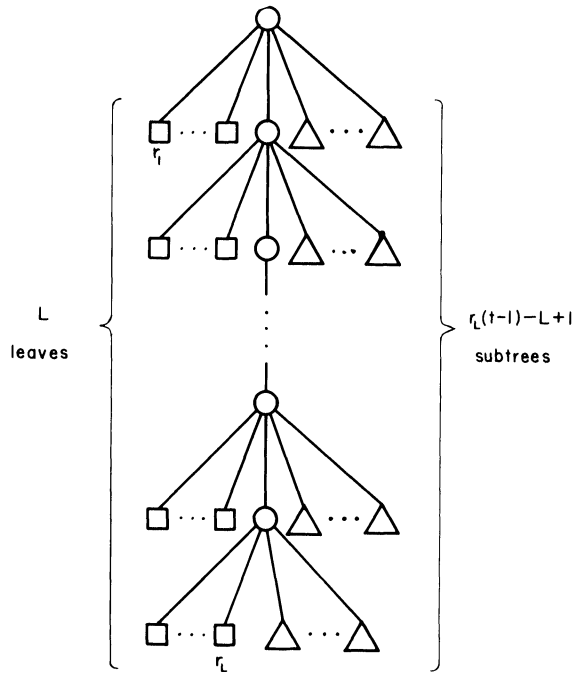


FIG. 4

our earlier sum (2) becomes $T(n-m, r_L-l, j-1)$. We can restrict p so that $0 \leqq p \leqq t-1$. $T(n, k, p)$ can then be interpreted as the number of t-ary trees on N leaves whose first $p+1$ level numbers are k. Using this interpretation we can easily establish the following boundary conditions and recurrence relation ($n \geqq 1$, $0 \leqq k \leqq n$, $0 \leqq p \leqq t-1$):

$$T(n, 0, p) = 0, \tag{4}$$

$$T(n, n, p) = 1, \tag{5}$$

$$T(n, k, t-1) = T(n-1, k-1, 0) \qquad (n > 1, k > 0), \tag{6}$$

$$T(n, k, p) = T(n, k+1, p) + T(n, k, p+1) \qquad (p < t-1, k < n). \tag{7}$$

The equations (4) and (5) are trivial. To prove (6) consider a t-ary tree with N leaves whose first t leaves are at level k. We can left reduce once to get a t-ary tree

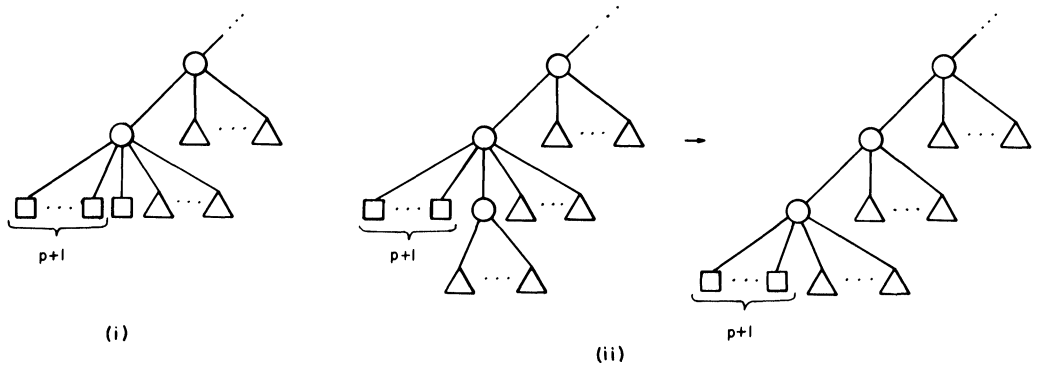


FIG. 5

with $n-1$ internal nodes and first leaf at level $k-1$. Hence, (6) is true. To prove (7) consider a t-ary tree on N leaves whose first $p+1$ leaves are at level k. The closest brother to the right of the $(p+1)$st leaf is either (i) a leaf, or (ii) is the root of t nonempty subtrees. These cases are illustrated in Fig. 5. There are $T(n, k, p+1)$ trees satisfying (i) and $T(n, k+1, p)$ trees satisfying (ii). Thus (7) is established. Note that $T(n, 1, t-2) = T_{n-1}$.

Using (4), (5), (6) and (7) we can easily make tables of the $T(n, k, p)$ using $\frac{1}{2}n(n-1)(t-1)$ additions (for a table up to size n). Tables 1 and 2 give $T(n, k, p)$ for $t=3$ and $n \leqq 5$.

TABLE 1
$p=0$.

n \ k	1	2	3	4	5
1	1				
2	2	1			
3	7	4	1		
4	30	18	6	1	
5	143	88	33	8	1

TABLE 2
$p=1$.

n \ k	1	2	3	4	5
1	1				
2	1	1			
2	3	3	1		
4	12	12	5	1	
5	55	55	25	7	1

There is a simple expression for the $T(n, k, p)$. This expression has been studied before and is presented in Riordan's book [4, p. 49]. The reader is referred there for further references.

THEOREM 2.

$$T(n, k, p) = \frac{tk-k-p}{tn-k-p}\binom{tn-k-p}{n-k}.$$

Proof. First note that

$$\frac{tk-k-p}{tn-k-p}\binom{tn-k-p}{n-k} = \binom{tn-k-p}{n-k} - t\binom{tn-k-p-1}{n-k-1}. \tag{8}$$

We must show that the

$$\frac{tk-k-p}{tn-k-p}\binom{tn-k-p}{n-k}$$

satisfy (4), (5), (6) and (7). Equations (4) and (5) are immediate. Since

$$\frac{tk-k-(t-1)}{tn-k-(t-1)}\binom{tn-k-(t-1)}{n-k} = \frac{t(k-1)-k+1}{t(n-1)-k+1}\binom{t(n-1)-k+1}{n-k},$$

(6) is also true. With the use of (8), verification of (7) becomes a simple application of the basic binomial identity:

$$\binom{tn-k-p-1}{n-k} - t\binom{tn-k-p-2}{n-k-1} + \binom{tn-k-p-1}{n-k-1} - t\binom{tn-k-p-2}{n-k-2}$$

$$= \binom{tn-k-p}{n-k} - t\binom{tn-k-p-1}{n-k-1} \quad \text{Q.E.D.}$$

We can now give a formula for the A_M where $1 \leqq M \leqq N$. Note that $A_N = 0$ since a_N is uniquely determined by the previous level numbers, $a_1, a_2, \cdots, a_{N-1}$. Let $M = m(t-1)+j$ $(1 \leqq j \leqq t-1)$ and suppose that $r_1, r_2, \cdots, r_L$ is the left reduced

sequence for $a_1, a_2, \cdots, a_{M-1}$. Note that $L = l(t-1)+j-1$ for some $l \leqq m$. Let $\mu = \mu_M$ be the smallest integer such that $a_1, a_2, \cdots, a_{M-1}, \mu_M$ is a feasible initial sequence for N. By the corollary to Theorem 1 we have that

$$\mu = \begin{cases} l+1 & \text{if } r_L = l, \\ r_L & \text{otherwise.} \end{cases}$$

It is now clear that

$$A_M = \sum_{i=0}^{a_M - \mu - 1} T(n-m, \mu - l + i, j-1). \tag{9}$$

Example 1. What is the rank of 2, 2, 3, 4, 4, 4, 3, 1, 2, 2, 2? (See Table 3.)

TABLE 3

M	$r_1, r_2, \cdots, r_L$	μ_M	a_M	$a_M - \mu_M - 1$	l	m	$j-1$
1		1	2	0	0	0	0
2	2	2	2	—	0	0	1
3	2, 2	2	3	0	1	1	0
4	2, 2, 3	3	4	0	1	1	1
5	2, 2, 3, 4	4	4	—	2	2	0
6	2, 2, 3, 4, 4	4	4	—	2	2	1
7	2, 2, 3, 3	3	3	—	2	3	0
8	1	1	1	—	0	3	1
6	1, 1	2	2	—	1	4	0
10	1, 1, 2	2	2	—	1	4	1

Thus

$$A_1 + A_2 + \cdots + A_{10} = A_1 + A_3 + A_4$$
$$= T(5, 1, 0) + T(4, 1, 0) + T(4, 2, 1) = 143 + 30 + 12 = 185$$

is the rank.

The following ALGOL-like procedure, RANK, takes as input a feasible sequence $a_1, a_2, \cdots, a_N$ and returns the rank of that sequence in the integer *rank*.

```
procedure RANK (a_1, a_2, ..., a_N: sequence);
    begin
    m ← 1; L ← 0; l ← 0; r_0 ← 0;
    for m ← 0 to n − 1 do
        begin
        for j ← 1 to t − 1 do
            begin
            μ ← if r_L = l then l + 1 else r_L;
            for i ← 0 to a_M − μ − 1 do rank ← rank + T(n − m, μ − l + i, j − 1);
            REDUCE (r_1, r_2, ..., r_L, a_M);
            M ← M + 1;
            end;
        l ← l + 1;
        end;
    end of RANK;
```

Since the **while** loop of REDUCE will be executed at most n times, the only unknown affecting the running time of RANK is the number of times the assignment $rank \leftarrow rank + T(n-m, \mu - l + i, j-1)$ is executed. In the next section we will show that it can be executed at most n times, and therefore that the running time of RANK is $O(n(t-1)) = O(N)$. Note that we assume a table of the $T(n, k, p)$ has already been computed. To compute a table of the $T(n, k, p)$ requires time $O(nN)$; however, given $0!, 1!, \cdots N!$ (which can be computed theoretically in time $O(N)$) and with the use of Theorem 2, a particular value of $T(n, k, p)$ can be computed in time $O(1)$. But because of the explosive nature of $N!$ and the fact that making a table requires only additions, it will be generally preferable to use the first approach (especially if RANK is to be executed repeatedly).

4. A one-to-one correspondence. Let L_t be the set of lattice points defined by

$$L_t = \{(x, y): x, y \geqq 0 \text{ integers}, (t-1)y \leqq x\}.$$

We will consider walks on this lattice where one is confined to move only to the right or upwards, i.e. $(x, y) \leftarrow (x+1, y)$ or $(x, y) \leftarrow (x, y+1)$. L_3 is shown in Fig. 6. Define

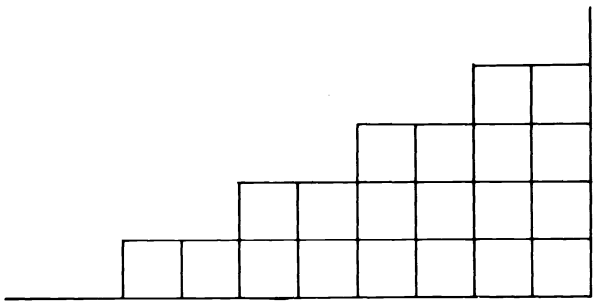


FIG. 6

$L(M, k) = L(m, k, j)$ to be the number of walks from $(0, 0)$ to (M, k) where $M = m(t-1)+j$ $(0 \leqq j < t-1)$. Then we clearly have

$$L(m, 0, j) = 1, \qquad L(m, m, 0) = L(m, m-1, 0),$$

$$L(m, k, j) = L(m, k-1, j) + L(m, k, j-1).$$

These are essentially the relations describing the $T(n, k, p)$. Indeed, $L(m, k, j) = T(m+1, m-k+1, t-j-2)$. In particular, $L(n, n, 0) = T(n+1, 1, t-2) = T_n$. Thus there is a one-to-one correspondence between t-ary trees on N leaves and walks on L_t starting at $(0, 0)$ and ending at $(n(t-1), n)$. We will now make explicit this one-to-one correspondence by showing how it relates to our ranking scheme. First, note that

$$\sum_{k \leqq k'} L(M, k) = L(M+1, k') \tag{10}$$

because any walk to $(M+1, k')$ must pass through one of the points $(M, 0), (M, 1), \cdots, (M, k')$, and after passing through that point has only one way of continuing to $(M+1, k')$. Using (10) we can prove the following

LEMMA. *Given any integer K such that $0 \leqq K < L(M, k)$ there exist unique integers $0 = k_0 \leqq k_1 \leqq \cdots \leqq k_M = k$ such that*

$$K = \sum_{i=0}^{M-1} \sum_{r=k_i+1}^{k_{i+1}} L(i, r). \tag{11}$$

Proof (by induction on M). If $M = 1$ then we must have $k = k_0 = k_1 = K = 0$, and the lemma is true by the empty sum convention. Otherwise assume that the lemma is true up to $M-1$. Let k_{M-1} be the smallest nonnegative integer such that

$$\sum_{r=k_{M-1}+1}^{k_M} L(M-1, r) \leqq K.$$

Let $\alpha = K - \sum_{r=k_{M-1}+1}^{k_M} L(M-1, r)$. Then $0 \leqq \alpha < L(M-1, k_{M-1})$, so by the inductive assumption there exist unique integers $0 = k_0 \leqq k_1 \leqq \cdots \leqq k_{M-1}$ such that

$$\alpha = \sum_{i=0}^{M-2} \sum_{r=k_i+1}^{k_{i+1}} L(i, r).$$

Now we need only show that the choice of k_{M-1} was unique. Certainly we could not have chosen k_{M-1} any smaller. If we had chosen k_{M-1} larger than $\alpha \geqq L(M-1, k_{M-1})$. But by (10), the largest $\sum_{i=0}^{M-2} \sum_{r=k_i+1}^{k_{i+1}} L(M-1, r)$ could be is $L(M-1, k_{M-1}) - L(M-1, 0) = L(M-1, k_{M-1}) - 1$; so there is no way to choose the remaining k_i. Q.E.D.

The integers $k_0, k_1, \cdots, k_M$ define a unique walk on L_t passing through the points $(0, k_0), (1, k_0), (1, k_1), (2, k_1), \cdots, (M, k_M)$. Note that $k_i \leqq \lfloor i/(t-1) \rfloor$. The converse of the lemma also holds; i.e., given integers $0 = k_0 \leqq k_1 \leqq \cdots \leqq k_M = k$ such that $k_i \leqq \lfloor i/(t-1) \rfloor$ $(0 \leqq i \leqq M)$, then (11) satisfies $0 \leqq K < L(M, k)$. To find the walk corresponding to a t-ary tree we could first find its rank by using the procedure RANK, and then determine the $k_0, k_1, \cdots, k_{M-1} = n$ for $K = rank$ and thus produce a walk. Surprisingly, however, the $k_0, k_1, \cdots . k_{N-1}$ are determined already in RANK by the relation $k_{N-1} = n$ and

$$k_M - k_{M-1} = \begin{cases} a_{N-M} - \mu_{N-M} + 1 & \text{if } M \equiv 0 \pmod{t-1} \text{ and } k_M = m, \\ a_{N-M} - \mu_{N-M} & \text{otherwise.} \end{cases} \tag{12}$$

Example 2. What walk corresponds to the tree of Example 1? We find there that $k_0, k_1, \cdots, k_{10} = 0, 0, 1, 1, 1, 1, 1, 2, 3, 3, 5$ and the walk therefore looks like Fig. 7.

If we label the edge from $(M+1, k-1)$ to $(M+1, k)$ with $L(M, k)$ and then sum the labels on any walk from $(0, 0)$ to $(N-1, n)$ then we get the rank of the tree corresponding to that walk. The labels are shown in Fig. 7; the above walk has rank

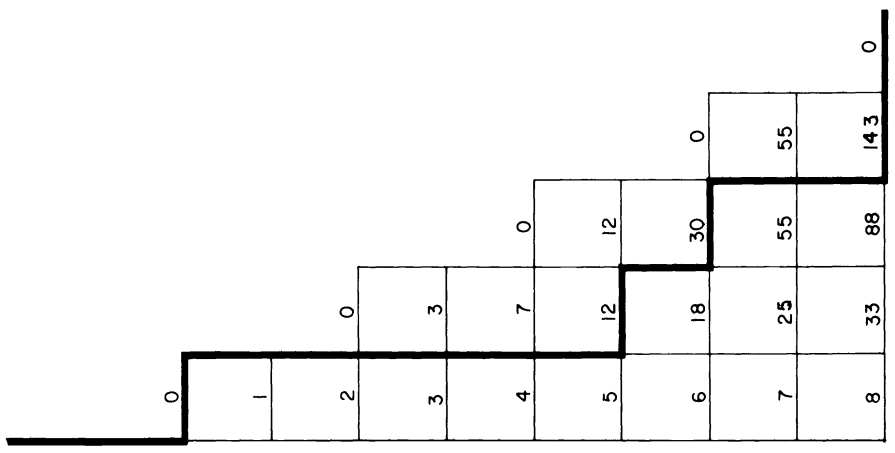


FIG. 7

$L(9,5)+L(9,4)+L(7,3)+L(6,2)+L(1,1)=0+143+30+12+0=185$. These are just the $T(n,k,p)$ that were added in RANK. This shows that the number of times the assignment $rank \leftarrow rank + T(n-m, \mu-l+i, j-1)$ can be executed is at most n (i.e. the number of vertical edges used in a walk from $(0,0)$ to $(N-1,n)$). We now present an algorithm for producing the t-ary tree having a given rank.

5. The unranking algorithm. To find the uth tree on N leaves we could proceed as follows. First, find the walk from $(0,0)$ to $(N-1,n)$ corresponding to u; then from the $k_0, k_1, \cdots, k_{N-1}$ determined by that walk try to find the a_M and μ_M for $1 \leqq M \leqq N$.

Suppose we try to find the 200th trinary tree on 11 leaves. We easily find the path in Fig. 6 by starting at (10, 5) and proceeding down as far as possible (to (10, 3))

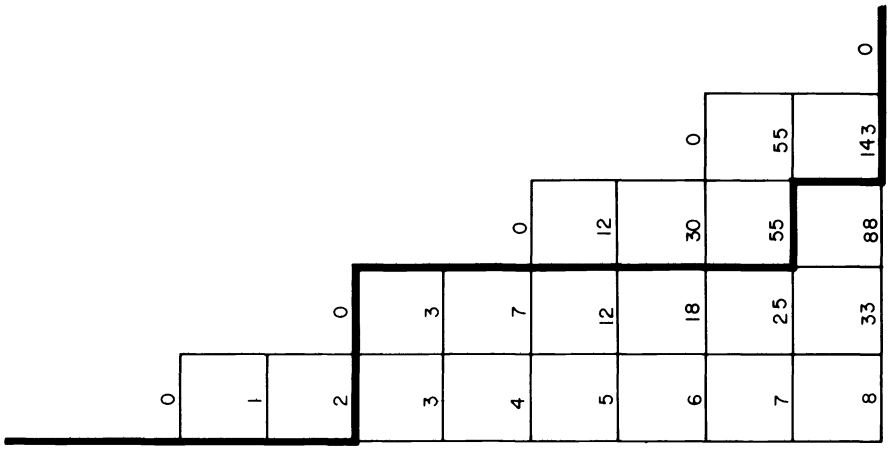


FIG. 8

without exceeding 200. Now go one step to the left and repeat the process. (See Fig. 8.) Thus, $k_0, k_1, \cdots, k_{10} = 0,0,0,0,2,2,2,2,2,3,5$. Rewriting (12) we have

$$a_{N-M} - \mu_{N-M} = \begin{cases} k_M - k_{M-1} - 1 & \text{if } M \equiv 0 \pmod{t-1} \text{ and } k_M = m, \\ k_M - k_{M-1} & \text{otherwise.} \end{cases}$$

Since $\mu_1 = 1$ we have $a_1 = 1 + k_{10} - k_9 - 1 = 2$. Thus $\mu_2 = 2$; so $a_2 = 2 + k_9 - k_8 = 3$. This can be continued until we get the sequence of level numbers 2, 3, 3, 3, 2, 1, 3, 3, 3, 2.

However, we do not have to find the walk and the k_i to unrank u. The following ALGOL-like procedure UNRANK takes as input the rank of the desired tree and n the number of internal nodes of the tree ($0 \leqq rank < T_n$), and returns the feasible sequence $a_1, a_2, \cdots, a_N$ having that rank. It essentially reverses the steps used in the RANK procedure. UNRANK also has running time $O(N)$, for the same reasons that RANK had running time $O(N)$.

```
procedure UNRANK (n, rank: integer);
    begin
    M ← 1; L ← 0; r0 ← 0: l ← 0;
    for m ← 0 to n − 1 do
        begin
        for j ← 1 to t − 1 do
```

```
            begin
            μ ← if r_L = l then l + 1 else r_L;
            i ← 0; tsum ← 0;
            repeat
                tsum ← tsum + T(n − m, μ − l + i, j − 1);
                i ← i + 1;
            until rank < tsum;
            rank ← rank − tsum + T(n − m, μ − l + i − 1, j − 1);
            a_M ← μ + i − 1;
            REDUCE (r_1, r_2, · · · , r_L, a_M);
            M ← M + 1;
            end;
        l ← l + 1;
        end;
    end of UNRANK.
```

6. Analysis of the tree generating algorithm. The time required for one iteration of NEXTTREE is proportional to the number, p of leaves from the right until t leaves at the same level are encountered. In the worst case this may be $O(N)$, but on the average it is on the order of

$$\frac{1}{T_n} \sum_{p=0}^{(n-1)(t-1)} (p+1) I(n, p), \tag{13}$$

where $I(n, p)$ is the number of t-ary trees on N leaves whose first p level numbers form a t-increasing sequence and such that $a_{p+1} = a_{p+2} = \cdots = a_{p+t}$ and $a_p < a_{p+1}$.

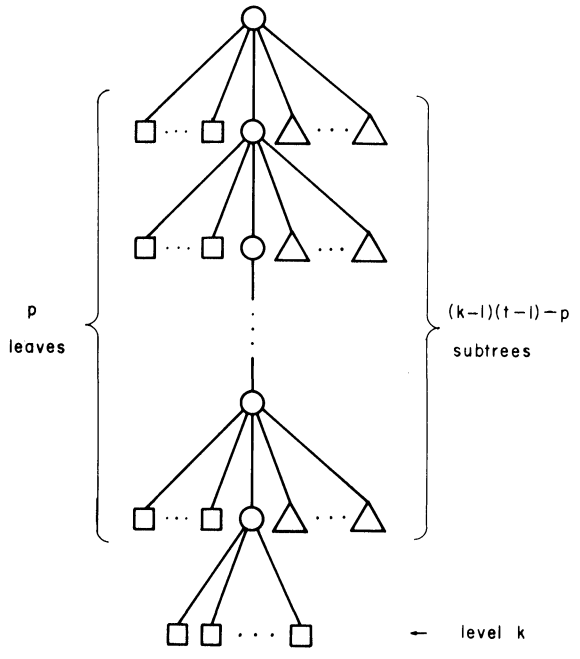


FIG. 9

Referring to Fig. 9, we see that $I(n, p)$ can be written as

$$\sum_{k=1}^{n} \sum_{\substack{\mu_1+\mu_2+\cdots+\mu_{k-1}=p \\ 0\leqq\mu_i\leqq t-1}} \sum_{\substack{\nu_1+\nu_2+\cdots+\nu_{(k-1)(t-1)-p}=n-k \\ \nu_i\geqq 0}} T_{\nu_1} T_{\nu_2} \cdots T_{\nu_{(k-1)(t-1)-p}}$$

$$= \sum_{k=1}^{n} \sum_{\substack{\mu_1+\mu_2+\cdots+\mu_{k-1}=p \\ 0\leqq\mu_i\leqq t-1}} T(n-1, k-1, p) \tag{14}$$

where μ_i is the number of the first p leaves that are at level i and $a_{p+1} = a_{p+2} = \cdots = a_{p+t} = k$.

The $I(n, p)$ satisfy the following boundary conditions and recursion

$$I(n, (n-1)(t-1)) = 1, \tag{15}$$

$$I(n, 0) = I(n, 1) = \cdots = I(n, t-1) = T_{n-1}, \tag{16}$$

$$I(n, p) = I(n-1, p-t+1) + I(n, p+1). \tag{17}$$

These relations will all be given combinatorial proofs. The one tree indicated by (15) is the lexicographically smallest tree with N leaves. To prove (16) we will exhibit a one-to-one correspondence between T_{n-1} and $I(n, p)$ where $0 \leqq p < t$. Consider a tree in $I(n, p)$; if we perform one reduction from the left then we get a tree in T_{n-1}. Conversely, given a tree in T_{n-1} we can replace the $(p+1)$st leaf from the left with a one-tree, yielding a tree in $I(n, p)$ (if $p \geqq t$ then we cannot be assured that the tree is in $I(n, p)$). Combining (14) and (16) results in the identity (if $0 \leqq p < t$)

$$T_n = \sum_{k=1}^{n} \binom{k+p-1}{p} T(n, k, p).$$

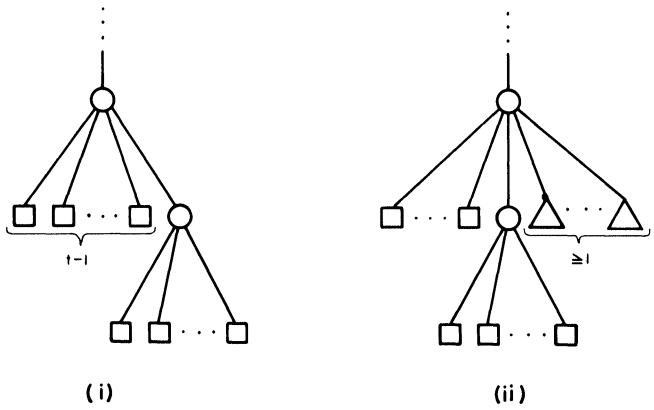


FIG. 10

To prove (17) consider a tree in $I(n, p)$. One of two cases occurs: either the father of the $(p+1)$st leaf (i) is the tth son of his father or (ii) is not the tth son of his father. These cases are illustrated in Fig. 10. If we reduce $a_{p+1}, \cdots, a_{p+t}$ in case (i) then we get a tree in $I(n-1, p-t+1)$; and vice versa. Case (ii) is only slightly more difficult to visualize. Here we reduce $a_{p+1}, \cdots, a_{p+t}$ but then replace the leaf with level number a_{p+t+1} by a one-tree. This yields a tree counted by $I(n, p+1)$. This procedure can be reversed to get a tree satisfying case (ii). The $I(n, p)$ also count the number of walks on

a set of lattice points that might be described as the "dual" of L_t. It is the lattice

$$L_t' = \{(x, y)\colon x, y \geqq 0 \text{ integers}, y \leqq (t-1)x\}$$

which is shown in Fig. 11 for $t = 3$. The number at a lattice point counts the number of walks from (0, 0) to that point. The dotted lines and vertical lines shows how the

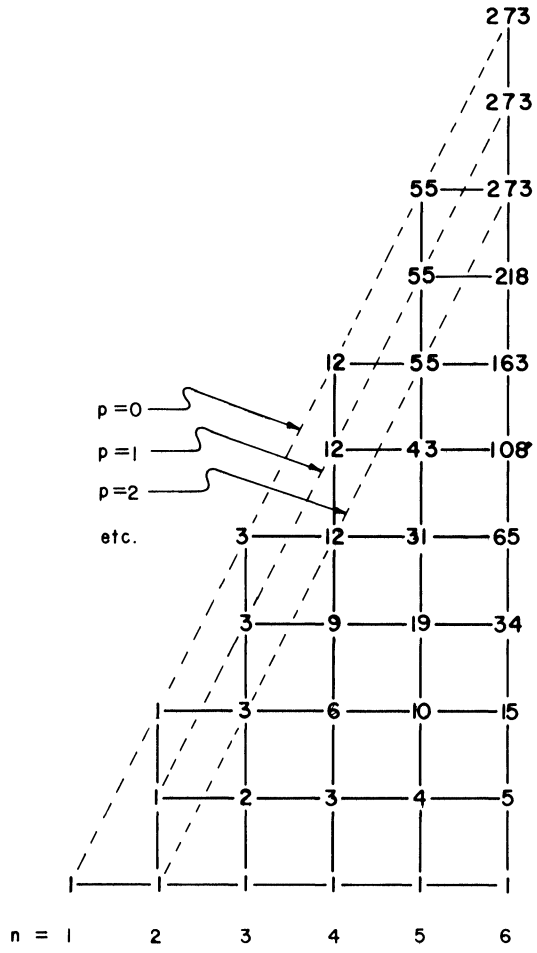


FIG. 11

number of walks relates to the $I(n, p)$. Iterating (17) results in

$$\sum_{j=p}^{(n-1)(t-1)} I(n, j) = I(n+1, p+t-1).$$

Thus

$$\begin{aligned}\sum_{p \geqq 0} (p+1)I(n, p) &= \sum_{j \geqq 0} I(n, j) + \sum_{j \geqq 1} I(n, j) + \cdots + \sum_{j \geqq (n-1)(t-1)} I(n, j) \\ &= I(n+1, t-1) + I(n+1, t-1+1) + \cdots + I(n+1, n(t-1)) \\ &= \sum_{j \geqq t-1} I(n+1, j) = I(n+2, 2(t-1)).\end{aligned}$$

However,

$$\begin{aligned} I(n+2, 2(t-1)) &= I(n+2, 2(t-1)-1) - I(n+1, t-2) \\ &= I(n+2, 2(t-1)-1) - T_n \\ &= I(n+2, 2(t-1)-2) - 2T_n \\ &= I(n+2, t-1) - (t-1)T_n = T_{n+1} - (t-1)T_n. \end{aligned}$$

Thus we have

$$\sum_{p \geqq 0} (p+1) I(n, p) = T_{n+1} - (t-1)T_n.$$

Thus the average running time of NEXTTREE is $O(T_{n+1}/T_n - t)$. To get a better idea of what this is we estimate T_n/T_{n-1}.

$$\begin{aligned} T_n = \frac{1}{(t-1)n+1}\binom{tn}{n} &= \frac{1}{(t-1)n+1} \frac{(tn)!}{n!(tn-n)!} \\ &= \frac{(t-1)(n-1)+1}{(t-1)n+1} \frac{1}{n} \frac{(tn)(tn-1)\cdots(tn-(t-1))}{(tn-n)(tn-n-1)\cdots(tn-n-(t-2))} T_{n-1} \\ &\leqq \frac{tn}{n} \frac{(tn)^{t-1}}{(tn-n-t+2)^{t-1}} T_{n-1} \\ &\leqq t\left(\frac{(\tau+1)(\eta+1)}{\tau\eta}\right)^{\tau} T_{n-1} \quad \text{where } \tau = t-1 \text{ and } \eta = n-1 \\ &\leqq t\left(1 + \frac{\tau+\eta+1}{\tau\eta}\right)^{\tau} T_{n-1} \\ &\leqq t \exp\left(\frac{\tau+\eta+1}{\eta}\right) T_{n-1} \leqq te^3 T_{n-1} \quad \text{if } n \geqq t. \end{aligned}$$

Hence NEXTTREE has a running time on the average of $O(t)$. Also note that because NEXTTREE contains the loop

for $i \leftarrow 1$ **to** t **do** $a_{k-i} \leftarrow a_{k-t} + 1$;

then the average running time must be at least $O(t)$. Because the $T(n, k, p)$ had such a simple expression, it is plausible that the $I(n\,p)$ also have a simple expression. However, the author was unable to find one. The reason that it was possible in the $t = 2$ case [6] is that $L_2 = L_2'$.

REFERENCES

[1] D. A. KLARNER, *Correspondences between plane trees and binary sequences*, J. Combinatorial Theory, 9 (1970), pp. 401–411.

[2] G. D. KNOTT, *A numbering system for binary trees*, Comm. ACM, 20 (1977), pp. 113–115.

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

[4] J. RIORDAN, *Combinatorial Identities*, John Wiley, New York, 1968.

[5] D. ROTEM, *On a correspondence between binary trees and a certain type of permutation*, Information Processing Lett., 4 (1975), pp. 58–61.

[6] F. RUSKEY AND T. C. HU, *Generating binary trees lexicographically*, this Journal, 6 (1977), pp. 745–758.

[7] A. E. TROJANOWSKI, *On the ordering, enumeration and ranking of k-ary trees*, preprint, Computer Science Dept., Univ. of Illinois, Urbana-Champaign, 1977.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0005 $01.00/0

POLYNOMIAL TIME ENUMERATION REDUCIBILITY*

ALAN L. SELMAN†

Abstract. A viable polynomial time enumeration reducibility is defined and studied. Let $\leqq_{pe}$ denote this reducibility. $\leqq_{pe}$ is intrinsic to certain tradeoffs between nondeterministic oracle recognition of sets and deterministic oracle computations between functions. A set belongs to $\mathcal{NP}$ if and only if the set is, in some natural sense, polynomial enumerable. $\leqq_{pe}$ is defined so that $A \leqq_{pe} B$ just in case for every set C, every polynomial enumeration of B relative to C yields some polynomial enumeration of A relative to C. Various properties of $\leqq_{pe}$ are shown. In particular, $\leqq_{pe}$ is a maximal transitive subrelation of $\leqq_T^{\mathcal{NP}}$. Also, $\leqq_{pe}$ is equal to $\leqq_c^{\mathcal{NP}}$ on low level complexity classes, but the equality does not hold over all recursive sets.

Key words. relative computability, enumeration reducibility, nondeterministic, polynomial time, complexity classes, oracle Turing machines, polynomial enumerable

Introduction. This paper contributes to the growing body of research on computation bounded reducibilities. Let $\mathcal{P}$ ($\mathcal{NP}$) be the class of sets recognized by deterministic (nondeterministic) Turing machines which run in polynmial time. The class $\mathcal{P}$ encodes the collection of problems that are "practically computable." There are known connections between the important $\mathcal{P} = \mathcal{NP}$? question, the question of closure under complements of $\mathcal{NP}$, and the behavior of various polynomial time bounded reducibilities [8]. Thus we are especially interested in questions about relative computability that arise at the level of polynomial time bounded complexity.

Our intent here is to demonstrate the existence of a viable polynomial time enumeration reducibility. One motivation to this research is that a satisfactory definition of polynomial time reducibility between functions, rather than sets, remains elusive. (Work in this direction can be found in [4] and [9]). In recursive function theory, enumeration reducibility contributes to the understanding and "making precise" of relative computability [11], [12]. We introduce the most conservative meaningful notion of polynomial time relative computability between functions. Our work demonstrates in the current context (polynomial time complexity) that relative nondeterministic reducibility between sets can be replaced by relative deterministic computability between functions. In addition, our work on polynomial time enumeration reducibility enables us to focus our attention on a proper definition for nondeterministic relative computability between functions.

The relation "nondeterministic polynomial time in" ($\leqq_T^{\mathcal{NP}}$, in the notation of [8]) has proved useful in the work of Meyer and Stockmeyer [10] and Baker, Gill and Solovay [1]. This relation, however, is not transitive [8]. The greater importance of the deterministic reducibilities, $\leqq_m^{\mathcal{P}}$, and $\leqq_T^{\mathcal{P}}$, of Cook [2] and Karp [5] is, in part, due to their nice properties of reflexivity and transitivity. Thus, $(\leqq_m^{\mathcal{P}}) \cap (\leqq_m^{\mathcal{P}})^{-1}$ and $(\leqq_T^{\mathcal{P}}) \cap (\leqq_T^{\mathcal{P}})^{-1}$ are equivalence relations. It will be easy to show that polynomial time enumeration reducibility (denoted, $\leqq_{pe}$) is a subrelation of $\leqq_T^{\mathcal{NP}}$ which shares these properties. In addition, we will show that $\leqq_{pe}$ possesses the following two desirable properties: $\mathcal{NP}$ is the $\mathbf{0}$ degree for the degree structure of $\leqq_{pe}$, and $\leqq_{pe}$ is as "large" in $\leqq_T^{\mathcal{NP}}$ as possible, i.e., $\leqq_{pe}$ is a maximal transitive subrelation of $\leqq_T^{\mathcal{NP}}$.

Let A and B be two recursive sets of strings. Suppose $A \leqq_T^{\mathcal{NP}} B$ via a nondeterministic oracle Turing machine M. Then, of course, given an input string x, the length of each oracle query x about membership in B is bounded by the running time of M

* Received by the editors January 17, 1977, and in revised form September 27, 1977.

† Department of Computer Science, Iowa State University, Ames, Iowa 50010. This research was supported in part by the National Science Foundation under Grants DCR 75-06340 and MCS 77-23493.

on x. In addition, if for some set C, $B \leqq_T^{\mathcal{NP}} C$, an attempt to recognize A relative to C may require exponential time—thereby making $\leqq_T^{\mathcal{NP}}$ nontransitive—because if the length of z is of order a polynomial of the length of x, then the time required to deterministically determine, relative to C, that a query z does *not* belong to B may be exponential. The definition of $\leqq_{pe}$ will be based on very different fundamental considerations. However, in contrast to the situation just described, in terms of oracle Turing machines it will turn out that $A \leqq_{pe} B$ just in the case that the length of oracle queries that receive "no" answers about membership in B can be made "arbitrarily small." More precisely, for each integer m there is an oracle Turing machine M that witnesses $A \leqq_T^{\mathcal{NP}} B$ and for every x in A there is an accepting computation of M in which every oracle query about membership in B that receives a "no" answer has length at most $O((\log |x|)^{1/m})$. The reader may observe at this point that the property just described is a weakening of the machine that witnesses $A \leqq_c^{\mathcal{NP}} B$. In the latter case, for each input x there is an accepting computation in which no queries receive a "no" answer.

In the following section it is shown that a set A belongs to $\mathcal{NP}$ if and only if A is, in some natural sense, polynomial-enumerable. Moreover, this concept is easily relativized. Relative computability between functions is also discussed in this section. Section 2, then, presents alternative fundamental motivations and definitions of $\leqq_{pe}$, based on the ability to enumerate precisely the sets in $\mathcal{NP}$. In § 3, a principal technical result proves that the alternative definitions of $\leqq_{pe}$ given in § 2 are equivalent—thereby giving evidence to the claim that our definition is correct. Section 4 compares $\leqq_{pe}$, and other maximal transitive subrelations of $\leqq_T^{\mathcal{NP}}$, with the reducibilities studied in Ladner, Lynch and Selman [8]. It is proved that "negative questions" are unnecessary on the interesting low level complexity classes, but that this simplification is not true in general.

Given a function f, let $\mathbf{f}$ denote the graph of f, and let C_A denote the characteristic function of the set A. It is proved in the concluding § 5 that $\leqq_{pe}$ satisfies properties akin to the well-known relationships $f \leqq_T g \leftrightarrow \mathbf{f} \leqq_{pe} \mathbf{g}$ and $A \leqq_T B \leftrightarrow \mathbf{C}_A \leqq_{pe} \mathbf{C}_B$ satisfied by ordinary enumeration reducibility.

We further establish the notation to be used. All sets are intended to be recursive languages over the alphabet $\{0, 1\}$. Numbers are identified with their binary representations. As was used above, the length of a string is denoted $|x|$. Therefore, $|x| \simeq \log x$, the binary logarithm of x. We will let $\sigma = \langle \cdot, \cdot \rangle$ be a fixed pairing function with inverses σ_1 and σ_2 so that σ, σ_1, and σ_2 are each computable in polynomial time. Furthermore, we assume that there is a polynomial s so that, for each x and y, $|\sigma(x, y)| = s(|x|, |y|)$.

Reducibilities will be denoted as in [8]. To summarize:

$\leqq_T^{\mathcal{P}}$ polynomial time Turing reducibility

$\leqq_{tt}^{\mathcal{P}}$ polynomial time truth-table reducibility

$\leqq_c^{\mathcal{P}}$ polynomial time conjunctive reducibility

$\leqq_m^{\mathcal{P}}$ polynomial time many-one reducibility

To each such reducibility $\leqq_r^{\mathcal{P}}$, there is the corresponding nondeterministic polynomial time reducibility $\leqq_r^{\mathcal{NP}}$.

1. Polynomial enumerability.

1.1. Enumerations. Let $\mathcal{L}$ be the set of polynomial time bounded computable functions [3], and let $\mathcal{L}(A)$ be the set of functions computable in polynomial time by

an oracle Turing machine with oracle A. Given a function f and a polynomial p, we will say that f is p-bounded if, for every x, $|f(x)| \leqq p(|x|)$. Clearly, if there is a set A such that $f \in \mathscr{L}(A)$, then, for some polynomial p, f is p-bounded. By the following theorem the converse is also true.

THEOREM 1. *There exists a set A such that $f \in \mathscr{L}(A)$ if and only if there is a polynomial p such that $|f(x)| \leqq p(|x|)$, for each x.*

Proof. Suppose that p is a polynomial and that $|f(x)| \leqq p(|x|)$. We define a set A that encodes the bitwise computation of f. A consists of encoded triples $\langle a, x, k\rangle$, where $a \in \{0, 1\}$, $x \in \{0, 1\}^*$, and $k \in \{0, 1\}^*$ is the binary representation of an integer k, defined according to the following rules:

$\langle 0, x, k\rangle \in A$ if and only if $f(x)$ has a kth bit;

$\langle 1, x, k\rangle \in A$ if and only if the kth bit of $f(x)$ is 1.

It should be clear that f is computable from A. In fact, to show that f is computable from A in polynomial time, observe that in a computation of $f(x)$, $\langle 0, x, |f(x)|+1\rangle$ is the longest query made, and $O(|f(x)|)$ queries are made altogether. Since $|f(x)| \leqq p(|x|)$, it follows that $f \in \mathscr{L}(A)$.

We next define some concepts that have obvious analogies with recursive enumerability.

DEFINITION 1. A function f is a *polynomial-enumerating* (*polynomial-enumerating in A*) function if

(1) f is in $\mathscr{L}$ (f is in $\mathscr{L}(A)$), and

(2) there is a polynomial q so that

$$\forall y[y \in \text{range } f \to \exists x[|x| \leqq q(|y|) \,\&\, y = f(x)]].$$

A set B is *polynomial-enumerable* (*polynomial-enumerable in A*) if $B = \varnothing$ or there is a polynomial-enumerating (in A) function such that $B = \text{range } f$. In this case, f is called a *polynomial-enumeration* (relative to A) of B.

Let us agree that to each many-one function f there is a possible multitude of inverse functions f^{-1} (defined on range f), each of which is determined by the condition that $f^{-1}(y)$ is one of the strings x for which $f(x)=y$. Then, clause (2) of Definition 1 states that *f has a q-bounded inverse.*

Theorem 1 makes it possible to specify the class of all polynomial-enumerating functions without reference to oracles.

DEFINITION 2. Define the class $\mathscr{ENUM}$ to be the collection of all functions f for which there exist polynomials p and q such that

(1) f is p-bounded, and

(2) f has a q-bounded inverse.

COROLLARY 1. *f is polynomial-enumerating in A, for some set A, if and only if f belongs to $\mathscr{ENUM}$.*

In discourse we will frequently use the phrase "polynomial-enumeration" generically. For example, given a nonempty set B, the collection of all polynomial-enumerations of B will refer to the set of all functions f in $\mathscr{ENUM}$ satisfying range $f = B$.

If f is nondecreasing, then the second clause of Definition 2 is automatically satisfied. Observe that not every function can be a polynomial-enumeration, even relative to a set A. For example, the function $f(x)=2^x$ is not p-bounded, for any polynomial p, and the function $f(x) = \log x$ has no q-bounded inverse. Thus 2^x and $\log x$ are not polynomial-enumerations of their ranges relative to any set A. In

contrast, every enumeration of a set B is a recursive enumeration relative to some set A.

Every nonempty set B possesses the *trivial* polynomial-enumeration g (relative to itself) defined by

$$g(x) = \textbf{if } x \in B \textbf{ then } x \textbf{ else } b,$$

where b belongs to B.

Let $\mathcal{P}(A) = \{B \mid B \leqq_T^{\mathcal{P}} A\}$ and let $\mathcal{NP}(A) = \{B \mid B \leqq_T^{\mathcal{NP}} A\}$.

We will make use of the following known characterization of $\mathcal{NP}(A)$ ([2] and [5]) in terms of polynomial time-bounded quantifiers. Namely, B belongs to $\mathcal{NP}(A)$ if and only if there is a polynomial q and a binary relation R in $\mathcal{P}(A)$ so that

$$\forall x[x \in B \leftrightarrow \exists y[|y| \leqq q(|x|) \,\&\, R(x, y)]].$$

A simple padding trick makes it possible to replace the "$\leqq$" sign with equality. We will always assume the equality sign. In this case we refer to q as *the size of the computation* of B (relative to A).

Recall that application of the pairing function σ to x and y has length $s(|x|, |y|)$.

THEOREM 2. *If B belongs to $\mathcal{NP}(A)$ via a computation of size q, then B has a polynomial enumeration f relative to A such that f has a Q-bounded inverse, where $Q(\cdot) = s(\cdot, q(\cdot))$.*

Proof. Assume $B \neq \varnothing$, and let $b \in B$. Suppose

$$\forall x[x \in B \leftrightarrow \exists y[|y| = q(|x|) \,\&\, R(x, y)]],$$

where $R \in \mathcal{P}(A)$.

Define $f(i) = \textbf{if } |\sigma_2(i)| = q(|\sigma_1(i)|) \,\&\, R(\sigma_1(i), \sigma_2(i)) \textbf{ then } \sigma_1(i) \textbf{ else } b$. (Intuitively, $i = \langle x, y\rangle$ encodes an input value x and a computation y. $f(i) = x$, if y is a computation of B of size q that accepts x.)

We show that f is a polynomial-enumeration of B relative to A. It is clear that f belongs to $\mathcal{L}(A)$, and that range $f \subseteq B$. If $x \in B$, then there exists y such that $|y| = q(|x|) \,\&\, R(x, y)$. Let $i = \langle x, y\rangle$. Then, $f(i) = \sigma_1(i) = x$. So, range $f = B$. Moreover, $|i| = |\sigma(x, y)| = s(|x|, |y|) = s(|x|, q(|x|)) = Q(|x|)$. So,

$$x \in B \leftrightarrow \exists i[|i| = Q(|x|) \,\&\, f(i) = x].$$

Thus, we have shown that f has a Q-bounded inverse, and the proof is complete.

The converse of Theorem 2 is also true.

THEOREM 3. *If B has a polynomial-enumeration f relative to A such that f has a q-bounded inverse, then $B \in \mathcal{NP}(A)$ via a computation of size q.*

The proof is straightforward so we just give a sketch. If the hypothesis holds, then

$$\forall y[y \in B \leftrightarrow \exists x[|x| \leqq q(|y|) \,\&\, f(x) = y]].$$

Define $R(x, y)$ to be $[f(x) = y]$. Clearly R belongs to $\mathcal{P}(A)$. Now, pad to obtain a computation of size q.

The following corollary is obvious, but had best not go unnoticed.

COROLLARY 2. *B belongs to $\mathcal{NP}$ if and only if B is polynomial-enumerable. B belongs to $\mathcal{NP}(A)$ if and only if B is polynomial-enumerable in A.*

There is one additional interesting feature about the proofs of Theorems 2 and 3, taken together, that we need to observe. If B has a polynomial-enumeration f relative to A, for some set A, then application of Theorem 3 followed by Theorem 2 yields

another polynomial-enumeration g relative to A having a Q-bounded inverse. But this time we have "equality." That is,

$$\forall x \in B \leftrightarrow \exists i(|i| = Q(|x|) \,\&\, g(i) = x].$$

Henceforth, whenever we say that f is a polynomial enumeration of a set B having a q-bounded inverse, we will assume that to each $x \in B$ there is some y so that $f(y) = x$ and $|y| = q(|x|)$.

1.2. Computations. We have just shown that B belongs to $\mathcal{NP}(A)$ if and only if B is polynomial-enumerable in A. Thus, the nondeterministic recognition of B relative to A is equivalent to the deterministic computation of a function f relative to A. Under what conditions is B recognizable from a polynomial-enumeration of A? Under what conditions is a polynomial-enumeration of B relative to A computable from a given polynomial enumeration of A? In the next section we will respond to these questions, but first we will specify how Turing machines with oracles that compute functions are to be used.

An oracle Turing machine M is either designed to recognize a set A, a set acceptor, or to compute a function f, a transducer. We assume that M contains a read-only input tape, and, in the case of a transducer, a write-only output tape. In either case, it is always assumed in complexity theory that M costs, and that our resources are limited to some resource complexity function T_M (in this paper, the running time.) We conceive of an oracle as auxiliary hardware with only negligible cost that M may access freely during a computation. If this auxiliary hardware is an oracle for some function g, then these considerations lead us to the following stipulations.

M contains a write-only input oracle tape, a separate read-only output tape, and a special oracle call state q. When M enters state q the result of applying the oracle to the string currently on the oracle input tape appears on the oracle output tape. Thus, given an input x to the oracle, the oracle, if called, must return a value $g(x)$. The oracle may not provide its own input, so that any change to the oracle input must be made by M and at the expense of M. Because M is charged for using the oracle, it is possible that M may read only a portion of the oracle's output if the oracle output is too long to read within the resource T_M.

Based on this model, the following definition of a strict polynomial time bounded reducibility between functions will be useful for our purposes.

DEFINITION 3. Define $f \leqq_{\mathrm{T}}^{\mathcal{P}} g$ if there is an oracle Turing machine tranducer M (as above) with oracle g and a polynomial p such that, for each $x \in \{0, 1\}^*$, M computes $f(x)$ in time $p(|x|)$.

This definition is equivalent to the usual definition of $\leqq_{\mathrm{T}}^{\mathcal{P}}$ for sets, via characteristic functions. It can be proved that $\leqq_{\mathrm{T}}^{\mathcal{P}}$-reducibility between functions is transitive, but not in general reflexive. However the restriction of $\leqq_{\mathrm{T}}^{\mathcal{P}}$ to the class of p-bounded functions is clearly reflexive. In this paper our application will always be to polynomial-enumerations (members of $\mathcal{ENUM}$). Further, since it is the class of "feasibly computable" functions whose "polynomial time complexity" we wish to compare, it is entirely reasonable to restrict our attention only to those recursive functions which can, at least, be "written down" within polynomial time. For these reasons, the functions we wish to compute and the oracles we use will be p-bounded.

In the final section we will obtain some results about relative nondeterministic computability between functions. The following definition again takes the conservative view.

DEFINITION 4. Define $f \leqq_T^{\mathcal{NP}} g$ if there is a nondeterministic oracle Turing machine transducer M with oracle g and a polynomial p such that

(1) for each input string x to M there is a computation of M that computes $f(x)$ in time $p(|x|)$, and

(2) if M on input x halts with a word y on its output tape, then $y = f(x)$.

The ideas developed in this subsection 1.2 were developed jointly with T. P. Baker and will be expanded on in a forthcoming paper.

THEOREM 4. *$B \leqq_T^{\mathcal{NP}} A$ if and only if there is a polynomial-enumeration f of B relative to A such that $f \leqq_T^{\mathcal{P}} g$, where g is the trivial polynomial-enumeration of A.*

Proof. By Corollary 2, it is only necessary to show that $f \leqq_T^{\mathcal{P}} A$ is equivalent to $f \leqq_T^{\mathcal{P}} g$. This is clearly so; a set oracle for A can be replaced by a function oracle for the trivial polynomial-enumeration g, and vice versa.

2. Polynomial time enumeration reducibility.

2.1. Nonconstructive approach. The fundamental motivation of enumeration reducibility is that A is to be enumeration reducible to B just in the case that an enumeration of A is computable from any enumeration of B. Thus, we want A to be polynomial time enumeration reducible to B just in the case that every polynomial-enumeration of B (relative to any set X) yields some polynomial-enumeration of A (relative to X). We set down the following definition.

DEFINITION 5. *A is polynomial time enumeration reducible to B* ($A \leqq_{pe} B$) if and only if, for every set X, if B has a polynomial-enumeration relative to X, then A has a polynomial-enumeration relative to X.

By use of Corollary 2, this may be equivalently stated:

$$A \leqq_{pe} B \leftrightarrow \forall X[B \leqq_T^{\mathcal{NP}} X \rightarrow A \leqq_T^{\mathcal{NP}} X].$$

A similar characteristization is proved for (ordinary) enumeration reducibility in [13]. Observe that no mechanism is presented that, given a machine to recognize B relative to X, tells us how to recognize A from X. We have adopted this nonstandard nonconstructive approach because it expresses the intrinsic motivation most transparently, and because, surprisingly, several of the interesting properties of $\leqq_{pe}$ can be proved from this definition.

THEOREM 5. *$\leqq_{pe}$ is reflexive and transitive. $A \leqq_{pe} B$ implies $A \leqq_T^{\mathcal{NP}} B$. The class $\mathcal{NP}$ is the $\mathbf{0}$-degree for the $\leqq_{pe}$-degree structure.*

The proof of each of these statements is trivial. For example, if $A \leqq_{pe} B$, then $A \leqq_T^{\mathcal{NP}} B$ follows, because $B \leqq_T^{\mathcal{NP}} B$. The third statement follows from the fact that a set A belongs to $\mathcal{NP}$ if and only if for every set X, A belongs to $\mathcal{NP}(X)$.

THEOREM 6. *$\leqq_{pe}$ is a maximal transitive subrelation of $\leqq_T^{\mathcal{NP}}$. That is, if $\mathcal{R}$ is any binary relation such that $\leqq_{pe} \subsetneqq \mathcal{R} \subseteq \leqq_T^{\mathcal{NP}}$, then $\mathcal{R}$ is not transitive.*

Proof. Let $\mathcal{R}$ be a binary relation such that $\leqq_{pe} \subsetneqq \mathcal{R} \subseteq \leqq_T^{\mathcal{NP}}$. Then there exists sets A and B such that $A \not\leqq_{pe} B$, but $A\mathcal{R}B$. So, $\exists X[B \leqq_T^{\mathcal{NP}} X \,\&\, A \not\leqq_T^{\mathcal{NP}} X]$. Because $B \leqq_T^{\mathcal{NP}} X$, there is a relation $R \in \mathcal{P}(X)$ and a polynomial q so that

$$\forall x[x \in B \leftrightarrow \exists y[|y| = q(|x|) \,\&\, R(x, y)]].$$

it is easy to see that, for any set C, if $R \leqq_T^{\mathcal{NP}} C$, then $B \leqq_T^{\mathcal{NP}} C$. (Given an input x, just guess a string y such that $|y| = q(|x|)$, and then apply the acceptor for R with oracle C on the input $\langle x, y\rangle$.) Thus, $B \leqq_{pe} R$. So, $B\mathcal{R}R$. If $\mathcal{R}$ is transitive, then $A\mathcal{R}R$ follows from our assumption that $A\mathcal{R}B$ and from $B\mathcal{R}R$. Hence, if $\mathcal{R}$ is transitive, then $A \leqq_T^{\mathcal{NP}} R$. But, $A \leqq_T^{\mathcal{NP}} R$ and $R \in \mathcal{P}(X)$ implies $A \leqq_T^{\mathcal{NP}} X$. X was chosen so that $A \not\leqq_T^{\mathcal{NP}} X$. Thus we have a contradiction; $\mathcal{R}$ is therefore not transitive.

Consider two sets A and B for which $A \leqq_{pe} B$. If $B \leqq_T^{\mathcal{NP}} X$, for some set X, let us consider how a machine that witnesses $A \leqq_T^{\mathcal{NP}} X$ might be obtained. From $A \leqq_{pe} B$ we have $A \leqq_T^{\mathcal{NP}} B$, but, $A \leqq_T^{\mathcal{NP}} B$ and $B \leqq_T^{\mathcal{NP}} X$ do not, in general, imply $A \leqq_T^{\mathcal{NP}} X$. We may expect that $A \leqq_{pe} B$ insures the existence of a "nice" machine for which $A \leqq_T^{\mathcal{NP}} B$ so that, using the "nice" machine and a machine that witnesses $B \leqq_T^{\mathcal{NP}} X$, we can derive a machine that witnesses $A \leqq_T^{\mathcal{NP}} X$. Since there may be distinct sets X_1 and X_2 such that $B \leqq_T^{\mathcal{NP}} X_1$ via a computation of size q_1 and $B \leqq_T^{\mathcal{NP}} X_2$ via a computation of size q_2, where q_1 differs from q_2, let us anticipate that the machine from which we can derive $A \leqq_T^{\mathcal{NP}} X_1$ will be different than the machine from which we can derive $A \leqq_T^{\mathcal{NP}} X_2$. We are led to the following definition.

DEFINITION 6. Define $A \leqq_{pe}^q B$ if and only if, for every set X, if $B \leqq_T^{\mathcal{NP}} X$ by a computation of size q, then $A \leqq_T^{\mathcal{NP}} X$.

Observe that, for $A \leqq_{pe}^q B$ and $B \leqq_T^{\mathcal{NP}} X$ via a computation of size q, no conclusion can be inferred about the size of a resultant computation of A from X—$A \leqq_{pe}^q B$ just guarantees that there is one.

LEMMA 1. $\leqq_{pe} = \bigcap_q \leqq_{pe}^q$. *Thus, $A \leqq_{pe} B$ if and only if, for each polynomial q, $A \leqq_{pe}^q B$.*

Given a polymomial q, let Q be the polynomial in Theorem 2. from Theorems 2 and 3, we have the following:

$A \leqq_{pe}^q B$ implies that, for every set X, if B has a polynomial enumeration relative to X with a q-bounded inverse, then A has a polynomial enumeration relative to X.

If

$$\forall X[B \text{ has a polynomial-enumeration relative to } X \text{ with a } Q\text{-bounded inverse} \rightarrow A \text{ has a polynomial enumeration relative to } X],$$

then $A \leqq_{pe}^q B$.

2.2. Constructive approach. We next adopt the point of view that A is polynomial time enumeration reducible to B just in the case that there are processes which operate in polynomial time and which, whenever they are given polynomial enumerations of B, compute polynomial-enumerations of A. From this stance, we rewrite Definition 5 as follows (and we will prove the alternative definitions equivalent in the next section.)

DEFINITION 5′. For each polynomial q, define $A \leqq_{pe'}^q B$ if and only if there exists an oracle Turing machine transducer M with function oracle g that operates in polynomial time such that, for any set X, if g is any polynomial-enumeration of B relative to X with a q-bounded inverse, then M computes some polynomial-enumeration f of A relative to X.

Define $A \leqq_{pe'} B$ if and only if, for each polynomial q, $A \leqq_{pe'}^q B$.

Definition 5′ is written so as to parallel Definitions 5 and 6 as closely as possible. However, recalling (Corollary 1) that the collection of all polynomial-enumerations can be classified without reference to oracles, we see that Definition 5′ is actually somewhat more complicated than necessary.

THEOREM 7. *$A \leqq_{pe'}^q B$ if and only if there is an oracle Turing machine transducer M with function oracle g and polynomial running time that has the following property: if* range $g = B$, *g belongs to $\mathcal{ENUM}$, and g has a q-bounded inverse, then M computes a function f such that* range $f = A$ *and f belongs to $\mathcal{ENUM}$.*

The proof from left to right follows from Corollary 1. The proof in the other direction follows from the transitivity of $\leqq_T^{\mathcal{P}}$; i.e., if, for some set X, $f \leqq_T^{\mathcal{P}} g$ and $g \leqq_T^{\mathcal{P}} X$, then $f \leqq_T^{\mathcal{P}} X$.

$\leqq_{\text{pe}'}$ can also be characterized in terms of nondeterministic set acceptors rather than machines that compute functions. Lemma 2 to follow presents this characterization.

LEMMA 2. *Let q be a polynomial. For any two sets A and B the following are equivalent*:

(1) *there is a nondeterministic oracle Turing machine that witnesses $A \leqq_{\text{T}}^{\mathcal{NP}} B$ and a constant k such that, for each input string x in A, there is an accepting computation in which every query z about membership in B that receives a "no" answer satisfies $q(|z|) \leqq k \log |x|$.*

(2) *there is a set W in $\mathcal{NP}$ (over the alphabet $\{0, 1, c\}$), a polynomial p, and a constant k such that*

$$\begin{aligned}\forall x[x \in A \leftrightarrow \exists \alpha[&|\alpha| \leqq p(|x|) \ \& \ xc\alpha \in W \\ &\& \ \alpha = cy_1 \cdots cy_n ccz_1 \cdots cz_m \\ &\& \ y_1, \cdots, y_n, z_1, \cdots, z_m \in \{0, 1\}^* \\ &\& \ \forall j \leqq m, q(|z_j|) \leqq k \log |x| \\ &\& \ \{y_1, \cdots, y_n\} \subseteq B \ \& \ \{z_1, \cdots, z_m\} \subseteq \bar{B}]].\end{aligned}$$

The proof technique can be found in [8]. We will show that $A \leqq_{\text{pe}'} B$ if and only if, for each polynomial q, there is an oracle Turing machine M that satisfies the property (1) of Lemma 2. M is the "nice" machine alluded to in the previous subsection. The following theorem gives the proof in one direction; the other direction will appear as a consequence of results in the next section.

THEOREM 8. *Let q be a polynomial. For any two sets A and B, if there exists an oracle Turing machine M that satisfies property (1) of Lemma 2, then $a \leqq_{\text{pe}'}^{q} B$.*

Proof. Given A and B, suppose the hypothesis is satisfied. Using Lemma 2, let W in $\mathcal{NP}$, polynomial p, and constant k, satisfy property (2). Consider then the following nondeterministic oracle Turing machine M designed to satisfy property (1).

On input x, M makes the sequence of steps:

1. M guesses a string α such that $|\alpha| \leqq p(|x|)$;
2. checks whether $xc\alpha$ belongs to W;
3. checks whether $\alpha = cy_1 \cdots cy_n ccz_1 \cdots cz_m$;
4. for each z_j, M checks whether $q(|z_j|) \leqq k \log |x|$;
5. in sequence, M places each y_i on the oracle tape and enters the query state; if the answer is "yes", then M continues;
6. in sequence, M places each z_j on the oracle tape and enters the query state; if the answer is "no", then M continues.

M accepts the input string x only if each test is successful. Clearly, M satisfies property (1). We now define a nondeterministic Turing acceptor M_1 with function oracle g. On input x, steps 1–4 of M_1 are the same as M. Steps 5 and 6 are the following:

5. In sequence, for each y_i, M guesses a string ξ such that $|\xi| = q(|y_i|)$ and checks whether $g(\xi) = y_i$; i.e., M write ξ on the oracle input tape, goes into the oracle call state, and then checks whether y_i is on the oracle output tape.
6. In sequence, for each z_j, M generates each string ξ such that $|\xi| = q(|z_j|)$, writes ξ on the oracle input tape, goes into the oracle call state, and checks whether z is *not* returned.

M_1 accepts x only if each test is successful. We claim than whenever g is a polynomial-enumeration of B ($g \in \mathcal{ENUM}$ and range $g = B$) having a q-bounded inverse, then M_1 recognizes A in polynomial time. M_1 recognizes A because M_1 simulates M. We need to show that M_1 operates in polynomial time. Steps 1–4 can clearly be executed in polynomial time. Since g is p_1-bounded, for some polynomial p_1, step 5 can be executed in polynomial time. To see that step 6 can be executed in polynomial time, observe first that

$$z_j \in B \leftrightarrow \exists \xi [g(\xi) = z_j \ \& \ |\xi| = q(|z_j|)].$$

There are $2^{q(|z_j|)}$ distinct strings ξ such that $|\xi| = q(|z_j|)$. Since $q(|z_j|) \leqq k \log |x|$, it follows that at most $|x|^k$ values $g(\xi)$ need to be obtained. That is, M_1 can check whether z_j does not belong to B by making no more than $|x|^k$ oracle calls, and checking for each such call that z_j is not the value returned. Thus step 6 can also be executed in polynomial time, so our claim is proved.

Next, to complete the proof of Theorem 8, we construct a deterministic Turing machine transducer M_2 with oracle g so that whenever g is a polynomial-enumeration of B having a q-bounded inverse, then M_2 computes some polynomial-enumeration of A. Let a belong to A. On input $\xi = \langle x, y \rangle$, M_2 is to check whether y is an accepting computation of M_1 on input x. If so, then M_2 outputs x; otherwise M_2 outputs a. It is evident that M_2 has the required property (see the proof of Theorem 2). Thus, the proof is complete.

COROLLARY 3. *Given sets A and B, if for every polynomial q there exists a nondeterministic oracle Turing machine that witnesses $A \leqq_T^{\mathcal{NP}} B$ and a constant k such that, for each input string x in A there is an accepting computation in which every query z about membership in B that receives a "no" answer satisfies $q(|z|) \leqq k \log |x|$, then $A \leqq_{pe'} B$.*

3. Main results.

THEOREM 9. *$A \leqq_{pe'} B$ implies $A \leqq_{pe} B$.*

The proof follows directly from the definitions. The following theorem is central to the converse.

THEOREM 10. *For each polynomial q, if $A \leqq_{pe}^{q} B$, then there is a set W in $\mathcal{NP}$, a polynomial p, and a constant k for which property (2) in lemma 2 is satisfied.*

Before turning to the proof of Theorem 10, let us note that, as a consequence, our various characterizations of polynomial time enumeration reducibility are all equivalent. In particular $\leqq_{pe}$ is identical to $\leqq_{pe'}$. Therefore, beyond this section the notation $\leqq_{pe'}$ will no longer be useful. The important conclusions are summarized in the following corollary.

COROLLARY 4. *The following are all equivalent.*

(1) $\forall X[B$ *has a polynomial-enumeration relative to* $X \to A$ *has a polynomial-enumeration relative to* $X]$.

(2) $\forall X[B \leqq_T^{\mathcal{NP}} X \to A \leqq_T^{\mathcal{NP}} X]$.

(3) *For every polynomial q there is an oracle Turing machine transducer M with function oracle g and polynomial running time such that M computes a polynomial-enumeration of A (relative to X) whenever g is a polynomial-enumeration of B (relative to X) having a q-bounded inverse.*

(4) *For every polynomial q there is a nondeterministic oracle Turing machine acceptor M that witnesses $A \leqq_T^{\mathcal{NP}} B$ and a constant k such that, for each input string x in A there is an accepting computation in which every query z about membership in B that receives a "no" answer satisfies $q(|z|) \leqq k \log |x|$.*

(5) *For every polynomial q, there is a set W in $\mathcal{NP}$, a polynomial p, and a constant k such that*

$$\begin{aligned}\forall x[x \in A \leftrightarrow \exists \alpha[|\alpha| \leqq p(|x|) \ \&\ xc\alpha \in W \\ \&\ \alpha = cy_1 \cdots cy_n ccz_1 \cdots cz_m \\ \&\ y_1, \cdots, y_n, z_1, \cdots, z_m \in \{0, 1\}^* \\ \&\ \forall j \leqq m \quad q(|z_j| \leqq k \log |x| \\ \&\ \{y_1, \cdots, y_n\} \subseteq B \ \&\ \{z_1, \cdots, z_m\} \subseteq \bar{B}]].\end{aligned}$$

Proof. Part (1)→(2) is clear; part (2)→(5) is Theorem 10; part (5)→(4) by lemma 2; part (4)→(3) is Corollary 3; and, part (3)→(1) is Theorem 9. Thus, all the equivalences are proved.

Note that (3)→(4) is the converse of Corollary 3, thereby completing the argument begun in the previous section that $\leqq_{\text{pe}'}$ can be characterized either by nondeterministic set acceptors or by deterministic oracle machines that compute functions.

The proof of Theorem 10 will take up the remainder of this section. This is the difficult direction, for we want to show that if, for every set X, $B \leqq_{\text{T}}^{\mathcal{NP}} X$ implies $A \leqq_{\text{T}}^{\mathcal{NP}} X$, then there is a machine that constructively effects the implication. It should be clear that no straightforward design of such a machine can be accomplished. In fact, given a fixed polynomial q, we show, unless there is an appropriate set W in $\mathcal{NP}$, polynomial p, and constant k, that there exists a set C such that $B \leqq_{\text{T}}^{\mathcal{NP}} C$ and $A \not\leqq_{\text{T}}^{\mathcal{NP}} C$. C is obtained by a construction that combines a diagonalization so that $A \not\leqq_{\text{T}}^{\mathcal{NP}} C$ with an encoding so that $B \leqq_{\text{T}}^{\mathcal{NP}} C$.

In order to effect $A \not\leqq_{\text{T}}^{\mathcal{NP}} C$, we make use of the result [8] that $A \leqq_{\text{T}}^{\mathcal{NP}} C \leftrightarrow A \leqq_{\text{tt}}^{\mathcal{NP}} C$. For the convenience of the reader, we repeat here the definition of $\leqq_{\text{tt}}^{\mathcal{NP}}$.

Let Δ be a fixed alphabet for encoding Boolean functions, and let $c \notin \Delta \cup \{0, 1\}$.

A *tt-condition* is a member of $\Delta^* c(c\{0, 1\}^*)^*$.

A *tt-condition evaluator* e is a recursive mapping of $\Delta^* c\{0, 1\}^*$ into $\{0, 1\}$.

A tt-condition $\alpha ccy_1 cy_2 c \cdots cy_k$ is *e-satisfied* by $B \subseteq \{0, 1\}$ if and only if $e(\alpha c C_B(y_1) \cdots C_B(y_k)) = 1$.

$A \leqq_{\text{tt}}^{\mathcal{NP}} B$ if and only if there is a nondeterministic Turing machine transducer g that runs in polynomial time and a polynomial time computable evaluator e such that $x \in A$ just in case on input x, g computes a tt-condition y which is e-satisfied by B.

The nondeterministic transducer g, we will call a nondeterministic *tt-condition generator.*

Next, we assume an effective enumeration $(g, e)_i$ of all pairs of nondeterministic tt-condition generators and tt-condition evaluators with associated polynomial run times, so that g and e are both time bounded by the polynomial p_i.

Proof of Theorem 10. Let q be a fixed polynomial, and let A and B be given so that Lemma 2, property (2) is not satisfied. C will be constructed in stages. C may be thought of as a subset of $\{0, 1\}^* \times \{0, 1\}^*$, via the pairing function $\langle \cdot, \cdot \rangle$. We will effect $B \leqq_{\text{T}}^{\mathcal{NP}} C$ by constructing C so that

$$x \in B \leftrightarrow \exists y[|y| = q(|x|) \ \&\ \langle x, y \rangle \in C].$$

At stage 0, C is empty. At stage i, let us suppose that C has been determined on some finite domain U_i; let $C_i = C \cap U_i$.

For a set S, define $\text{FIRST}(S)=\{x|\langle x, y\rangle \in S\}$. We will assume as an induction hypothesis that

$$\forall x \in \text{FIRST}(U_i)[x \in B \leftrightarrow \exists y[|y| = q(|x|) \,\&\, \langle x, y\rangle \in C_i]].$$

We next extend the domain of definition of C to some $U_{i+1} = U_i \cup V$; C_{i+1} will then be some set $C_{i+1} = C_i \cup D$, where $D \subseteq V$. To insure that values are not redefined, we only consider extensions that satisfy the predicate $\text{EXTEND}(U_i, C_i, V, D)$, defined by

$$\text{EXTEND}(U_i, C_i, V, D) \equiv [D \subseteq V \,\&\, D \cap (U_i - C_i) = \varnothing \,\&\, C_i \cap (V - D) = \varnothing].$$

Also, to insure that the induction hypothesis can be met at the next stage, we only consider extensions that satisfy the predicate $\text{COMPATIBLE}(V, D)$ defined by

$$\text{COMPATIBLE}(V, D) \equiv [|y| = q(|x|) \,\&\, \langle x, y\rangle \in D \rightarrow x \in B, \text{ and}$$
$$\forall y[|y| = q(|x|) \rightarrow \langle x, y\rangle \in V \,\&\, \langle x, y\rangle \notin D] \rightarrow x \notin B].$$

Finally, let $(g, e)_i$ be given. g and e are time bounded by the polynomial $p = p_i$.

There are three logical cases which we consider.

Case 1. $\exists x[x \notin A \,\&\, \exists V, D[\text{EXTEND}(U_i, C_i, V, D) \,\&\, \text{COMPATIBLE}(V, D)$ & on input x, g computes a tt-condition $\alpha ccy_1 \cdots cy_k$ & $V = \{y_1, \cdots, y_k\}$ & $\alpha ccy_1 \cdots cy_k$ is e-satisfied by $D]]$.

Case 2. $\exists x[x \in A \,\&\, \forall V, D[\text{EXTEND}(U_i, C_i, V, D) \,\&\, \text{COMPATIBLE}(V, D)$ & on input x, g computes a tt-condition $\alpha ccy_1 \cdots cy_k$ & $V = \{y_1, \cdots, y_k\} \rightarrow$ $\alpha ccy_1 \cdots cy_k$ is not e-satisfied by $D]]$.

Case 3. $\forall x[x \in A \leftrightarrow \exists V, D[\text{EXTEND}(U_i, C_i, V, D) \,\&\, \text{COMPATIBLE}(V, D)$ & on input x, g computes a tt-condition $\alpha ccy_1 \cdots cy_k$ & $V = \{y_1, \cdots, y_k\}$ & $\alpha ccy_1 \cdots cy_k$ is e-satisfied by $D]]$.

We will prove, in fact, that, for each stage i, Case 3 cannot occur; that is, Case 3 leads to a contradiction. The details of this argument are rather complex, so the proof will be postponed until the end of the construction as a separate lemma. It follows that either Case 1 holds or Case 2 holds. We proceed now on this assumption.

Case 1. Choose $a \notin A$, V, and D, $D \subseteq V$, accordingly. Extend C_i and U_i to C_{i+1} and U_{i+1} by the following procedure.

1. $C_{i+1} \leftarrow C_i \cup D$; $U_{i+1} \leftarrow U_i \cup V$;
2. **for** $x \in B \,\&\, x \in \text{FIRST}(U_{i+1}) \,\&\, x \notin \text{FIRST}(U_i)$
   do
   if $\exists z \quad |z| = q(|x|) \,\&\, \langle x, z\rangle \notin V$
   then begin
   $C_{i+1} \leftarrow C_{i+1} \cup \{\langle x, z\rangle\}$;
   $U_{i+1} \leftarrow U_{i+1} \cup \{\langle x, z\rangle\}$
   end;
3. Let α be the smallest string not in $\text{FIRST}(U_{i+1})$;
   for b such that $|b| = q(|\alpha|)$
   do begin
   $U_{i+1} \leftarrow U_{i+1} \cup \{\langle \alpha, b\rangle\}$;
   if $\alpha \in B$ **then** $C_{i+1} \leftarrow C_{i+1} \cup \{\langle \alpha, b\rangle\}$
   end

We will show now that the induction hypothesis is satisfied at $i+1$.

We show first that

(I) $$|y| = q(|x|) \,\&\, \langle x, y\rangle \in C_{i+1} \rightarrow x \in B.$$

If $\langle x, y\rangle \in C_i$, then $x \in B$, by the induction hypothesis at i. If $\langle x, y\rangle \in D$, then $y \in B$, by COMPATIBLE(V, D). Thus (I) is true for C_{i+1} after execution of step 1 of the procedure. Both steps 2 and 3 insert a pair $\langle x, y\rangle$ into C_{i+1} only if $x \in B$. Thus (I) is true.

We must next show that

(II) $\quad x \in \text{FIRST}(U_{i+1})$ and $x \in B$ implies $\exists y[|y| = q(|x|) \,\&\, \langle x, y\rangle \in C_{i+1}]$.

If $x \in \text{FIRST}(U_i)$, the conclusion follows by the induction hypothesis at i.

Suppose $x \notin \text{FIRST}(U_i)$. By COMPATIBLE$(V, D)$,

$$\neg\forall y[|y| = q(|x|) \to \langle x, y\rangle \in V \,\&\, \langle x, y\rangle \notin D].$$

Thus, there exists y so that either $(|y| = q(|x|) \,\&\, \langle x, y\rangle \notin V)$ or $(|y| = q(|x|) \,\&\, \langle x, y\rangle \in D)$. In the latter case the conclusion follows, since $D \subseteq C_{i+1}$. In the former case we have, after execution of step 1,

$$x \in B \,\&\, x \in \text{FIRST}(U_{i+1}) \,\&\, x \notin \text{FIRST}(U_i).$$

Therefore, step 2 inserts $\langle x, y\rangle$ into C_{i+1}.

Finally, for the string α considered in step 3, if $\alpha \in B$, then step 3 inserts each pair $\langle \alpha, b\rangle$ into C_{i+1}, where $|b| = q(|\alpha|)$. Thus (II) is true, and the proof is complete.

Case 2. Extend C_i and U_i by the following procedure.

1. $C_{i+1} \leftarrow C_i$; $U_{i+1} \leftarrow U_i$;
2. This step is identical to step 3 of the previous procedure.

It is readily shown that, in this case too, the induction hypothesis is satisfied at $i+1$.

We are now ready to show that

$$x \in B \leftrightarrow \exists y[|y| = q(|x|) \,\&\, \langle x, y\rangle \in C].$$

Since each extension is nontrivial, $\text{FIRST}(C) = \{0, 1\}^*$. If $x \in B$, then, for some i, $x \in \text{FIRST}(U_i)$. Thus, by the induction hypothesis $\exists y[|y| = q(|x|) \,\&\, \langle x, y\rangle \in C]$. Conversely, if $|y| = q(|x|) \,\&\, \langle x, y\rangle \in C$, then, for some i, $\langle x, y\rangle \in C_i$. Therefore, $x \in B$ again follows from the induction hypothesis.

We show next that $A \not\leq_{\text{tt}}^{\mathcal{NP}} C$. If $A \leq_{\text{tt}}^{\mathcal{NP}} C$, then for some i, $(g, e)_i$ is a witness to the reduction.

We are assuming that either Case 1 or Case 2 holds at stage i. Suppose Case 1 holds. Then,

$$\begin{aligned}\exists x \quad & x \notin A \text{ and } \exists V, D[\text{EXTEND}(U_i, C_i, V, D)\\ & \,\&\, \text{COMPATIBLE}(V, D) \,\&\, \text{on input } x,\ g \text{ computes}\\ & \text{a tt-condition } \alpha c c y_1 \cdots c y_k \,\&\, V = \{y_1, \cdots, y_k\}\\ & \,\&\, \alpha c c y_1 \cdots c y_k \text{ is } e\text{-satisfied by } D].\end{aligned}$$

The construction at Case 1 makes $D \subseteq C_{i+1} \subseteq C$ and $V - D \subseteq U_{i+1} - C_{i+1} \subseteq \bar{C}$. Thus, $\alpha c c y_1 \cdots c y_k$ is e-satisfied by C. So, if Case 1 holds at stage i, then $A \not\leq_{\text{tt}}^{\mathcal{NP}} C$ via $(g, e)_i$.

On the other hand, suppose Case 2 holds. Choose $x \in A$ according to Case 2. It is necessary to show that if g computes a tt-condition $\alpha c c y_1 \cdots c y_k$ on input x, then $\alpha c c y_1 \cdots c y_k$ is not e-satisfied by C. Let $V = \{y_1, \cdots, y_k\}$, and let $D = V \cap C$. Then, $\alpha c c y_1 \cdots c y_k$ is e-satisfied by C if and only if $\alpha c c y_1 \cdots c y_k$ is e-satisfied by D. EXTEND(U_i, C_i, V, D), because $D \subseteq V$, $U_i - C_i \subseteq \bar{C}$, whereas $D \subseteq C$, and $C_i \subseteq C$, whereas $V - D \subseteq C$. To show COMPATIBLE(V, D), $|y| = q(|x|) \,\&\, \langle x, y\rangle \in D \to x \in B$,

since $D \subseteq C$. If $\forall y[|y| = q(|x|) \to \langle x, y\rangle \in V \ \& \ \langle x, y\rangle \notin D]$, then, since $V - D \subseteq \bar{C}$, we have $\forall y[|y| = q(|x|) \to \langle x, y\rangle \notin C]$. Therefore, $x \notin B$. Therefore, by the Case 2 hypothesis, $\alpha c c y_1 \cdots c y_k$ is not e-satisfied by D. Hence, $A \not\leq_{\mathrm{tt}}^{\mathcal{NP}} C$.

We make one final remark concerning the construction of C. Namely, the construction of C is effective relative to the sets A and B. Thus, since A and B are recursive, so is C. Given x and a pair $(g, e)_i$, to determine whether Case 1 or Case 2 holds is essentially no more difficult than checking whether $x \in A$, testing all computations of g on input x, and checking whether COMPATIBLE(V, D), the latter condition being recursive in B. Moreover, since at each i either Case 1 holds or Case 2 holds, a loop through all strings will halt at either Case 1 or Case 2.

With the exception of showing, for each stage i, that Case 3 cannot hold, the proof of Theorem 10 is now complete.

LEMMA 3. *For each stage i, the Case 3 hypothesis implies the existence of a set W in $\mathcal{NP}$, a polynomial r, and a constant k such that*

$$\begin{aligned}\forall x[x \in A \leftrightarrow \exists \alpha[&|\alpha| \leqq r(|x|) \ \& \ x c \alpha \in W \\ &\& \ \alpha = c y_1 \cdots c y_n c c z_1 \cdots c z_m \ \& \ y_1, \cdots, y_n, z_1, \cdots, z_m \in \{0,1\}^* \\ &\& \ \forall j \leqq m \quad q(|z_j|) \leqq k \log |x| \\ &\& \ \{y_1, \cdots, y_n\} \subseteq B \ \& \ \{z_1, \cdots, z_m\} \subseteq \bar{B}]].\end{aligned}$$

Therefore, at each stage i, Case 3 does not hold.

Proof. Let us assume at stage i that Case 3 holds. Thus,

$$\begin{aligned}\forall x[x \in A \leftrightarrow \exists V, D[&\mathrm{EXTEND}(U_i, C_i, V, D) \\ &\& \ \mathrm{COMPATIBLE}(V, D) \ \& \text{ on input } x, g \text{ computes a} \\ &\text{tt-condition } \alpha c c y_1 \cdots c y_k \ \& \ V = \{y_1, \cdots, y_k\} \\ &\& \ \alpha c c y_1 \cdots c y_k \text{ is } e\text{-satisfied by } D]].\end{aligned}$$

Encode a pair of finite sets (V, D), $D \subseteq V$, by the string

$$c x_1 c 1 c x_2 c 1 \cdots c x_k c 1 c z_1 c 0 c z_2 c o \cdots c z_l c 0,$$

where $D = \{x_1, \cdots, x_k\}$, and $V = \{x_1, \cdots, x_k, z_1, \cdots, z_l\}$.

If g computes a tt-condition $\alpha c c y_1 \cdots c y_k$ on input x, then $|c y_1 \cdots c y_k| \leqq p(|x|)$. Thus, if $V = \{y_1, \cdots, y_k\}$, D is any subset of V, and δ is an encoding of (V, D), it follows that $|\delta| \leqq 2 \cdot p(|x|)$.

We design a nondeterministic Turing machine T as follows:

T performs the following sequence of processes. After each step except the last, T either goes to the next step or halts without accepting.

1. On input β to T, T checks whether $\beta = x c c u_1 \cdots c u_n c c v_1 c \cdots c v_m$, where u_i, $v_j \in \{0, 1\}^*$, $i \leqq n$, $j \leqq m$. If so,

2. T guesses a string δ such that $|\delta| \leqq 2 \cdot p(|x|)$. T then checks whether δ is an encoding of finite sets (V, D) and checks whether EXTEND(U_i, C_i, V, D). (Observe that (U_i, C_i) can be stored in the finite-control of T.) If so,

3. T checks whether

$$\{u_1, \cdots, u_n\} = \{u \mid \exists v \quad |v| = q(|u|) \ \& \ \langle u, v\rangle \in D\}$$

and

$$\{v_1, \cdots, v_m\} = \{u \mid \forall v(|v| = q(|u|) \to (\langle u, v\rangle \in V \ \& \ \langle u, v\rangle \notin D))\}.$$

If so,

4. T next behaves like the transducer g on input x. If g generates a tt-condition $\alpha ccy_1 \cdots cy_s$, then

5. T checks whether $\{y_1, \cdots, y_s\} = V$. If so,

6. T writes $\alpha ca_1 \cdots a_s$ on one of its work tapes, where $a_i = 1$, if $y_i \in D$, and $a_i = 0$, if $y_i \in V - D$.

7. T behaves like e on input $\alpha ca_1 \cdots a_s$. T accepts β if and only if $e(\alpha ca_1 \cdots a_s) = 1$.

Let W be the set of strings accepted by T. It follows that W is in $\mathcal{NP}$ over $\{0, 1, c\}^*$. Moreover,

$$\begin{aligned}\beta \in W \leftrightarrow \beta = xccu_1 \cdots cu_n ccv_1 \cdots cv_m \\ \& \ \exists V, D[\mathrm{EXTEND}(U_i, C_i, V, D) \\ \& \ g \text{ computes a tt-condition } \alpha ccy_1 \cdots cy_s \\ \text{on input } x \ \& \ \alpha ccy_1 \cdots cy_s \text{ is } e\text{-satisfied by } D \\ \& \ \{u_1, \cdots, u_n\} = \{u \mid \exists v(|v| = q(|u|) \ \& \ \langle u, v\rangle \in D)\} \\ \& \ \{v_1, \cdots, v_m\} = \{u \mid \forall v(|u| = |v| \rightarrow (\langle u, v\rangle \in V \ \& \ \langle u, v\rangle \notin D))\}].\end{aligned}$$

So after guessing an encoding of a pair (V, D), T checks all the conditions of the Case 3 hypothesis, except COMPATIBLE. The pair (V, D), then, satisfies COMPATIBLE(V, D) if and only if $\{u_1, \cdots, u_n\} \subseteq B$ and $\{v_1, \cdots, v_m\} \subseteq \bar{B}$.

Since $x \in A$ if and only if there is a pair (V, D) that satisfies the Case 3 hypothesis, and since the encoding of such a pair has polynomial length, it follows that there is a polynomial r such that $x \in A$ if and only if there is an appropriate string $\beta = xcu_1 \cdots cu_n ccv_1 \cdots cv_m$ that is accepted by T and such that $|cu_1 \cdots cu_n ccv_1 \cdots cv_m| \leqq r(|x|)$.

Further, suppose

$$\forall v(|v| = q(|u|) \rightarrow \langle u, v\rangle \in V \ \& \ \langle u, v\rangle \notin D).$$

Considering step 2 in the definition of T, $|\delta| \leqq 2 \cdot p(|x|)$. There are $2^{q(|u|)}$ strings v such that $|v| = q(|u|)$. For each of these $\langle u, v\rangle$ occurs in δ. Thus, $2^{q(|u|)} \leqq 2 \cdot p(|x|)$. Thus, $q(|u|) \leqq 1 + \log p(|x|)$. It follows that there is a constant k such that, for each member $v_j \in \{v_1, \cdots, v_m\}$, $q(|v_j|) \leqq k \log |x|$.

To summarize, Case 3 implies the existence of a set W in $\mathcal{NP}$, a polynomial r, and a constant k such that

$$\begin{aligned}x \in A \leftrightarrow \exists \alpha[|\alpha| \leqq r(|x|) \ \& \ xc\alpha \in W \\ \& \ \alpha = cu_1 \cdots cu_n cv_1 \cdots cv_m \\ \& \ \{u_1, \cdots, u_n\} \subseteq B \ \& \ \{v_1, \cdots, v_m\} \subseteq \bar{B} \\ \& \ \forall j \leqq m \quad q(|v_j|) \leqq k \log |x|].\end{aligned}$$

Our proof is complete; Case 3 of the Theorem 10 does not hold.

4. $\leqq_{pe}$ and other reducibilities.

4.1. Comparison with nondeterministic conjunctive and Turing reducibilities. We turn to relating polynomial time enumeration reducibility with the other nondeterministic polynomial time reducibilities studied in [8]. First, the following provides a useful example.

LEMMA 4. *Suppose, for two sets A and B, that $x \in A \leftrightarrow \log\log x \in \bar{B}$. Then, $A \leqq_{\text{pe}} B$.*

Proof. $\log\log x \approx \log|x|$. Thus, $|\log\log x| \approx \log\log|x|$. For every polynomial q there is a constant k such that

$$q(\log\log|x|) \leqq k \log|x|.$$

The conclusion follows from Corollary 4, part (4) or (5).

Referring to Corollary 4 part (5) it is proved in [8] that $\leqq_{\text{c}}^{\mathcal{NP}}$ is the result of eliminating all appearances of the z_j:

$A \leqq_{\text{c}}^{\mathcal{NP}} B$ if and only if there is a set W in $\mathcal{NP}$ (over $\{0, 1, c\}$) and a polynomial p such that x belongs to A just in case there is a string α so that $|\alpha| \leqq p(|x|)$, $xc\alpha \in W$, $\alpha = cy_1 \cdots cy_n$ with $y_1, \cdots, y_n \in \{0, 1\}^*$ and $\{y_1, \cdots, y_n\} \subseteq B$.

Moreover it is not difficult to prove that $\leqq_{\text{T}}^{\mathcal{NP}}$ results if the restriction to the length of the z_j is eliminated:

$A \leqq_{\text{T}}^{\mathcal{NP}} B$ if and only if there is a set W in $\mathcal{NP}$ (over $\{0, 1, c\}$) and a polynomial p such that x belongs to A just in case there is a string α so that $|\alpha| \leqq p(|x|)$, $xc\alpha \in W$, $\alpha = cy_1 \cdots cy_n cz_1 \cdots cz_m$ with $y_1, \cdots, y_n$, $z_1, \cdots, z_m \in \{0, 1\}^*$, $\{y_1, \cdots, y_n\} \subseteq B$, and $\{z_1, \cdots, z_m\} \subseteq \bar{B}$.

In fact, we have the following theorem.

THEOREM 11. $\leqq_{\text{c}}^{\mathcal{NP}} \subsetneq \leqq_{\text{pe}} \subsetneq \leqq_{\text{T}}^{\mathcal{NP}}$.[1]

Proof. The inclusions are clear. $\leqq_{\text{pe}} \neq \leqq_{\text{T}}^{\mathcal{NP}}$ follows, since $\leqq_{\text{T}}^{\mathcal{NP}}$ is not transitive.

We prove that $\leqq_{\text{c}}^{\mathcal{NP}} \neq \leqq_{\text{pe}}$ by a diagonalization. We construct sets A and B so that $x \in A \leftrightarrow \log\log x \in \bar{B}$ and $A \not\leqq_{\text{c}}^{\mathcal{NP}} B$. The result then follows, by use of Lemma 4.

Only numbers (strings) of the form 2^{2^m} are placed into A. We use the characterization of $\leqq_{\text{c}}^{\mathcal{NP}}$ written above.

At stage i of the construction there are numbers n_i and m_i so that membership in A has been determined on the domain $\{0, \cdots, n_i\}$ and membership in B has been determined on the domain $\{0, \cdots, m_i\}$. let $A(i)$ and $B(i)$ be the sets of numbers thus far placed into A and B, respectively.

Let x_i be the first number so that, for some m, $x_i = 2^{2^m}$, $x_i > n_i$, and $m > m_i$. Let W and p be the $\mathcal{NP}$ set and polynomial, respectively, to be diagonalized over at stage i. We treat two cases.

Case 1. There is a string α and an extension $B'(i)$ of $B(i)$ so that $|\alpha| \leqq p(|x|)$, $x_i c\alpha$ belongs to W, $\alpha = cy_1 cy_2 \cdots cy_k$, and $\{y_1, \cdots, y_k\} \subseteq B'(i)$.

In this case, let $m_{i+1} = \max(m, y_1, \cdots, y_k)$. Take $B(i+1) = B(i) \cup \{m_i + 1, \cdots, m_{i+1}\}$. Let $n_{i+1} = x_i$, and take $A(i+1) = A(i)$.

Case 2. For every α so that $|\alpha| \leqq p(|x|)$, $\alpha = cy_1 cy_2 \cdots cy_k$, and $xc\alpha$ in W, and for all extensions $B'(i)$ of $B(i)$, $\{y_1, \cdots, y_k\} \not\subseteq B'(i)$.

Then, let $m_{i+1} = m$. Take $B(i+1) = B(i) \cup \{m_i + 1, \cdots, m-1\}$. Let $n_{i+1} = x_i$, and take $A(i+1) = A(i) \cup \{x_i\}$.

It is a straightforward task to see that the construction is correct.

let $\text{TIME}(2^{\mathcal{P}})$ be the class of sets recognized by deterministic Turing machines which run in time $2^{p(n)}$, for $p(n)$ a polynomial. In contrast to Theorem 11, we have the following result.

THEOREM 12. *$A \leqq_{\text{pe}} B$ and $B \in \text{TIME}(2^{\mathcal{P}})$ imply $A \leqq_{\text{c}}^{\mathcal{NP}} B$.*

Proof. Let M be a deterministic Turing machine that recognizes B in time $2^{q(n)}$, q a polynomial. Given the polynomial q, let W, p, and k be the $\mathcal{NP}$ set, polynomial, and constant, respectively, whose existence is guaranteed by Corollary 4 part (5).

[1] It is suggested in [7] that $\leqq_{\text{c}}^{\mathcal{NP}}$ is a polynomial time enumeration reducibility, and the author claims in [8] that $\leqq_{\text{c}}^{\mathcal{NP}}$ is a maximal transitive subrelation of $\leqq_{\text{T}}^{\mathcal{NP}}$. This theorem corrects those remarks.

Design a nondeterministic acceptor T as follows: Input to T is a string $xc\alpha = xccy_1 \cdots cy_m$ where $y_1, \cdots, y_n$ are in $\{0, 1\}^*$. T guesses strings $z_1, \cdots, z_m \in \{0, 1\}^*$ such that $|\alpha ccz_1 c \cdots cz_m| \leqq p(|x|)$ and, for each $i \leqq m$, $|z_i| \leqq q(|x|)$. T then behaves like an $\mathcal{NP}$-acceptor for W on input $xc\alpha ccz_1 c \cdots cz_m$. If this string belongs to W, then, for each z_i, T next behaves like M on input z_i and determines whether $z_i \in \bar{B}$. T accepts its input if each test is successful.

M determines whether $z_i \in \bar{B}$ in time

$$2^{q(|z_i|)} \leqq 2^{k \log |x|} \leqq |x|^k.$$

Therefore, T operates in polynomial time. Thus, the set W' accepted by T belongs to $\mathcal{NP}$. We conclude that $A \leqq_c^{\mathcal{NP}} B$ via the $\mathcal{NP}$ set W' and the polynomial p.

Thus, $\leqq_c^{\mathcal{NP}}$ and $\leqq_{pe}$ are identical on all of the low level complexity classes. In particular, $\leqq_c^{\mathcal{NP}}$ and $\leqq_{pe}$ are identical on PSPACE (the class of sets recognized by deterministic Turing machines which run in polynomial space) and the polynomial hierarchy [10]. This latter fact suggests that the techniques of Leggett [6], which use $\leqq_c^{\mathcal{NP}}$ to determine the classification of certain interesting sets in the polynomial hierarchy, are the best possible.

We hasten to add, however, that it is not known whether $\leqq_c^{\mathcal{NP}}$ $(= \leqq_{pe})$ is a maximal transitive subrelation of $\leqq_T^{\mathcal{NP}}$ on TIME$(2^{\mathcal{P}})$. Indeed, it is not known whether $\mathcal{NP}$ is properly included in TIME$(2^{\mathcal{P}})$.

Let $\mathcal{C}$ denote any of the classes TIME$(2^{\mathcal{P}})$, PSPACE, or the collection of sets in the polynomial hierarchy. We conjecture that $\mathcal{NP} \neq \mathcal{C}$ implies $\leqq_c^{\mathcal{NP}}$ is a maximal transitive subrelation of $\leqq_T^{\mathcal{NP}}$ on $\mathcal{C}$. (For the case of the polynomial hierarchy, we note that it is easily proved that $\leqq_T^{\mathcal{NP}}$ is transitive on the polynomial hierarchy if and only if the hierarchy collapses to $\mathcal{NP}$.)

As a final remark, observe that the set B constructed in the proof of Theorem 11 does not belong to TIME$(2^{\mathcal{P}})$.

4.2. Other reducibilities. The development in this paper exploits the analogy between the relations $\leqq_T^{\mathcal{NP}}$ and "recursively enumerable in." By the results of [1], however, this analogy should not be drawn too far, because $A \leqq_T^{\mathcal{NP}} B \,\&\, \bar{A} \leqq_T^{\mathcal{NP}} B$ does not imply $A \leqq_T^{\mathcal{P}} B$.

DEFINITION 7. $A \leqq_s B \leftrightarrow A \leqq_T^{\mathcal{NP}} B \,\&\, \bar{A} \leqq_T^{\mathcal{NP}} B$.

THEOREM 13. *$A \leqq_s B$ if and only if for every set C, $C \leqq_T^{\mathcal{NP}} A \rightarrow C \leqq_T^{\mathcal{NP}} B$.*

Proof. From $C \leqq_T^{\mathcal{NP}} A$, $A \leqq_T^{\mathcal{NP}} B$, and $\bar{A} \leqq_T^{\mathcal{NP}} B$, it is easy to construct the computation that makes $C \leqq_T^{\mathcal{NP}} B$. To prove the "if" part, simply take C to be A and then take C to be $\bar{A}$.

Compare this characterization of $\leqq_s$ with the original nonconstructive definition of $\leqq_{pe}$ (Definition 5).

Let co-$\mathcal{NP} = \{A \mid \bar{A} \text{ in } \mathcal{NP}\}$.

COROLLARY 5. *$\leqq_s$ is reflexive and transitive. The class $\mathcal{NP} \cap$ co-$\mathcal{NP}$ is the* **0***-degree for the $\leqq_s$ degree structure.*

The question of whether $\mathcal{P} = \mathcal{NP} \cap$ co-$\mathcal{NP}$ is open. The characterization of $\leqq_s$ in Theorem 13 can be strengthened to the following.

THEOREM 14. *$A \leqq_s B$ if and only if for every set C,*

$$C \leqq_T^{\mathcal{NP}} A \,\&\, \bar{C} \leqq_T^{\mathcal{NP}} A \rightarrow C \leqq_T^{\mathcal{NP}} B.$$

Thus, $A \leqq_s B \leftrightarrow \forall C[C \leqq_s A \rightarrow C \leqq_T^{\mathcal{NP}} B]$.

From this we derive the following corollary.

COROLLARY 6. *$\leqq_s$ is a maximal transitive subrelation of $\leqq_T^{\mathcal{NP}}$.*

Proof. Let $\mathcal{R}$ be a relation so that $\leqq_s \subsetneq \mathcal{R} \subseteq \leqq_T^{\mathcal{NP}}$. Choose A and B so that $A \not\leqq_s B$ and $A\mathcal{R}B$. Then, there is a set C so that $C \leqq_s A$ and $C \not\leqq_T^{\mathcal{NP}} B$. $C\mathcal{R}A$ and $A\mathcal{R}B$, but not $C \leqq_T^{\mathcal{NP}} B$. Therefore, $\mathcal{R}$ is not transitive.

To summarize, $\leqq_{pe}$ is a maximal transitive subrelation of $\leqq_T^{\mathcal{NP}}$ that contains the hierarchy of positive reducibilities such as $\leqq_m^{\mathcal{P}}$, $\leqq_m^{\mathcal{NP}}$, and $\leqq_c^{\mathcal{NP}}$. On the other hand, $\leqq_s$ is a maximal transitive subrelation of $\leqq_T^{\mathcal{NP}}$ that contains the hierarchy of deterministic reducibilities such as $\leqq_{tt}^{\mathcal{P}}$ and $\leqq_T^{\mathcal{P}}$.

5. Functions and characteristic functions. Given a function f, let $\mathbf{f} = \{\langle x, y\rangle | f(x) = y\}$. As mentioned in the Introduction, an important feature of enumeration reducibility is the relationship $f \leqq_T g \leftrightarrow \mathbf{f} \leq_e \mathbf{g}$. Theorem 15, to follow, is the analogue for $\leqq_{pe}$. Once again we point out that *all functions considered* are p-bounded, for some polynomial p.

We proceed by a sequence of lemmas. Also, we assume that the reader understands "mixed" reducibilities, for example, $A \leqq_T^{\mathcal{NP}} g$, $f \leqq_T^{\mathcal{NP}} A$, and $f \leqq_c^{\mathcal{NP}} A$.

LEMMA 5. (1) $\mathbf{f} \leq_T^{\mathcal{P}} f$,
(2) $f \leqq_c^{\mathcal{NP}} \mathbf{f}$.

The constructions required are straightforward.

LEMMA 6. (1) $A \leqq_T^{\mathcal{NP}} f \to A \leqq_c^{\mathcal{NP}} \mathbf{f}$.
(2) $g \leqq_T^{\mathcal{NP}} f \to g \leqq_c^{\mathcal{NP}} \mathbf{f}$.
(3) $A \leqq_T^{\mathcal{NP}} \mathbf{f} \to A \leqq_T^{\mathcal{NP}} f$.
(4) $A \leqq_T^{\mathcal{NP}} B \leftrightarrow A \leqq_T^{\mathcal{NP}} C_B$.

Proof. We indicate the proof of statement (1); the others are similar. Let M be an oracle acceptor with function oracle f that witnesses $A \leqq_T^{\mathcal{NP}} f$. Design M' with a set oracle to simulate M such that if M enters the oracle call state with a word x written on the oracle input tape, then M' guesses a word y with $|y| \leqq p(|x|)$ (the p-bound of f), writes $\langle x, y\rangle$ on its query tape, and enters the query state. If the M oracle returns y, then M' continues to simulate M. Otherwise, M halts without accepting. Therefore, only positive questions are asked of the $\mathbf{f}$ oracle. So, $A \leqq_c^{\mathcal{NP}} \mathbf{f}$.

In the following Lemma 7, the objects computed are changed. The constructions are again straightforward.

LEMMA 7. (1) $\mathbf{f} \leqq_T^{\mathcal{NP}} g$ *implies* $f \leqq_T^{\mathcal{NP}} g$.
(2) $f \leqq_c^{\mathcal{NP}} A$ *implies* $\mathbf{f} \leqq_c^{\mathcal{NP}} A$.

LEMMA 8. $A \leqq_{pe} B$ *if and only if, for every function* g, $B \leqq_T^{\mathcal{NP}} g$ *implies* $A \leqq_T^{\mathcal{NP}} g$.

Proof. Suppose $A \leqq_{pe} B$ and $B \leqq_T^{\mathcal{NP}} g$. By Lemma 6(1), $B \leqq_T^{\mathcal{NP}} \mathbf{g}$. Therefore, $A \leqq_T^{\mathcal{NP}} \mathbf{g}$. Lemma 6(3) gives $A \leqq_T^{\mathcal{NP}} g$. The proof in the other direction follows immediately from Lemma 6(4) and the definition of $\leqq_{pe}$.

THEOREM 15. $f \leqq_T^{\mathcal{NP}} g$ *if and only if* $\mathbf{f} \leqq_{pe} \mathbf{g}$.

Proof. Suppose $\mathbf{f} \leqq_{pe} \mathbf{g}$. By Lemma 5(1), $\mathbf{g} \leqq_T^{\mathcal{P}} g$. Therefore, by use of Lemma 8, $\mathbf{f} \leqq_T^{\mathcal{NP}} g$. $f \leqq_T^{\mathcal{NP}} g$ follows from Lemma 7(1).

Conversely, suppose $f \leqq_T^{\mathcal{NP}} g$. Then, by Lemma 6(2), $f \leqq_c^{\mathcal{NP}} \mathbf{g}$, and, by Lemma 7(2), $\mathbf{f} \leqq_c^{\mathcal{NP}} \mathbf{g}$. Therefore, $\mathbf{f} \leqq_{pe} \mathbf{g}$, using Theorem 11.

As a consequence of Theorem 15, $\leqq_T^{\mathcal{NP}}$ is a transitive relation over the class of p-bounded functions, very unlike the situation for sets.

Specializing to characteristic functions, we have the following.

LEMMA 9. $C_A \leqq_T^{\mathcal{NP}} B \leftrightarrow A \leqq_s B$.

THEOREM 16. $\mathbf{C}_A \leqq_{pe} \mathbf{C}_B \leftrightarrow A \leqq_s B$.

Recalling (Definition 7) that $A \leqq_s B$ if and only if $A \leqq_T^{\mathcal{NP}} B$ and $\bar{A} \leqq_T^{\mathcal{NP}} B$, we see that the construction needed to prove the lemma is clear. Theorem 16 follows from the previous theorem and Lemma 9.

Theorems 15 and 16 give credence to the correctness of the definitions involved. We conclude this paper with the following open problem.

Open problem. Define A is *strong polynomial time enumeration reducible* to B ($A \leqq_{\text{spe}} B$) if and only if there is a nondeterministic oracle Turing machine acceptor M that witnesses $A \leqq_T^{\mathcal{NP}} B$ and for every polynomial q there is a constant k such that, for each input string x to M, every query z about membership in B that receives a "no" answer satisfies $q(|z|) \leqq k \log |x|$.

Lemma 4, as an example, gives sets A and B so that $A \leqq_{\text{spe}} B$. Show that $\leqq_{\text{spe}} \neq \leqq_{\text{pe}}$.

Acknowledgment. The author acknowledges the hours of indispensable conversation with his colleague Theodore P. Baker. Also, the author wishes to express his gratitude to E. W. Leggett, jr. for finding an error in a previous version of this paper.

REFERENCES

[1] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the $\mathcal{P} =? \mathcal{NP}$ question*, this Journal, 4 (1975), pp. 431–442.

[2] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. of Third Annual ACM symposium on Theory of Computing, 1971, pp. 151–158.

[3] A. COBHAM, *The intrinsic computational difficulty of functions*, Proc. 1964 Congress for Logic, Mathematics, and Philosophy of Science, North-Holland, Amsterdam, 1965, pp. 24–30.

[4] R. L. CONSTABLE, *Type two computational complexity*, Proc. of Fifth Annual ACM Symposium on Theory of Computing, 1973, pp. 108–121.

[5] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, Miller and Thatcher, eds., Plenum Press, New York, 1973.

[6] E. W. LEGGETT, JR., *$\mathcal{P}$-completeness, $\mathcal{NP}$-completeness, and the polynomial hiearchy*, abstract, personal correspondence, 1975.

[7] R. LADNER, N. LYNCH AND A. L. SELMAN, *Comparison of polynomial-time reducibilities*, Proc. of Sixth Annual Symposium on Theory of Computing, 1974.

[8] ———, *A comparison of polynomial-time reducibilities*, Theoret. Comput. Sci., 1 (1975).

[9] K. MELHORN, *Polynomial and abstract subrecursive classes*, Proc. of Sixth Annual ACM Symposium on Theory of Computing, 1974, pp. 96–109.

[10] A. R. MEYER AND L. J. STOCKMEYER, *The equivalence problem for regular expressions with squaring requires exponential space*, 13th Annual IEEE Symposium of Switching and Automata Theory, 1972, pp. 125–129.

[11] J. R. MYHILL, *Note on degrees of partial functions*, Proc. Amer. Math. Soc., 12 (1961), pp. 519–521.

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

[13] A. L. SELMAN, *Arithmetical reducibilities I*, Z. Math. Logik Grundlagen Math., 17 (1971), pp. 335–350.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0006 $01.00/0

SOME POLYNOMIAL AND INTEGER DIVISIBILITY PROBLEMS ARE *NP*-HARD*

DAVID PLAISTED†

Abstract. Most known *NP*-complete problems are stated in terms of one of an exponential number of possibilities being true. In this paper we exhibit some *NP*-hard and *NP*-complete problems which are not of this form. In addition, we exhibit some new *NP*-complete problems of a more conventional nature. Many of these problems involve divisibility properties of integers or of sparse polynomials with coefficients of ± 1. This paper extends and refines earlier results of the author.

Key words. greatest common divisor, integer divisibility, integration, least common multiple, *NP*-complete, *NP*-hard, polynomial divisibility, sparse polynomials

In an earlier paper [3], the author showed that certain problems involving sparse polynomials and integers are *NP*-hard. In this paper we show that many related problems are also *NP*-hard. In addition, we exhibit some new *NP*-complete problems. Most of the new results concern problems in which the nondeterminism is "hidden." That is, the problems are not explicitly stated in terms of one of a number of possibilities being true. Furthermore, most of these problems are in the areas of number theory or the theory of functions of a complex variable. Thus there is a rich mathematical theory that can be brought to bear. These results, together with the earlier results, therefore introduce a class of *NP*-hard and *NP*-complete problems different from those known previously.

This paper is organized as follows: First, we summarize the results of the earlier paper. Then we present some implications of these results for determining polynomial divisibility. Next, we present some new *NP*-complete problems related to the given *NP*-hard problems. Then we present some extensions of the earlier results. In particular, we show that all the problems are still *NP*-hard if the nonzero coefficients of the polynomials are restricted to be $+1$ or -1. Some new *NP*-hard integer divisibility problems are discussed next. Finally, an *NP*-hard problem involving contour integrals and an *NP*-complete problem involving ordinary integrals are presented. Suggestions for further research are given.

We say that a problem A, considered as a set of input strings which should be accepted, is *polynomial transformable* (or *reducible*) to a problem B if there is a function f such that $f(x)$ is computable from x in time polynomial in the length of x, and such that $f(x) \in B$ iff $x \in A$. We say that a problem is *NP-hard* if every problem in *NP* is polynomial transformable to it. This definition of *NP*-hard differs from that used in [3]. Note that if any *NP*-hard problem has a polynomial time solution, then $P = NP$. We say that a problem is *NP-complete* if it is in *NP* and is *NP*-hard. We describe problems informally. For example, we refer to $\{\langle x, y\rangle : x > y \text{ and } x, y \in \mathbb{Z}\}$ as the problem of determining whether x is greater than y, given integers x and y.

Summary of previous results. The following problems have recently been shown to be *NP*-hard [3]. (Assume all polynomials are sparse polynomials with integer coefficients.)

P1. Given a finite set $\{p_1(x), \cdots, p_k(x)\}$ of polynomials and an integer N, to determine
   a) Whether the least common multiple of the set of polynomials is not $x^N - 1$.

* Received by the editors October 26, 1976, and in final revised form February 20, 1978.
† Department of Computer Science, University of Illinois, Urbana, Illinois 61801.

b) Whether the degree of the least common multiple of the set of polynomials is not N.

c) Whether the degree of the greatest common divisor of the set of polynomials is not zero.

P2. Given an integer N and a finite set of polynomials, to determine whether $x^N - 1$ is not a factor of the product of the polynomials in the set.

P3. Given a finite set of polynomials and an integer N, to determine whether the product of the polynomials in the set has fewer than N distinct (complex) zeros.

P4. To determine whether an exponential expression of integers is not a factor of another such expression. An exponential expression of integers is an expression formed from integers and the operations of addition, subtraction, multiplication, and exponentiation. (This problem was called P6 in [3].)

Note that P2 may well be in *NP* and may therefore be *NP*-complete. For example, if for some small integers b and q we have $b^N - 1 \equiv 0 \pmod{q}$ but $\prod_j p_j(b) \not\equiv 0 \pmod{q}$, then $x^N - 1$ is not a factor of $\prod_j p_j(x)$.

A remark. We now observe a consequence for polynomial divisibility algorithms of the fact that P2 is *NP*-hard. Suppose we are given a set $\{p_1(x), \cdots, p_k(x)\}$ of polynomials. If we are also given small integers b and q, we can determine $(\prod_j p_j(b)) \pmod{q}$ in polynomial time. This implies certain restrictions on algorithms for determining if a polynomial $t_1(x)$ is a factor of a polynomial $t_2(x)$, as follows: Suppose such an algorithm existed that made use of $t_1(x)$ explicitly but used $t_2(x)$ only indirectly. Suppose it only used the degree of $t_2(x)$ and the remainder $t_2(b) \pmod{q}$ for small integers b and q. Then this same algorithm could be applied to P2. Hence if $P \neq NP$ then no such algorithm exists that evaluates $t_2(b) \pmod{q}$ a small number of times on small integers b and q. To be precise, if L is the length of the representation of $t_2(x)$, then no such algorithm can exist that uses a number of evaluations of $t_2(b) \pmod{q}$ that is polynomial in the log of L, involving only integers b and q whose logarithm is polynomial in $\log(L)$. (We are assuming that essentially all of the work in the algorithm involves evaluating $t_1(b) \pmod{q}$ and $t_2(b) \pmod{q}$ for various b and q.)

Some *NP*-complete problems. We now show that the following problems are *NP*-complete:

Q1. Given sequences $\alpha_1, \cdots, \alpha_{k1}$ and $\beta_1, \cdots, \beta_{k2}$ of positive integers, to determine whether the polynomial $\prod_{j=1}^{k1} (x^{\alpha j} - 1)$ is *not* a factor of the polynomial $\prod_{j=1}^{k2} (x^{\beta j} - 1)$.

Q2. Given sequences $\alpha_1, \cdots, \alpha_{k1}$ and $\beta_1, \cdots, \beta_{k2}$ of positive integers, to determine whether there exists a positive integer b such that $\{j: b \text{ divides } \alpha_j\}$ has more elements than $\{j: b \text{ divides } \beta_j\}$.

Q3. Given sequences $R_1, \cdots, R_{k1}$ and $S_1, \cdots, S_{k2}$ of subsets of some (finite) set S, to determine whether there exists a subset T of S such that $\{j: T \subset R_j\}$ has more elements than $\{j: T \subset S_j\}$.

Q4. Given sequences $Y_1, \cdots, Y_{k1}$ and $Z_1, \cdots, Z_{k2}$ of m-tuples of nonnegative integers, to determine whether there exists an m-tuple V of nonnegative integers such that $\{j: V \leqq Y_j\}$ has more elements than $\{j: V \leqq Z_j\}$, where $V \leqq Z$ iff $V_i \leqq Z_i$ for $i = 1, \cdots, m$.

(We use sequences rather than sets to allow the same element to occur more than once.) The fact that Q4, Q3, and Q2 are *NP*-complete follows easily from the *NP*-completeness of Q1. Therefore we first show that Q1 is *NP*-complete.

The problem Q1 is in *NP* since it suffices to test the polynomials at appropriate roots of unity. More precisely, let $P_A(z)$ be $\prod_{j=1}^{k1} (z^{\alpha j} - 1)$ and let $P_B(z)$ be $\prod_{j=1}^{k2} (z^{\beta j} -$

1). Now, $P_A(z)$ is a factor of $P_B(z)$ iff for all roots ω of $P_A(z)$, ω is also a root of $P_B(z)$ and the multiplicity of the zero of $P_B(z)$ at ω is greater than or equal to the multiplicity of the zero of $P_A(z)$ at ω. But the zeroes of $P_A(z)$ are just roots of unity of the form $e^{2i\pi a/\alpha j}$ for integers a, $1 \leqq a \leqq \alpha j$ and for $1 \leqq j \leqq k1$. Thus we only need to test roots of unity of the form $e^{2i\pi m1/m2}$ where $m1$ and $m2$ are relatively prime, $1 \leqq m1 \leqq m2$, and $m2$ divides at least one of the αj, $1 \leqq j \leqq k1$. Then $P_A(z)$ is a factor of $P_B(z)$ iff for all such roots of unity ω, the number of j such that $\omega^{\alpha j} - 1 = 0$ is less than or equal to the number of j such that $\omega^{\beta j} - 1 = 0$. But $(e^{2i\pi m1/m2})^a - 1 = 0$ iff $m2$ divides a, with $m1$ and $m2$ as above, and we can determine this in polynomial time. Hence $P_A(z)$ is *not* a factor of $P_B(z)$ iff there exists an integer b such that $\{j\colon (b|\alpha j)\}$ has more elements than $\{j\colon (b|\beta j)\}$. Thus Q1 is in *NP*.

Furthermore, Q1 is *NP*-complete. This follows from results in [3]. For convenience, we now summarize the relevant earlier results:

Given a set $S = \{C1, \cdots, Ck\}$ of 3-literal clauses over the predicate symbols $\{P_1, \cdots, P_n\}$, we showed how to construct sparse polynomials Poly $(C1), \cdots,$ Poly (Ck) and Poly $(\neg C1), \cdots,$ Poly $(\neg Ck)$ having the following properties:

1. The Poly (Cj) and Poly $(\neg Cj)$ are computable in polynomial time and have integer coefficients.
2. For all j, $1 \leqq j \leqq k$, Poly $(Cj) *$ Poly $(\neg Cj) = x^N - 1$ where N is the product of the first n primes.
3. $x^N - 1$ is a factor of $\prod_{j=1}^{k}$ Poly $(\neg Cj)(x)$ iff S is inconsistent.
4. For all j, $1 \leqq j \leqq k$, Poly (Cj) can be expressed as Num (Cj)/Denom (Cj) where Num (Cj) and Denom (Cj) are products of four or less polynomials of the form $x^b - 1$ and b is a product of distinct primes.

(Actually, Poly is a function mapping from arbitrary propositional calculus formulae over the predicate symbols $\{P_1, \cdots, P_n\}$ onto integer-coefficient divisors of the polynomial $x^N - 1$.)

From the above results, it follows that S is consistent iff

$$(x^N - 1) \prod_{j=1}^{k} (\text{Num } (Cj)(x)) \text{ does not divide}$$

$$\prod_{j=1}^{k} ((x^N - 1) * \text{Denom } (Cj)(x)).$$

But this last problem is an instance of Q1. Hence Q1 is *NP*-hard, since determining whether such a set S is consistent is *NP*-hard. Hence Q1 is *NP*-complete.

Now, Q2 is clearly in *NP*. Also, we essentially showed above that Q1 is reducible to Q2. Hence Q2 is *NP*-complete.

Furthermore, Q3 is clearly in *NP*. In addition, the integers in the sequences $\alpha_1, \cdots, \alpha_{k1}$ and $\beta_1, \cdots, \beta_{k2}$ of Q2 are all products of distinct primes from among the first n primes, in all instances of Q2 obtained from Poly $(C1), \cdots,$ Poly (Ck) as indicated above. Hence these integers may be represented as sets of prime numbers, and divisibility may be represented by the subset relation. Thus Q1 is reducible to Q3, and so Q3 is *NP*-hard. Hence Q3 is *NP*-complete.

Finally, Q4 is clearly in *NP* and is also clearly *NP*-hard since it is a generalization of Q3. In particular, we represent a subset R of some universal set $\{a_1, \cdots, a_k\}$ by a vector V in which $V_i = 1$ if $a_i \in R$, and $V_i = 0$ otherwise. Thus Q4 is *NP*-complete.

Refinements of P1–P4. Now we refine the problems P1 through P4 in several ways to obtain more *NP*-hard problems. First we show that P1, P2, and P3 are all still *NP*-hard if the nonzero coefficients of the polynomials are restricted to be +1 or −1. In order to do this, it is necessary to refer back to the polynomials Poly (Cj) and Poly $(\neg Cj)$ (with notation as before). We showed in [3] that a set $S = \{Cl, \cdots, Ck\}$ of clauses is inconsistent iff any of the following (equivalent) conditions are true:

1. lcm {Poly $(\neg C1), \cdots,$ Poly $(\neg Ck)\} = x^N - 1$;
2. degree (lcm {Poly $(\neg C1), \cdots,$ Poly $(\neg Ck)\}) = N$;
3. degree (gcd {Poly $(C1), \cdots,$ Poly $(Ck)\}) = 0$;
4. $x^N - 1$ divides $\prod_{1 \leqq j \leqq k}$ Poly $(\neg Cj)$;
5. $\prod_{1 \leqq j \leqq k}$ Poly $(\neg Cj)$ has N distinct complex zeros.

(Here N is the product of the first n primes, as before.) Also, we remarked in the previous paper that for a 3-literal clause C, Poly (C) and Poly $(\neg C)$ have all coefficients either 0, +1, or −1 except in one case, namely, when C is all-positive. However, an all-positive clause $P_a \vee P_b \vee P_c$ can easily be replaced by $P_a \vee P_b \vee \bar{P}_d$ and $P_d \vee P_c$ without affecting the consistency of S, where P_d is a new predicate symbol. Furthermore, Poly $(P_d \vee P_c)$ and Poly $(\neg(P_d \vee P_c))$ will only have +1, 0, and −1 as coefficients, it turns out. Therefore, if S is a set of 3-literal clauses, the all-positive 3-literal clauses of S may all be eliminated as above to obtain a new set $S1$ of clauses. Also, for all clauses C of $S1$, Poly (C) and Poly $(\neg C)$ will all have coefficients of +1, 0, and −1. Furthermore, $S1$ is consistent iff S is. Thus P1, P2, and P3 are still *NP*-hard when all nonzero coefficients are restricted to be +1 or −1.

In addition, instead of choosing the first n primes to work with, we could have chosen any n relatively prime integers. For example, we could have chosen the mth powers of the first n primes, for fixed integer m. This implies that Q1 is still *NP*-complete when we restrict all exponents to be mth powers of distinct primes, for fixed m. Many more such restrictions on the exponents could be made.

The problem P3 can also be modified in the following way:

P3.1 Given a set $\{p_1(x), \cdots, p_k(x)\}$ of polynomials and an integer m, to determine the number of distinct complex zeros of $\prod_{j=1}^{k} p_j(x)$ having multiplicity *exactly* m.

This modified problem, although not strictly *NP*-hard, has the property that if it is in P then $P = NP$. The reason is that all the complex zeros of the polynomials Poly (Cj) and Poly $(\neg Cj)$ have multiplicity exactly one. Therefore the number of distinct complex zeros of $\prod_{j=1}^{k}$ Poly $(\neg Cj)$ can be determined by summing the number of zeros of multiplicity m, for $1 \leqq m \leqq k$. (The multiplicity of a zero essentially represents the number of clauses contradicting some interpretation.) Also, P3.1 is still *NP*-hard in the extended sense if we restrict all nonzero coefficients to be ±1.

Integer divisibility results. We now refine P4 in two ways. In [3] we showed that for any integer b with $b \geqq 4^k p_n^{4k}$, S is inconsistent iff $b^N - 1$ is a factor of $\prod_{j=1}^{k}$ Poly $(\neg Cj)(b)$. (Here p_n is the nth prime.) Let us choose a small integer m such that $2^m \geqq 4^k p_n^{4k}$. Such an m will be $O(k \log n)$. Let Poly$_m$ $(\neg Cj)$ be Poly $(\neg Cj)$ with all exponents multiplied by m. Then $2^{mN} - 1$ divides $\prod_{j=1}^{k}$ Poly$_m$ $(\neg Cj)(2)$ iff S is inconsistent. Similar results can be obtained for any integer (or Gaussian integer) whose absolute value is greater than one. (A Gaussian integer is a complex number whose real and imaginary parts are integers.) Hence for all integers (and Gaussian integers) b with $|b| > 1$, the following problem is *NP*-hard:

P4.1 Given an integer N and a set $\{p_1(x), \cdots, p_k(x)\}$ of sparse polynomials with integer coefficients, to determine whether $b^N - 1$ does not divide $\prod_{j=1}^{k} p_j(b)$.

This is still *NP*-hard if we restrict the nonzero coefficients of the $p_j(x)$ to be ± 1 as before. Note that P4.1 may be in *NP* since we can test divisibility by a small prime easily.

The second refinement of P4 is similar to the first. We know from above that $2^{mN}-1$ divides $\prod_{j=1}^{k}$ Poly $(\neg Cj)(2^m)$ iff S is inconsistent. But recall that

$$\text{Poly}\,(\neg Cj)=\frac{(x^N-1)\,\text{Denom}\,(Cj)}{\text{Num}\,(Cj)}.$$

This implies that S is inconsistent iff $2^{mN}-1$ divides

$$\prod_{j=1}^{k}\frac{(2^{mN}-1)\,\text{Denom}\,(Cj)(2^m)}{\text{Num}\,(Cj)(2^m)}.$$

The same construction clearly works for all integers (and Gaussian integers) having absolute value greater than one. Therefore the following problem is *NP*-hard for all integers (and Gaussian integers) q with $|q|>1$:

P4.2 Given sequences $a_1, a_2, \cdots, a_{k1}$ and $b_1, b_2, \cdots, b_{k2}$ of positive integers, to determine whether $\prod_{j=1}^{k1}(q^{a_j}-1)$ is not a factor of $\prod_{j=1}^{k2}(q^{b_j}-1)$.

(We use sequences rather than sets to allow the same integer to occur more than once.) Note that P4.2 is *NP*-hard for *each* integer q with $|q|>1$. Therefore it is *NP*-hard to determine whether

$$\prod_{j=1}^{k1}(2^{a_j}-1) \text{ is not a factor of } \prod_{j=1}^{k2}(2^{b_j}-1), \text{ for example.}$$

The problem P4.2 seems to be the most elegant of the problems considered here. This problem is still *NP*-hard if we restrict the a_j and b_j to be products of distinct primes, products of squares of distinct primes, et cetera. It is not known whether P4.2 is in *NP*.

Contour integrals. We now show that certain problems involving the evaluation of contour integrals are *NP*-hard. Specifically, the following problem is *NP*-hard:

P5. Given integers b and N and a set $\{p_1(Z), \cdots, p_k(Z)\}$ of sparse polynomials with integer coefficients, to determine whether the following contour integral is nonzero:

$$\int_C \frac{\prod_{j=1}^{k} p_j(Z)}{Z^N-1}\left(1-\frac{1}{Z^{kN}}\right)\frac{Z^N-b^N}{Z-b}\,dZ.$$

Here C is any contour including the origin in its interior.

Proof. We show that 3-consistency is reducible to P5. Suppose $S=\{C1, \cdots, Ck\}$ is a set of 3-literal clauses. Let $P_s(x)$ be $\prod_{j=1}^{k}$ Poly $(\neg Cj)$, with notation as before. Let $q_s(x)$ and $r_s(x)$ be defined by the equation $P_s(x)=q_s(x)(x^N-1)+r_s(x)$, where $r_s(x)$ is of degree less than N and N is the product of the first n primes. We showed in [3] that $r_s(x)\equiv 0$ iff S is inconsistent. Furthermore, if an integer b satisfies $|b|\geqq 4^k p_n^{4k}$ then $r_s(b)=0$ iff $r_s(x)\equiv 0$.

Suppose $r_s(x)=\sum_{j=0}^{N-1} a_j x^j$. We can show that

$$a_j=\frac{1}{2\pi i}\int_C \frac{P_s(Z)}{Z^N-1}\left(1-\frac{1}{Z^{kN}}\right)Z^{N-j-1}\,dZ,$$

where C is as above. We show this as follows: $P_s(Z)=q_s(Z)(Z^N-1)+r_s(Z)$, so

$$\frac{P_s(Z)}{(Z^N-1)}\left(1-\frac{1}{Z^{kN}}\right)Z^{N-j-1}=q_s(Z)Z^{N-j-1}-q_s(Z)Z^{N-j-1}/Z^{kN}$$
$$+\frac{r_s(Z)}{(Z^N-1)}\left(1-\frac{1}{Z^{kN}}\right)Z^{N-j-1}.$$

Now, the degree of $P_s(Z)$ is less than or equal to $k(N-1)$, so the degree of $q_s(Z)$ is less than or equal to $k(N-1)-N$ or $(k-1)N-k$. Since $q_s(Z)Z^{N-j-1}$ is analytic, it will not contribute to the above contour integral. Also, $q_s(Z)Z^{N-j-1}/Z^{kN}$ will not contribute because the degree of $q_s(Z)$ is less than or equal to $(k-1)N-k$. In particular, the largest power of Z in $q_s(Z)Z^{N-j-1}/Z^{kN}$ will never be larger than $[(k-1)N-k]+N-1-kN$, which is equal to $-k-1$. Since $k\geqq 1$, this exponent is less than or equal to -2 and therefore the coefficient of Z^{-1} will be zero.

The only term that will contribute to the contour integral is

$$\frac{1}{2\pi i}\int_C \frac{r_s(Z)}{(Z^N-1)}\left(1-\frac{1}{Z^{kN}}\right)Z^{N-j-1}\,dZ,$$

which equals

$$\frac{1}{2\pi i}\int_C r_s(Z)[Z^{-N}+Z^{-2N}+\cdots+Z^{-kN}]Z^{N-j-1}\,dZ.$$

Since the degree of $r_s(Z)$ is less than N, and since $0\leqq j<N$, the only term that will contribute to the integral is

$$\frac{1}{2\pi i}\int_C r_s(Z)Z^{-N}Z^{N-j-1}\,dZ$$

which equals $1/(2\pi i)\int_C r_s(Z)Z^{-j-1}\,dZ$. We now see that the coefficient of Z^{-1} in $r_s(Z)Z^{-j-1}$ is a_j. It follows from the residue theorem [2, pp. 129–131] that the value of the above contour integral is a_j as claimed. Therefore $r_s(b)=\sum_{j=0}^{N-1} a_j b^j$ is given by

$$\frac{1}{2\pi i}\int_C \frac{P_s(Z)}{Z^N-1}\left(1-\frac{1}{Z^{kN}}\right)\sum_{j=0}^{N-1} b^j Z^{N-j-1}\,dZ.$$

However, $\sum_{j=0}^{N-1} b^j Z^{N-j-1}=(Z^N-b^N)/(Z-b)$, and so we have reduced the problem of 3-consistency to the problem P5. Therefore P5 is *NP*-hard. (This is because the contour integral is zero iff $r_s(b)$ is zero, which is true iff $r_s(x)\equiv 0$, which is true iff S is inconsistent.)

Note as before that P5 is still *NP*-hard if we restrict the nonzero coefficients of the $p_j(Z)$ to be ± 1. Also, we can replace b by 2 in P5 without affecting the fact that P5 is *NP*-hard. This can be shown as usual by the device of multiplying all exponents of the polynomials $p_j(Z)$ by some integer m such that $2^m\geqq 4^k p_n^{4k}$. Similarly, we could replace b by 3 or any real or complex number whose absolute value is larger than 1.

There are at least two possible approaches to solving P5. Let $f(Z)$ be

$$\left(\frac{\prod_{j=1}^{k} p_j(Z)}{Z^N-1}\right)\left(1-\frac{1}{Z^{kN}}\right)\left(\frac{Z^N-b^N}{Z-b}\right).$$

Now, $f(Z)$ has a finite Laurent expansion about the origin. The contour integral of $f(Z)$ is zero iff the coefficient of Z^{-1} in $f(Z)$ is zero. But if this coefficient is zero, then we can integrate $f(Z)$ term by term to obtain $F(Z)$ such that $(d/dZ)F(Z)=f(Z)$ and

such that $F(Z)$ also has a finite Laurent expansion about the origin. Otherwise, no such function F exists. Hence one approach to solving P5 is to exhibit a function F as above. It might be interesting to determine when F can be expressed in a length polynomial in the length of the input to P5.

Another approach to solving P5 is to perform numerical evaluation of the contour integral. However, this also seems quite difficult.

Ordinary integrals. The following problem is slightly related to the preceding problems and is *NP*-complete:

P6. Given a set $\{a_1, \cdots, a_k\}$ of integers, to determine whether

$$\int_0^{2\pi} \cos(a_1\theta) \cdots \cos(a_k\theta)\, d\theta \neq 0.$$

Proof. Consider the function $g(x) = \prod_{j=1}^{k} (x^{a_j} + x^{-a_j})$. Note that $g(e^{i\theta}) = 2^k \prod_{j=1}^{k} \cos(a_j\theta)$. Thus the above integral is nonzero iff $\int_0^{2\pi} g(e^{i\theta})\, d\theta$ is nonzero. If $g(Z) = \sum_j b_j Z^j$ then $\int_0^{2\pi} g(e^{i\theta})\, d\theta$ equals $\sum_j b_j \int_0^{2\pi} e^{ij\theta}\, d\theta$ which equals $2\pi b_0$. Hence the constant term in the power series expansion of g is nonzero iff the above integral is nonzero. But this term is nonzero iff there is a partition of $\{a_1, \cdots, a_k\}$ into two sets $A1$ and $A2$ such that $\sum\{a: a \in A1\} = \sum\{a: a \in A2\}$. And this is just the partition problem, which is known to be *NP*-complete [1].

Conclusions. Some problem in algebra and number theory have been shown to be *NP*-hard or *NP*-complete. Many of these problems are "natural" in the sense that the nondeterminism is hidden. Together with earlier results [3], they represent a new class of *NP*-hard and *NP*-complete problems.

These problems are all based on an encoding of propositional calculus formulae into sparse polynomials with integer coefficients. Can this encoding be extended to first-order predicate calculus formulae in some way? If so, new complexity and undecidability results might be obtained.

REFERENCES

[1] R. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[2] K. KNOPP, *Theory of Functions Part I*, Dover, New York, 1945.

[3] D. PLAISTED, *Sparse complex polynomials and polynomial reducibility*, J. Comput. System Sci., 14 (1977), pp. 210–221.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0007 $01.00/0

MINIMAL-COMPARISON 2,3-TREES*

ARNOLD L. ROSENBERG† AND LAWRENCE SNYDER‡

Abstract. Those 2,3-trees that are minimal in expected number of comparisons per access for a given number of keys are characterized. The characterization yields directly a linear-time algorithm for constructing a minimal-comparison 2,3-tree for a given sorted set of keys. Regrettably, the property of comparison minimality is incompatible with the earlier-studied property of node-visit optimality. Specifically, the two types of optimality can coexist in a K-key 2,3-tree only for sixteen values of K, none exceeding 32. In contrast, comparison-minimal node-visit-*pessimal* K-key 2,3-trees exist for just over half the possible values of K.

Key words. 2,3-trees, 3-2 trees, search trees, searching, key-comparisons, comparisons per access, comparison-minimal trees

1. Introduction. Balanced search trees are an attractive data organization for large files that must efficiently support the operations of accessing, insertion and deletion. The term "efficiently" here connotes a worst-case execution time that is proportional to the logarithm of the number of keys in the file. This gross time bound is valid for any of the numerous varieties of balanced search trees [2, § 6.2.3]. But, of course, a finer examination exposes substantive differences in ease of programming and in efficiency of execution both among the various species of balanced search trees and, indeed, within each species. Detailed investigation of these differences will enhance one's ability to choose informedly among competing file organizations. It is our aim here to continue the study begun in [3] of the differences in efficiency among equally capacious 2,3-trees.

A 2,3-tree (also known as 2-3 tree, or 3-2 tree) is a balanced search tree with a particularly simple rule for inserting/deleting keys (cf. [2, § 6.2.3]). Each internal node of a 2,3-tree has either 2 or 3 successors and contains one fewer key than it has successors; all paths from the root of the tree to a leaf are equally long. Due to the nonbinary nature of the tree, there are (at least) two natural measures of the efficiency of a 2,3-tree, namely, the expected number of *comparisons* per access and the expected number of *node-visits* per access; these measures are distinct since each ternary node can engender two comparisons. The former cost measure would likely be the more important when an entire tree resides in main memory on a cpu equipped with only binary comparators; the latter measure would likely be preeminent either if ternary comparators were available or if the tree had to be segmented and stored external to main memory so that any edge-crossing carried with it the danger of a page fault. The node-visit efficiency measure has been studied in depth in [3]: the maximally efficient 2,3-trees are characterized in that paper, a linear-time algorithm for constructing node-visit-optimal 2,3-trees is presented, and the differences in structure between the optimal trees and their "typical" forest-mates are exposed. In this paper, we conduct the corresponding analysis of the expected-comparison cost measure, with the outcome consisting of the same three basic components. Additionally, in this paper we study the relationships between the two measures of the cost of a 2,3-tree.

The remainder of the paper is organized in three sections. Section 2 introduces a nonstandard presentation of 2,3-trees tailored to our task of defining and studying the

* Received by the editors May 25, 1977.

† Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

‡ Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, New York. Permanently at: Department of Computer Science, Yale University, New Haven, Connecticut 06520.

comparison-cost of such trees. Section 3 is devoted to three main endeavors: in § 3.A, the comparison-minimal 2,3-trees are characterized structurally; in § 3.B, a linear-time algorithm for constructing minimal-comparison 2,3-trees from an ordered set of keys is developed from the characterization theorem; in § 3.C, the comparison-costs of optimal trees are compared with those of nonoptimal ones, but the development leaves open the question of how much cheaper is a comparison-minimal tree from a comparison-maximal one. Section 4 is devoted to comparing the node-visit cost measure of [3] to our cost measure. We find that there are only sixteen values of K, none exceeding 32, for which there is a K-key 2,3-tree that is optimal with respect to both cost measures. In contrast, for just half the possible values of K, *every* node-visit pessimal tree is comparison optimal (though, thankfully, the converse is not true), and for just over half the values of K, some node-visit pessimal tree is comparison optimal.

Directions for further research. Aside from the problem left open in § 3.C, two postulates in our framework, both discernible in Algorithm A in § 2.B, merit further thought. First, we predicate our development on the use of 3-outcome ($<$, $=$, $>$) comparators. How would our results differ had 2-outcome ($\leqq$, $>$) comparators been used? Second, we assume a fixed regimen for searching through a 2,3-tree; specifically, the first, smaller key in a ternary node is examined before the second key. How would the development change if the first key to be examined in a ternary node was sometimes the smaller, sometimes the larger? What would be an intelligent way to decide which to look at first?

2. 2,3-trees and their comparison costs.

2.A. Trees. Our investigation will be facilitated by a nonstandard presentation of 2,3-trees.

Let S be a set of strings over the alphabet A. We say that S is *prefix-closed* if the string $x \in A^*$ is in S whenever any *successor* $x\alpha$ of x is in S for some $\alpha \in A$. For each $x \in S$, we denote by $\sigma_S(x)$ the set

$$\sigma_S(x) = \{\alpha \in A : x\alpha \in S\}$$

of letters that can extend x in S; and by $|x|$ we denote the *length* of x.

Let λ, μ, ρ, l, r be abstract symbols that we shall think of as ternary-left, ternary-middle, ternary-right, binary-left, and binary-right, respectively.

(2.1) A *bi-ter* (for *bi*nary-*ter*nary) *tree* is a rooted, oriented tree whose nodes comprise a prefix-closed set N of strings over the alphabet $\{\lambda, \mu, \rho, l, r\}$ such that

- (a) N is a disjoint union $N = N_0 + N_2 + N_3$ where
  - (i) each $x \in N_0$ is called a *leaf* and has $\sigma_N(x) = \varnothing$;
  - (ii) each $x \in N_2$ is called a *binary* node and has $\sigma_N(x) = \{l, r\}$;
  - (iii) each $x \in N_3$ is called a *ternary* node and has $\sigma_N(x) = \{\lambda, \mu, \rho\}$;
- (b) each edge of the tree has the form $\{x, x\alpha\}$ for $x \in N$ and $\alpha \in \sigma_N(x)$; we call node $x\alpha$ a *son* of node x.

We view a bi-ter tree as comprising *levels*: the root of the tree resides at level 0; and, recursively, the sons of a node at level h reside at level $h+1$.

The trees that will concern us are special types of bi-ter trees. We are concerned, of course, with 2,3-trees; but two other types of trees will play a central role in the characterization theorem in § 3; hence, for economy we have defined the general notion of bi-ter tree.

(2.2) A 2,3-*tree* is a bi-ter tree all of whose leaves share the same nonzero length and, hence, reside at the same nonzero level. Thus, the root of a 2,3-tree is not a leaf, and all root-to-leaf paths in a 2,3-tree are equally lengthy.

2.B. The comparison-cost measure. In order to motivate our measure of the comparison-cost of a 2,3-tree, we must explain how such trees are used as search trees.

A set of K ordered keys is stored in a $(K+1)$-leaf 2,3-tree as follows. Each binary node of the tree receives one key; each ternary node receives two keys; leaf nodes are dummy placeholders and receive no keys. The key(s) in each node are strictly intermediate in value between the keys in the node's left and right subtrees; if the node is ternary, then the keys in its center subtree are strictly intermediate in value between the node's two keys. See Fig. 1.

For each nonleaf node x in a 2,3-tree, let $x[1]$ denote the smaller and $x[2]$ the larger key residing at node x; if x is binary, then $x[2]$ does not exist.

The procedure for searching a 2,3-tree (for retrieval, alteration, insertion, or deletion) is described in the following algorithm which derives from Knuth [2, § 6.2.3].

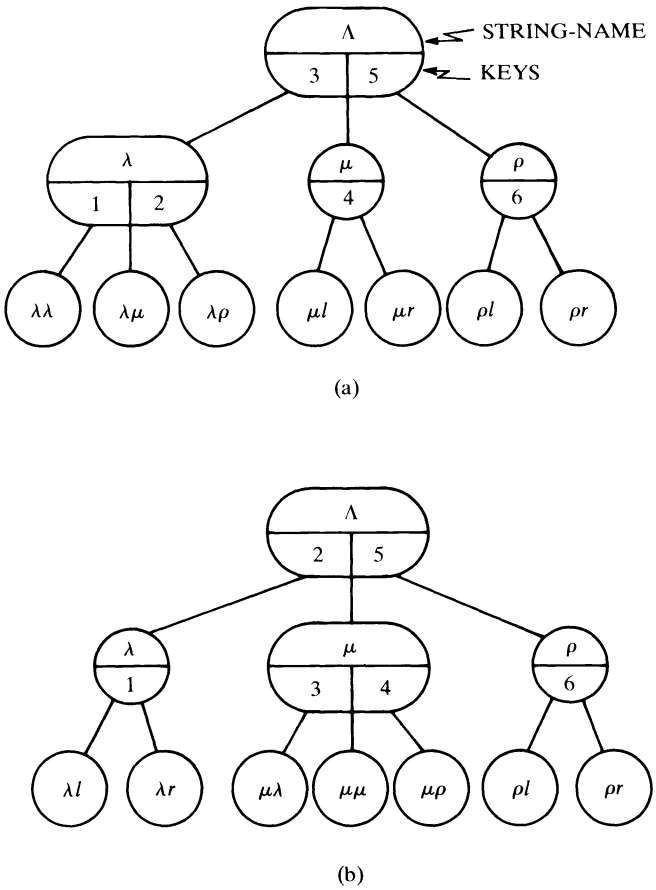
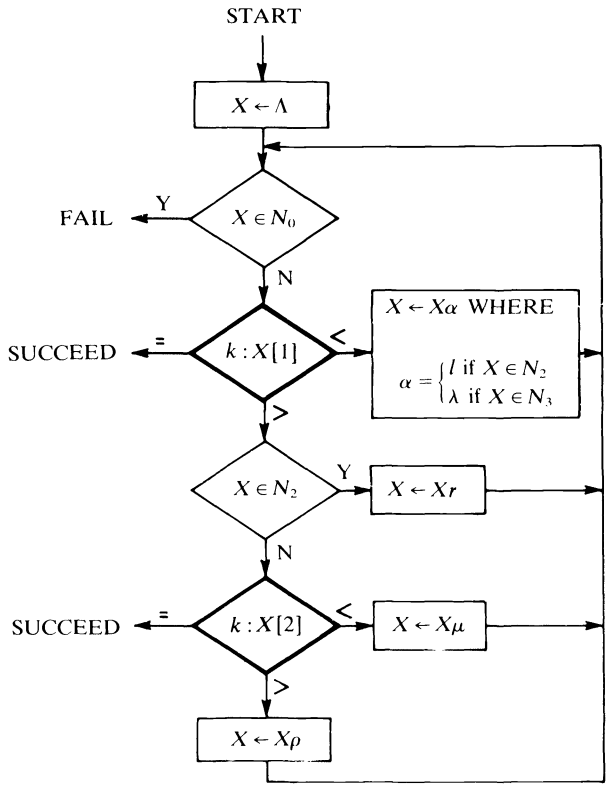


FIG. 1. *Two 6-key 2,3-trees with key sets* $\{1, \cdots, 6\}$ *and with each node's string-name exhibited.* Λ *denotes the empty string. In tree* (a), $N_0 = \{\lambda\lambda, \lambda\mu, \lambda\rho, \mu l, \mu r, \rho l, \rho r\}$, $N_2 = \{\mu, \rho\}$, *and* $N_3 = \{\Lambda, \lambda\}$; *in tree* (b), $N_0 = \{\lambda l, \lambda r, \mu\lambda, \mu\mu, \mu\rho, \rho l, \rho r\}$, $N_2 = \{\lambda, \rho\}$, *and* $N_3 = \{\Lambda, \mu\}$.

ALGORITHM A. Search for a key k in a 2,3-tree with node set $N = N_0 + N_2 + N_3$ and root Λ.



The two bold diamonds in the flowchart for Algorithm A denote the key-comparisons required to search a 2,3-tree. Note that a binary node $x \in N_2$ engenders precisely one key-comparison, namely, "$k: x[1]$", with the possible outcomes (i) $k < x[1]$, (ii) $k = x[1]$, (iii) $k > x[1]$. A ternary node $x \in N_3$ requires the comparison "$k: x[1]$", with outcomes (i)–(iii) possible, but, in the case of outcome (iii), x requires also the second key-comparison "$k: x[2]$" with possible outcomes (iv) $k < x[2]$, (v) $k = x[2]$, (vi) $k > x[2]$. In tree-oriented terminology, the possible outcomes from processing node x lead to the actions shown in Table 1.

TABLE 1

Outcome	Action	
	$x \in N_2$	$x \in N_3$
(i)	Search x's left subtree	Search x's left subtree
(ii)	Halt—success	Halt—success
(iii)	Search x's right subtree	Test $k: x[2]$
(iii) & (iv)	Not applicable	Search x's middle subtree
(iii) & (v)	Not applicable	Halt—success
(iii) & (vi)	Not applicable	Search x's right subtree

This analysis of Algorithm A motivates the following definition which will be useful in defining precisely our notion of the comparison-cost of a 2,3-tree.

(2.3) Let T be a 2,3-tree with node-set $N = N_0 + N_2 + N_3$; let Λ denote the empty (i.e., length 0) string. Define the function Place as follows:

(a) $\text{Place}(\Lambda[1]) = 1$.

(b) For $x \in N_2 + N_3$,

if $x \in N_2$, then $\text{Place}(xl[1]) = \text{Place}(xr[1]) = \text{Place}(x[1]) + 1$;

if $x \in N_3$, then $\text{Place}(x\lambda[1]) = \text{Place}(x[2]) = \text{Place}(x[1]) + 1$,

and $\text{Place}(x\mu[1]) = \text{Place}(x\rho[1]) = \text{Place}(x[1]) + 2$.

We leave to the reader the proof of the following result which ties together the function (2.3) and Algorithm A.

PROPOSITION 2.1. *Let the key k reside at node x of the 2,3-tree T; precisely, let $k = x[i]$, $i \in \{1, 2\}$. Exactly* $\text{Place}(x[i])$ *comparisons are needed to access k using Algorithm* A.

The foregoing renders natural the following measure of the comparison-cost of a 2,3-tree.

(2.4) The *comparison-cost* of a 2,3-tree T with node set $N = N_0 + N_2 + N_3$ is given by

$$\text{COST}(T) = \sum_{x \in N - N_0} \text{Place}(x[1]) + \sum_{x \in N_3} \text{Place}(x[2]).$$

Clearly $\text{COST}(T)$ is just K times the average number of comparisons needed to access a key in a $(K+1)$-leaf (hence K-key) 2,3-tree.

3. Minimal-comparison 2,3-trees. This section is devoted to three tasks. In § 3.A, we state and prove the main theorem of the paper, which characterizes structurally those 2,3-trees that have minimal comparison-costs for a given number of leaves (or, equivalently, of keys). Section 3.B contains the description and validation of a linear-time algorithm for constructing a minimal-comparison 2,3-tree for a given sorted list of keys. Finally, in § 3.C we compare minimal-comparison trees to their nonoptimal forest-mates.

3.A. The characterization theorem. Minimal-comparison 2,3-trees enjoy a simply stated characterization.

THEOREM MC. *A 2,3-tree T has minimal comparison-cost among trees with the same number of leaves if, and only if, T enjoys Property* M (for "minimal"):

Property M. Every ternary node of T has a string-name in the set $\{l, r, \lambda\}^*$ of $\{\mu, \rho\}$-less strings.

Theorem MC can be restated in purely tree-oriented terms:

THEOREM MC'. *A 2,3-tree T is comparison-minimal iff only binary nodes appear in the middle- and right-subtrees rooted at ternary nodes in T.*

Property M guarantees, for instance, the comparison-minimality of the tree of Fig. 1(a) while refuting that of the tree of Fig. 1(b). In fact, the comparison-cost of the former tree is 14 while that of the latter is 15.

The proof of Theorem MC requires some auxiliary notions and results.

(3.1) (a) A *binary tree* is a bi-ter tree (2.1) with $N_3 = \varnothing$, that is, one having no ternary nodes.

(b) A binary tree is *flat* if no two root-to-leaf paths in the tree differ in length by more than 1, or, equivalently, if no two leaves have string-names differing in length by more than 1.

One uses a binary tree as a search tree in much the same way that one so uses a 2,3-tree: one places one key in each nonleaf node of the tree in such a way that the key's value is strictly intermediate between the values of the keys in the node's left and right subtrees. Algorithm A will suffice to search through a tree so organized for a desired key. One determines easily (cf. [2, § 6.2.1]) that the average number of comparisons per access in a binary search tree is intimately related to the so-called *external path length* of the tree, from which fact one deduces the following, after having extended the notion of cost in (2.4) in the obvious way to binary trees.

LEMMA 3.1 [2, §§ 5.3.1, 6.2.1]. *A binary tree has minimal comparison-cost among equally capacious binary trees if, and only if, it is flat.*

Lemma 3.1 plays a central role in our proof of Theorem MC as we see now.

(3.2) Let T be a 2,3-tree with node set $N = N_0 + N_2 + N_3$. The *binarization* of T is the binary tree $\beta(T)$ defined as follows.

Let h be the string-homomorphism from $\{l, r, \lambda, \mu, \rho\}^*$ into $\{l, r\}^*$ defined by

$$h(l) = h(\lambda) = l,$$

$$h(r) = r,$$

$$h(\mu) = rl$$

$$h(\rho) = rr;$$

and extend h to strings in the obvious way.

The tree $\beta(T)$ has node set $\beta(N)$ obtained from N as follows.

(a) For each $x \in N_0 \cup N_2$, the string $h(x) \in \beta(N)$;

(b) for each $x \in N_3$, both $h(x)$ and $h(x)r \in \beta(N)$;

(c) provisos (a) and (b) account for all strings in $\beta(N)$.

The edges of $\beta(T)$ are as with any bi-ter tree.

The transformation β, which is depicted graphically in Fig. 2, is just a reformulation of Bayer's [1] representation of 2,3-trees as binary trees as reported in [2, § 6.2.3]. We leave to the reader the straightforward proof of the following crucial but simple proposition.

LEMMA 3.2. *For any 2,3-tree T, the transformation β is well-defined. Moreover, the binary tree $\beta(T)$ is equal in capacity (that is, holds the same number of keys, or, equivalently, has the same number of leaves) to T. Finally, the trees T and $\beta(T)$ have identical comparison-costs.*

We are now in a position to embark on our proof of Theorem MC.

Proof of Theorem MC. For any string $x \in \{l, r, \lambda, \mu, \rho\}^*$, we have, by (3.2),

$$|h(x)| = |x| + (\text{the number of } \mu\text{'s and } \rho\text{'s in } x). \tag{3.3}$$

Since the leftmost root-to-leaf path in a 2,3-tree comprises nodes with labels from the set $\{l, \lambda\}^*$, we deduce immediately from (3.3),

(3.4) The tree $\beta(T)$ is flat iff the tree T enjoys Property M.

This conclusion is immediate from the facts that no leaf in a 2,3-tree enjoying Property M contains more than a single occurrence of one letter from the set $\{\mu, \rho\}$, and that some leaf in a tree not enjoying the property must contain at least two letter-occurrences from the set.

Lemmas 3.1 and 3.2 combine with (3.4) to prove the sufficiency of Property M for comparison-minimality.

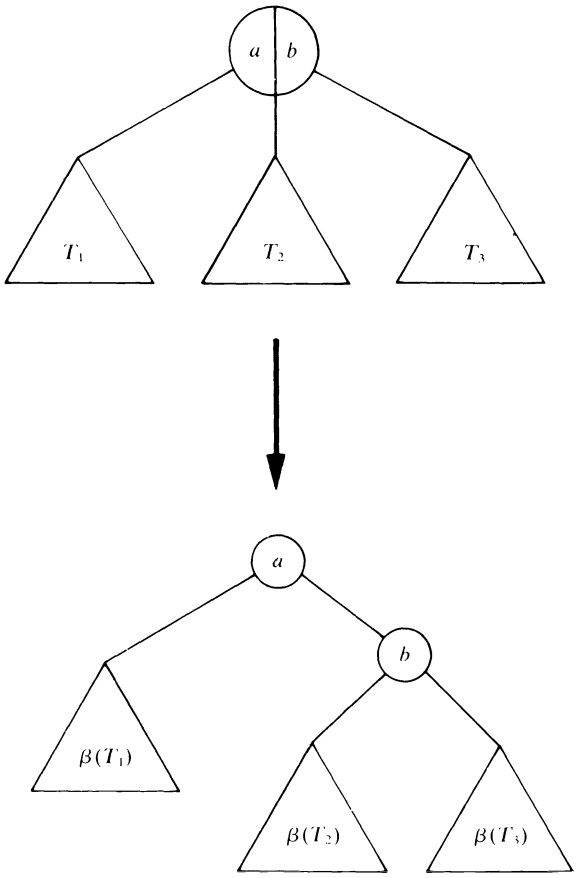


FIG. 2. *A schematic view of the binarization transformation* β *of* (3.2).

The necessity of Property M will follow from these same three sources as soon as we demonstrate the existence, for any number L of leaves, of an L-leaf 2,3-tree T whose binarization $\beta(T)$ is flat. By (3.4), it will suffice to show that some such tree T enjoys Property M. To this end, focus on any integer $L > 1$ and on its (minimal-length) binary representation

$$\alpha_0\alpha_1 \cdots \alpha_n. \tag{3.5}$$

PROCEDURE M. To construct an L-leaf 2,3-tree $T(L)$ with Property M: Let the 2,3-tree $T(L)$ have $n+1$ levels. All nodes of $T(L)$ save perhaps those on the leftmost root-to-leaf path are binary; hence, by design, $T(L)$ enjoys Property M! The nodes along the leftmost root-to-leaf path are binary or ternary according to the following rule:

(3.6) Let the path comprise, in this order, nodes $\nu_1, \nu_2, \cdots, \nu_{n+1}$ where ν_1 is the root and ν_{n+1} the leaf. Node ν_i is binary or ternary according as α_i in (3.5) is 0 or 1, respectively. Hence, the string-names of these nodes are given by

$$\text{name } (\nu_1) = \Lambda,$$

$$\text{name } (\nu_{i+1}) = \xi_1 \cdots \xi_i \quad \text{where } \xi_k = \begin{cases} l & \text{if } \alpha_k = 0, \\ \lambda & \text{if } \alpha_k = 1. \end{cases}$$

The reader can easily use Procedure M to construct the tree $T(7)$ in Fig. 1(a) and the tree $T(26)$ in Fig. 5(c) at the end of § 4. That $T(L)$ has L leaves is seen as follows. If $T(L)$ contained no ternary nodes, it would have 2^n leaves. By making node v_i in the left-most path of (3.6) ternary, we add 2^{n-i} leaves to the hitherto-constructed tree. Thus the tree constructed by rule (3.6) contains

$$\sum_{0 \leqq i \leqq n} \alpha_i 2^{n-i} = L$$

leaves. It is thus the desired L-leaf 2,3-tree with a flat binarization.

The necessity of Property M follows, completing the proof. □

3.B. Constructing minimal-comparison 2,3-trees. Procedure M, describing the skeletal structure of the minimal-comparison trees $T(L)$, translates into a linear-time algorithm for constructing a minimal-comparison 2,3-tree for a given sorted list of keys. In broad terms, the algorithm comprises two stages.

Stage 1. Given the cardinality c of the set of keys to be stored, use Procedure M to construct the "skeletal" tree $T(c+1)$. (Recall that a c-key tree has $c+1$ leaves.)

Stage 2. Visit the skeletal tree in FILLORDER [3, § 3.B], depositing keys in order in interior nodes as one goes. This order of traversal is described recursively by:

To visit a tree in FILLORDER,

1. visit the left subtree in FILLORDER;
2. visit the root and deposit a key;
3. visit the middle subtree in FILLORDER;
4. visit the root and deposit a key;

(3 and 4 for ternary roots only.)

5. visit the right subtree in FILLORDER.

Any implementation of this algorithm will likely perform Stages 1 and 2 in tandem for the sake of efficiency.

It is patently clear that the described algorithm operates in time $O(c)$ on a uniform-cost RAM.

3.C. Comparing comparison-costs. In this subsection we try to assess the savings afforded by minimal-comparison trees.

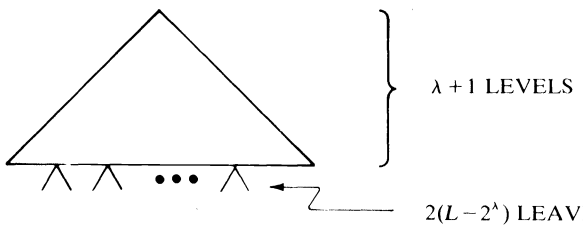
Minimal comparison-costs. Lemmas 3.1 and 3.2 provide us a vehicle for computing exactly the comparison-cost of any minimal-comparison 2,3-tree.

PROPOSITION 3.3. *Let T be a minimal-comparison L-leaf* 2,3*-tree. Then*

$$\text{COST}(T) = (\lambda+1)L - 2^{\lambda+1} + 1 \tag{3.7}$$

where $\lambda = \lfloor \log_2 L \rfloor$.

Proof. By dint of Lemmas 3.1 and 3.2, we can compute COST(T) by computing the comparison-cost of the flat L-leaf binary tree $\beta(T)$. Now, $\beta(T)$ is a $(\lambda+1)$-level ($\lambda = \lfloor \log_2 L \rfloor$) tree having 2^h nodes at each level $0 \leqq h \leqq \lambda$ and having $2(L-2^\lambda)$ nodes at level $\lambda+1$, as in the following figure.



Accordingly, the comparison-cost of $\beta(T)$, hence that of T, is given by:

$$\begin{aligned}\mathrm{COST}(T) &= \mathrm{COST}(\beta(T)) \\ &= \sum_{0 \leqq i < \lambda} (i+1)2^i + (\lambda+1)(L-2^\lambda) \\ &= (\lambda-1)2^\lambda + (\lambda+1)(L-2^\lambda)+1 \\ &= (\lambda+1)L - 2^{\lambda+1}+1. \quad \square\end{aligned}$$

We have not been able to find for maximal-comparison 2,3-trees a characterization that would allow us to determine definitively how much savings the minimal cost (3.7) represents. We do, however, have some information about these worst-case trees. A straightforward argument based on the fact that the tree $\beta(T)$ has at most twice as many levels as T proves the following

FACT.

$$\mathrm{COST}\begin{pmatrix}\text{comparison-minimal} \\ L\text{-leaf tree}\end{pmatrix} \geqq \frac{1}{2}\,\mathrm{COST}\begin{pmatrix}\text{comparison-maximal} \\ L\text{-leaf tree}\end{pmatrix},$$

thus bounding the improvement attainable from the minimal-comparison-tree algorithm just presented in § 3.B. Leo Guibas has informed the authors that he, Lyle Ramshaw, and Robert Sedgewick have found leaf cardinalities L for which

$$\mathrm{COST}\begin{pmatrix}\text{comparison-minimal} \\ L\text{-leaf tree}\end{pmatrix} \leqq .565\ \mathrm{COST}\begin{pmatrix}\text{comparison-maximal} \\ L\text{-leaf tree}\end{pmatrix},$$

.565 being an abbreviation of

$$\frac{3\log_2 3}{6\log_2 3 - 1}.$$

The preceding inequalities bracket in a rather small range the cost savings attainable by comparison-minimality.

The authors have discovered an explicit infinite family of 2,3-trees that have maximum costs for their capacities. While these maximal-cost trees are only half again as costly as equally capacious comparison-minimal trees, and are, thus, not as dramatic as the Guibas et al. trees as arguments for comparison-minimality, their simple recursive structure merits mention.

The trees $T[\ell]$. The trees we shall analyze are parameterized by their numbers of levels, starting with $\ell = 1$ for the 3-leaf ternary-rooted tree. Each tree $T[\ell]$ has a ternary root. Each of the tree's ternary nodes at level $k < \ell - 1$ has a left son that is binary and middle and right sons that are ternary; each of the tree's binary nodes at level $k < \ell - 1$ has two binary sons; all nodes at level $\ell - 1$ have leaves as sons. Thus, if we momentarily let l and λ revert to their meanings "binary-left" and "ternary-left" as in § 3.A, we find that the leaves of $T[l]$, $\ell \geqq 3$, have string-names in one of the sets

$$x\{l, r, \lambda\}\{l, r\} \quad \text{or} \quad x\{\mu, \rho\}\{\lambda, \mu, \rho\}$$

where x is the string-name of a leaf of $T[\ell - 2]$; see Fig. 3.

Now, one can establish easily by induction that the numbers of binary and ternary nodes at level k of $T[\ell]$, $k < \ell$, are given by

$$\mathrm{bin}(k) = 2^{k-1} + 2\,\mathrm{bin}(k-1) = k \cdot 2^{k-1}$$

$$\mathrm{tern}(k) = 2^k$$

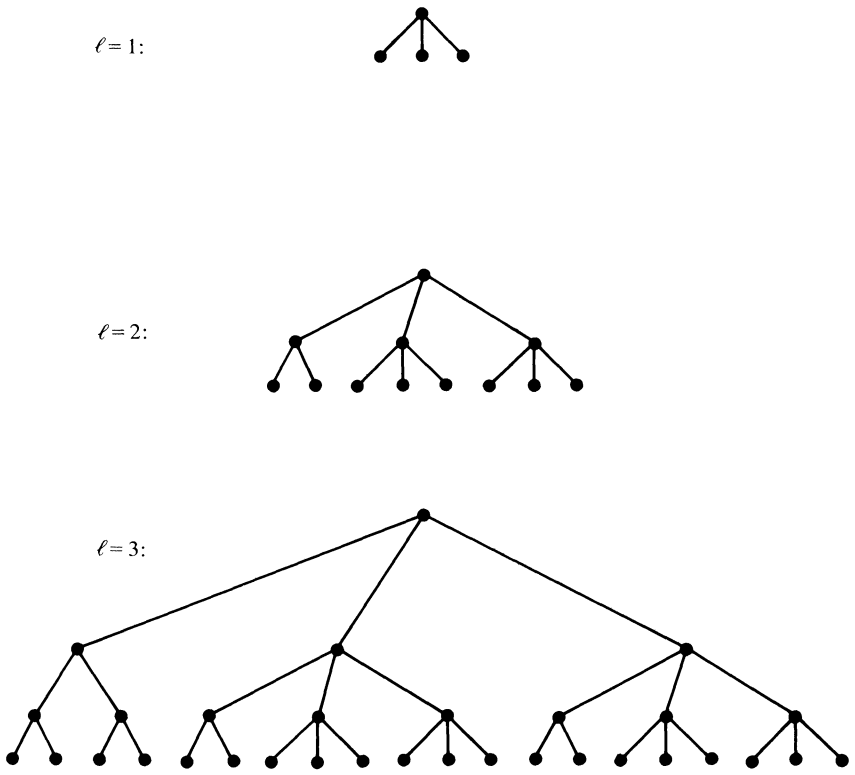


FIG. 3. *The first three trees* $T[\ell]$.

so that the number of leaves in $T[\ell]$ is given by

$$L(\ell) = (\ell+2)2^{\ell-1}. \tag{3.8}$$

The comparison-cost of $T[\ell]$. By dint of Lemma 3.2, we can evaluate the comparison-costs of the trees $T[\ell]$ by analyzing their binarizations $\beta(T[\ell])$. This analysis is simplified by noting the simple structure of these binary trees: $\beta(T[\ell])$ comprises a root whose left subtree is the complete $(\ell-2)$-level binary tree, and whose right subtree comprises a node both of whose subtrees are copies of $\beta(T[\ell-1])$; see Fig. 4. Accordingly, the number of keys at each level k of $\beta(T[\ell])$ is given (via a simple induction) by:

(a) for $0 \leqq k \leqq \ell-1$ there are 2^k keys at level k;

(b) for $\ell \leqq k \leqq 2\ell-1$ there are $2^{\ell-1}$ keys at level k; and, so,

$$\begin{aligned} \text{COST}(T[\ell]) = \text{COST}(\beta(T[\ell])) &= \sum_{0 \leqq k < \ell} (k+1)2^k + 2^{\ell-1} \cdot \sum_{\ell \leqq k < 2\ell} (k+1) \\ &= \ell 2^\ell - 2^\ell + 1 + 2^{\ell-2}(3\ell^2 + \ell) \\ &= 2^{\ell-2}(3\ell^2 + 5\ell - 4) + 1. \end{aligned} \tag{3.9}$$

To obtain a contrast to the costs (3.9) of the trees $T[\ell]$, we can use Procedure M from Theorem MC to construct a minimal-comparison tree $T(L(\ell))$ having the same number of leaves (3.8) as $T[\ell]$, and we can use equation (3.7) of Proposition 3.3 to

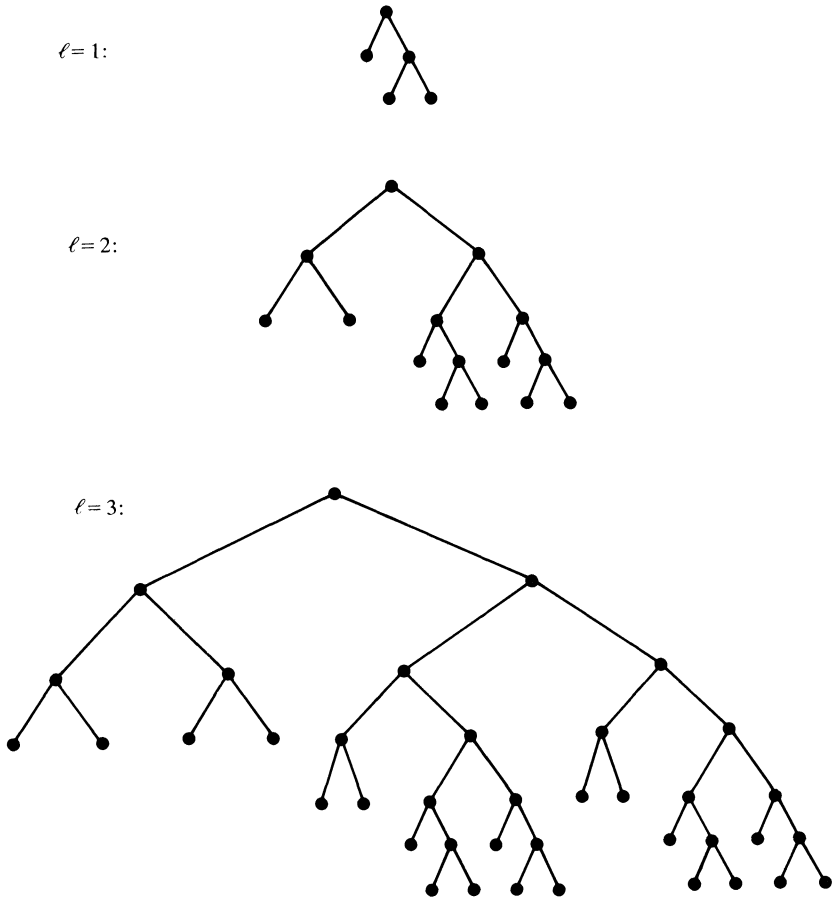


FIG. 4. *The binarizations of the first three trees* $T[\ell]$.

gauge the costs of these optimal trees. Letting $\lambda(\ell) = \lfloor \log_2 L(\ell) \rfloor$, we compute

$$\begin{aligned} \mathrm{COST}(T(L(\ell))) &= (\lambda(\ell)+1)L(\ell) - 2^{\lambda(\ell)+1} + 1 \\ &\leqq (\ell + \log_2(\ell+2))L(\ell) - L(\ell) + 1 \\ &= (\ell - 1 + \log_2(\ell+2))(\ell+2)2^{\ell-1} + 1 \\ &= 2^{\ell-2}(2\ell^2 + 2\ell - 4 + 2(\ell+2)\log_2(\ell+2)) + 1. \end{aligned} \tag{3.10}$$

From the analyses leading to equations (3.9) and (3.10) and from the fact that the trees $T[\ell]$ and $T(L(\ell))$ have the same number of leaves, we conclude the following, which suggests how much benefit comparison-minimality can yield, at least in the extreme.

PROPOSITION 3.4. *There is an infinite sequence of leaf-cardinalities*

$$L(1) < L(2) < \cdots,$$

(specifically, $L(\ell) = (\ell+2)2^{\ell-1}$*) with the property that*

$$\mathrm{COST}\left(\begin{matrix}\textit{comparison-minimal} \\ L(\ell)\textit{-leaf tree}\end{matrix}\right) \leqq \frac{2}{3}\mathrm{COST}\left(\begin{matrix}\textit{comparison-maximal} \\ L(\ell)\textit{-leaf tree}\end{matrix}\right) + O\left(\frac{\log \ell}{\ell}\right).$$

4. Comparing the two cost measures. The node-visit cost measure of [3] and our present cost measure are decidedly distinct in the sense that goodness relative to one measure is independent of goodness relative to the other, at least in general. However, the cost measures are not totally unrelated: one can often attain extrema of both measures simultaneously. This section is devoted to rendering these remarks precise and proving the resulting propositions.

4.A. The node-visit cost measure. A comparison of the two cost measures presupposes at least minimal familiarity with both. We now excerpt from [3] just enough information to render our comparison intelligible and rigorous. We refer the reader to [3] for all motivation and proofs and for a more detailed development.

(4.1) The *profile* of a $(d+1)$-level 2,3-tree is the sequence of integers $\Pi = \nu_0, \nu_1, \cdots, \nu_d$ where each ν_h is the number of nodes at level h of the tree.

(4.2) The *node-visit cost* of the 2,3-tree T having profile $\Pi = \nu_0, \cdots, \nu_d$ is given by

$$\mathrm{COST}_{\mathrm{nv}}(T) = d\nu_d - \sum_{i<d} \nu_i.$$

We call a 2,3-tree *bushy* if its nv-cost is minimal among 2,3-trees with the same number of leaves, and we call it *scrawny* if its nv-cost is maximal among the same population.

The cost in (4.2) may seem somewhat unintuitive. This expression is, in fact, derived in [3] from a more cumbersome but more intuitive equivalent.

THEOREM NV [3]. *Let the* 2,3*-tree* T *have profile* $\Pi = \nu_0, \nu_1, \cdots, \nu_d$.

(a) T *is bushy iff its profile is* dense *in the sense that*

$$\text{(4.3)} \qquad \begin{aligned} &\text{(a)} \quad d = \lceil \log_3 \nu_d \rceil; \\ &\text{(b)} \quad \nu_k = \min\left(3^k, \lfloor \nu_{k+1} \div 2 \rfloor\right), \qquad k = 0, \cdots, d-1. \end{aligned}$$

(b) T *is scrawny iff its profile is* sparse *in the sense that*

$$\text{(4.4)} \qquad \begin{aligned} &\text{(a)} \quad d = \lfloor \log_2 \nu_d \rfloor; \\ &\text{(b)} \quad \nu_k = \max\left(2^k, \lceil \nu_{k+1} \div 3 \rceil\right), \qquad k = 0, \cdots, d-1. \end{aligned}$$

The following corollary of Theorem NV is useful in characterizing bushy minimal-comparison trees.

COROLLARY 4.1 [3]. *If* $\Pi = \nu_0, \nu_1, \cdots, \nu_d$ *is a dense profile, then* $\nu_k = 3^k$ *for all* $k < d-2$.

4.B. Bushy minimal-comparison trees. The characterization of those values of L for which there exists an L-leaf bushy minimal-comparison tree takes a surprisingly simple form.

PROPOSITION 4.2. *There is a bushy* L*-leaf minimal-comparison* 2,3*-tree for precisely the following values of* L: $2 \leqq L \leqq 7$, $10 \leqq L \leqq 15$, *and* $28 \leqq L \leqq 31$.

Proof. We leave to the reader the straightforward verification that bushy minimal-comparison trees exist for the indicated sixteen values of L.

For the remaining values of L, an L-leaf bushy tree must have $\nu_1 = 3$ and $\nu_2 \geqq 8$, hence a ternary node at level 0 with at least two ternary sons, hence not be a minimal-comparison tree. These bounds on ν_1 and ν_2 can easily be checked directly

for $L \leqq 27$. For $32 \leqq L \leqq 81$, the bounds follow from the inequality

$$\lfloor L/4 \rfloor \geqq 8$$

together with (4.3)(b). For $L > 81$, the bounds are immediate from Corollary 4.1. □

4.C. Scrawny minimal-comparison trees. Theorem NV indicates that bushy 2,3-trees tend to be "as ternary as possible" while scrawny 2,3-trees tend to be "as binary as possible". One would expect, therefore, that scrawny trees have a much larger overlap with minimal-comparison trees than do bushy ones. In fact, it is a not unnatural conjecture that all minimal-comparison trees are scrawny. We shall see now that one's expectation is realized, though the conjecture is false. In sharp contrast with the finite overlap of bushiness with comparison-minimality, one finds that scrawniness and comparison-minimality overlap just over half the time.

PROPOSITION 4.3. *For any integer L, let $\lambda(L) = \lfloor \log_2 L \rfloor$. There is a scrawny L-leaf minimal-comparison 2,3-tree whenever L lies in the range*

$$2^{\lambda(L)} \leqq L \leqq 3 \cdot 2^{\lambda(L)-1} + 1. \tag{4.5}$$

Further, when L lies in the subrange

$$2^{\lambda(L)} \leqq L \leqq 3 \cdot 2^{\lambda(L)-1},$$

every *scrawny L-leaf 2,3-tree is comparison-minimal. Finally, for $L \geqq 8$, the sufficiency of lying in the range* (4.5) *is also necessary.*

Proof. Note that, for any integer L, an L-leaf scrawny 2,3-tree has depth $\lambda(L)$; cf. (4.4)(a). Our argument divides naturally into three parts.

Say first that L lies in the range

$$2^{\lambda(L)} \leqq L \leqq 3 \cdot 2^{\lambda(L)-1}.$$

For any L in this range,

$$\lceil L/3 \rceil \leqq 2^{\lambda(L)-1}.$$

By (4.4)(b) of Theorem NV, then, an L-leaf scrawny 2,3-tree has ternary nodes, if at all, only on level $\lambda(L) - 1$. By Theorem MC, then, every L-leaf scrawny tree is comparison-minimal.

Consider next the case

$$4 \leqq L = 3 \cdot 2^{\lambda(L)-1} + 1.$$

Since

$$\lceil L/3 \rceil = 2^{\lambda(L)-1} + 1,$$

equation (4.4)(b) assures us that in any L-leaf scrawny tree,

$$\nu_{\lambda(L)-1} = \lceil L/3 \rceil = 2^{\lambda(L)-1} + 1, \tag{4.6}$$

so that the tree has two binary nodes at level $\lambda(L) - 1$. Further, by (4.4)(b), level $\lambda(L) - 2$ of a scrawny L-leaf tree has a number of nodes

$$\nu_{\lambda(L)-2} = \max\left(2^{\lambda(L)-2}, \lceil (2^{\lambda(L)-1} + 1) \div 3 \rceil\right).$$

Now, for any integer $k \geqq 1$,

$$2^{k-1} \geqq \lceil (2^k + 1) \div 3 \rceil,$$

so, for $L \geqq 4$, we have

$$\nu_{\lambda(L)-2} = 2^{\lambda(L)-2}. \tag{4.7}$$

Equations (4.4)(b), (4.6), and (4.7) combine to specify completely the profile of any L-leaf scrawny tree: it has two binary nodes and the rest ternary nodes at level $\lambda(L)-1$; it has one ternary node and the rest binary nodes at level $\lambda(L)-2$; and it has only binary nodes at all lower-numbered levels. One of these trees, namely the one whose level $\lambda(L)-2$ ternary node has two binary sons, is easily seen via Theorem MC to be comparison-minimal.

We have established thus far, that, for any L in the range (4.5), there is a scrawny L-leaf minimal-comparison 2,3-tree. To complete the proof, we must show now that no such tree exists for L not in the indicated range. To this end, let L lie in the range

$$14 \leqq 3 \cdot 2^{\lambda(L)-1} + 2 \leqq L < 2^{\lambda(L)+1}. \tag{4.8}$$

We consider simultaneously the three possible forms of L:

$$\begin{aligned} &\text{(a)} \quad L = 3 \cdot 2^{\lambda(L)-1} + 3x, && 3 \leqq 3x < 2^{\lambda(L)-1}; \\ &\text{(b)} \quad L = 3 \cdot 2^{\lambda(L)-1} + 3y + 1, && 4 \leqq 3y + 1 < 2^{\lambda(L)-1}; \\ &\text{(c)} \quad L = 3 \cdot 2^{\lambda(L)-1} + 3z + 2, && 2 \leqq 3z + 2 < 2^{\lambda(L)-1}. \end{aligned} \tag{4.9}$$

Knowing only the range (4.8) of L, we can assert, using (4.4)(b), that, for any L-leaf scrawny tree,

$$\nu_{\lambda(L)-1} = \lceil L/3 \rceil.$$

Applying this information to the three cases in (4.9), we deduce

(4.10) In case (4.9)(a), the scrawny tree has only ternary nodes at level $\lambda(L)-1$. In case (4.9)(b), the tree has two binary and the rest ternary nodes at level $\lambda(L)-1$. In case (4.9)(c), the tree has one binary and the rest ternary nodes at level $\lambda(L)-1$. And, in all cases, the bound $L \geqq 14$ in (4.8) assures us that the tree has at least four ternary nodes at level $\lambda(L)-1$.

Continuing with our analysis, the upper bounds on x, y, and z in (4.9) combine with (4.4)(b) to assure us that, in any L-leaf scrawny tree,

$$\nu_{\lambda(L)-2} = 2^{\lambda(L)-2}.$$

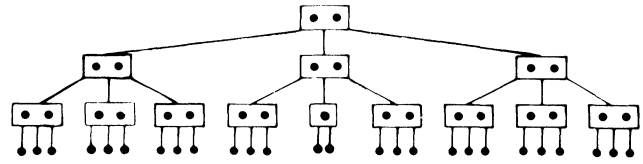
But this information exposes the entire profile of the tree. Specifically,

(4.11) On all levels $< \lambda(L)-2$, the tree has only binary nodes. (In fact, this is true for all scrawny trees by a dual to Corollary 4.1 proved in [3].)

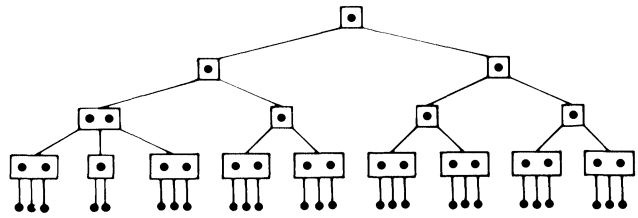
(4.12) In case (4.9)(a), the tree has $x \geqq 1$ ternary nodes at level $\lambda(L)-2$.
In case (4.9)(b), the tree has $y+1 \geqq 2$ ternary nodes at level $\lambda(L)-2$.
In case (4.9)(c), the tree has $z+1 \geqq 1$ ternary nodes at level $\lambda(L)-2$.

In all cases, (4.12) combines with (4.10) to guarantee that any scrawny L-leaf tree has at level $\lambda(L)-2$ at least one ternary node two of whose sons are also ternary. By Theorem MC no such tree is comparison-minimal. □

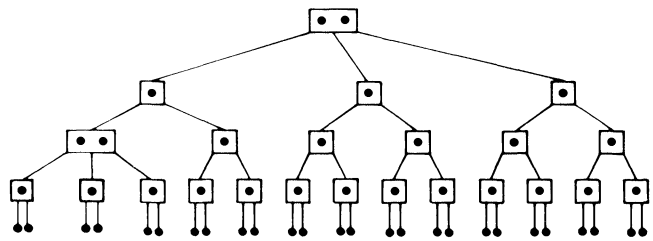
The reader can easily verify that $L = 26$ is the first place where bushiness, scrawniness, and comparison-minimality are mutually exclusive. Figure 5 exhibits (a) a bushy 26-leaf tree, (b) a scrawny 26-leaf tree, and (c) a minimal-comparison 26-leaf tree. The comparative costs of these trees are tabulated in Table 2. (The disparity in



(a)



(b)



(c)

FIG. 5. (a) *A bushy,* (b) *a scrawny, and* (c) *a minimal-comparison* 26*-leaf* 2,3*-tree. The trees* (a) *and* (b) *are minimal in comparison-cost among* 26*-leaf bushy and scrawny trees, respectively.*

c-costs could have been made even greater had we not chosen the minimal c-cost bushy and scrawny trees for Figure 5).

TABLE 2

	T_{bushy}	T_{scrawny}	T_{mc}
nv-cost	65	88	81
c-cost	103	100	99

4.D. A final difference between the cost measures. As a final remark, we note that minimal-comparison trees are somewhat more likely to occur "in nature" than are minimal-node-visit trees. Specifically, we noted in [3] that *no* sequence of 3^n key-insertions will give rise to a bushy 3^n-leaf 2,3-tree. In contrast, a minimal-comparison 2,3-tree is guaranteed to arise if one constructs a tree by inserting keys in decreasing order.

Acknowledgment. Conversations with R. E. Miller and N. Pippenger during the preparation of the quadruply-authored [3] made the topic of the current paper more accessible and more tractable to us. The authors are also grateful to L. Guibas for a number of interesting comments, most notably regarding § 3.C.

REFERENCES

[1] R. BAYER, *Binary B-trees for virtual memory*, Proc. ACM-SIGFIDET Workshop on Data Description, Access and Control, E. F. Codd and A. L. Dean, eds., San Diego, CA, Nov. 11–12, 1971, pp. 219–235.

[2] D. E. KNUTH, *The Art of Computer Programming. III: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[3] R. E. MILLER, N. PIPPENGER, A. L. ROSENBERG AND L. SNYDER, *Optimal 2,3-trees*, this Journal, to appear.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0008 $01.00/0

TREE SIZE BY PARTIAL BACKTRACKING*

PAUL W. PURDOM†

Abstract. Knuth [1] recently showed how to estimate the size of a backtrack tree by repeatedly following random paths from the root. Often the efficiency of his method can be greatly improved by occasionally following more than one path from a node. This results in estimating the size of the backtrack tree by doing a very abbreviated partial backtrack search. An analysis shows that this modification results in an improvement which increases exponentially with the height of the tree. Experimental results for a particular tree of height 84 show an order of magnitude improvement. The measuring method is easy to add to a backtrack program.

Key words. backtrack, partial backtrack, tree size, analysis of algorithms, Monte Carlo method, matrix multiplication, bilinear forms

1. Introduction. The author had been trying to develop an improved method for multiplying 3 by 3 matrices using a backtrack program when he heard of Knuth's [1] method of estimating the efficiency of backtrack programs. The author had planned to make extensive use of Knuth's method, but found that for his particular trees a huge number of runs were required to obtain even an order of magnitude estimate of the tree sizes. This experience was quite different from that of several others [1], [2]. The author finally determined that the important characteristic of his trees was their great height combined with the large fraction of nodes on each level which had no sons. Eventually the author discovered a slight modification of Knuth's algorithm which produces a large increase in efficiency.

Knuth [1] gives a good discussion of the uses of backtrack programs, which will not be repeated here. The method he proposes for measuring the size of a backtrack tree is to repeatedly follow random paths from the root and to estimate that the number of nodes in the tree is the average over several runs of $1+d_1+d_1d_2+\cdots$ where d_i is the number of successors to the node on level i. This gives an unbiased estimate of the number of nodes in the tree. It is particularly efficient when used to estimate the size of a complete binary tree of height h, since it concludes that the size is 2^h-1 after examining $2h-1$ nodes. On the other hand it has trouble with a tree in which each node has zero or two sons, and no more than one node per level has sons which have sons themselves. Part of such a tree is shown in Fig. 1, where only the nodes which Knuth's algorithm considers expanding (i.e., nodes with sons) are shown. The original tree can be obtained by adding two sons to each node with no sons. For such a tree Knuth's algorithm will examine an average of $7-2^{-h+3}$ nodes. While the expected value of the estimate is $4h-1$, the most frequent result of a single run will be 7. Knuth's algorithm would have to be run 2^{h-1} times to have a reasonable chance of examining the bottom level. The best way to find the size of a tree of the type shown in Fig. 1 is to examine all the nodes, using a complete backtrack algorithm.

Tall skinny trees can thus best be measured by doing a complete backtrack search, while short fat trees can be efficiently measured using Knuth's algorithm. For intermediate trees neither of these methods works well. The tree can have so many nodes that it is not practical to do a complete backtrack search, while at the same time it has so many nodes with no successors at each level that Knuth's algorithm has difficulty learning about the deeper levels of the tree. While Knuth [1] offers several suggestions that help in this situation, they are often difficult to apply, and none of

* Received by the editors March 29, 1977, and in final revised form February 21, 1978.
† Computer Science Department, Indiana University, Bloomington, Indiana 47401.

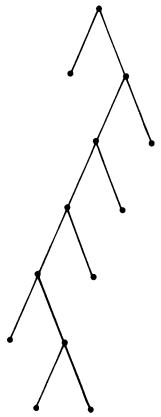


FIG. 1. *A tall skinny backtrack tree.*

them get to the heart of the problem. What is needed is an algorithm that will sample the deeper levels, which usually contain most of the nodes, with about the same probability as the upper levels. The way to increase the number of nodes observed at the deeper levels is to occasionally investigate more than one branch out of a node. If the average number of branches to follow is selected properly then one may be able to look at an average of about one node per level. Doing this will greatly improve the efficiency of the estimating program.

2. Backtrack measuring algorithms. A general backtrack program finds all the vectors $(x_1, x_2, \cdots, x_n)$ which satisfy some property $P_n(x_1, x_2, \cdots, x_n)$. There are also intermediate properties $P_k(x_1, x_2, \cdots, x_k)$ such that $P_{k+1}(x_1, x_2, \cdots, x_{k+1}) \supset P_k(x_1 \cdots x_k)$ for $0 \leqq k < n$. When $P_k(x_1, x_2, \cdots, x_k)$ is false, backtracking saves considerable work since no extension of $(x_1, x_2, \cdots, x_k)$ needs to be considered as a possible solution.

The following algorithm is the common backtrack algorithm with modifications for measuring tree sizes. The values x_k are kept on a stack, where k is the stack pointer. The parameter n is the length of the vectors $(x_1, x_2, \cdots, x_n)$ which satisfy the final property $P_n(x_1, x_2, \cdots, x_n)$. The function $C(x_1, x_2, \cdots, x_k)$ gives the cost of examining node $(x_1, x_2, \cdots, x_k)$. Setting C to one for all nodes results in a total cost equal to the number of nodes in the tree. The stack entry r_k contains the number of sons remaining to be selected from the node on level $k-1$. The variable t accumulates the estimated cost of the tree. To monitor the performance of the algorithm the optional arrays c_k and f_k accumulate the estimated cost of examining the nodes on level k and the number of nodes investigated on level k. For convenience the algorithm assumes that the values for x_k are consecutive integers starting with one. Only minor changes are needed to handle arbitrary discrete x_k.

BASIC BACKTRACK ESTIMATING ALGORITHM.

Step B1. [Initialize] Set k to 0, t to 0, and d_0 to 1. For $1 \leqq j \leqq n$ set c_j to 0 and f_j to 0.

Step B2. [Go down] If $k = n$ then output the solution $(x_1, x_2, \cdots, x_n)$ and go to step B6. Otherwise set k to $k+1$. Set a to the number of values that x_k can take on. If a is zero then go to step B6. Set m to number of values that will be investigated. This must be an integer such that $1 \leqq m \leqq a$. Set d_k to $d_{k-1}a/m$, r_k to m, and x_k to $a+1$.

Step B3. [More] If x_k is 1 then go to step B6.

Step B4. [Next] Set x_k to $x_k - 1$. With probability $1 - r_k/x_k$ reject x_k by going to

step B3. Otherwise set r_k to $r_k - 1$, t to $t + d_k C(x_1, x_2, \cdots, x_k)$, c_k to $c_k + d_k C(x_1, x_2, \cdots, x_k)$, and f_k to $f_k + 1$.

Step B5. [Test] If $P_k(x_1, x_2, \cdots, x_k)$ is true then go to step B2.

Step B6. [Go up] Set k to $k-1$. If $k = 0$ then stop. Otherwise go to step B3.

This algorithm is the traditional backtrack algorithm with additions at steps B1, B2, and B4 for measuring tree size. Step B4 has the algorithm for selecting m of a values at random [3], [4]. The efficiency of the algorithm depends on the value of m selected at step B2. How to select m will be considered in detail later. However m is selected, the expected value of the sum over all nodes visited on level k of $d_k C(x_1, x_2, \cdots, x_k)$ is the sum of the cost of the nodes on level k.

Knuth's algorithm for measuring tree size modifies the standard backtrack algorithm by first testing all the successors to a node before selecting which one to follow. The following algorithm combines this idea with partial backtracking. It replaces the stack r_k with a stack of sets S_k. The stack S_k contains the set of values remaining to be considered for x_k. The set Q contains all the values of x_k for which $P(x_1, x_2, \cdots, x_k)$ is true.

MODIFIED BACKTRACK ESTIMATING ALGORITHM.

Step M1. [Initialize] Set k to 0, t to 0, and d_0 to 1. For $1 \leqq k \leqq n$ set c_k to 0 and f_k to 0.

Step M2. [Go down] If $k = n$ output solution $(x_1, x_2, \cdots, x_n)$ and go to step M5. Set k to $k+1$. Set Q to the set of all x_k such that $P_k(x_1, x_2, \cdots, x_k)$ is true. For each x_k tested for inclusion in Q, set t to $t + d_{k-1} C(x_1, x_2, \cdots, x_k)$, and f_k to $f_k + 1$. Set a to the number of elements in Q. If Q is empty then go to step M5. Set m to the number of values that will be investigated, so that $1 \leq m \leq a$. Set d_k to $d_{k-1} a/m$.

Step M3. [Select values] Set S_k to a randomly selected subset of m values from Q.

Step M4. [Next] If S_k is empty then go to step M5. Otherwise remove an element from S_k and set x_k to the element. Go to step M2.

Step M5. [Go up] Set k t o $k-1$. If $k = 0$ then stop. Otherwise go to step M4.

The stack of sets S_k can be replaced by a stack of values if one is willing to test each node twice. Simplifications can also be made if $m \leqq 2$ at all times. The modified algorithm does not test P_0, which is nearly always true.

The modified algorithm looks at all the successors of a node, x_{k-1}, to see where $P_k(x_1, x_2, \cdots, x_k)$ is true, but follows only m of them during partial backtracking. If $P_k(x_1, x_2, \cdots, x_k)$ is always false when $(x_1, x_2, \cdots, x_k)$ has no successors (and is not a solution) then the tree of nodes considered for expansion by the modified algorithm can be obtained from the original backtrack tree by deleting the nonsolution nodes which have no successors. Usually the resulting tree will also have some nodes with no successors. Knuth's [1] algorithm is the modified algorithm with m always equal to one.

The efficiency of either version of the algorithm depends on how m is chosen. If it is always one, then one has essentially Knuth's original algorithm. If it is always set to a then one has complete backtracking. Small values of m make it difficult to observe deep levels in the tree, while large values of m require that a huge number of nodes be examined. There is often an intermediate value of m which will largely avoid the first problem without causing the second.

3. Analysis. Figure 2 shows the first three members of a sequence of sets of trees which are useful for studying the effect of the value of m. For each tree in a set, there is a probability indicating how often the tree is selected from the set. The set T has the tree 0, consisting of just a root, with probability one. For $i > 1$ set T_i has tree 0 with

probability $1-b$. It also has each tree of the form (t_1, t_2), which has a root, left subtree t_1, and right subtree t_2, where t_1 and t_2 are selected from T_{i-1}. The tree (t_1, t_2) occurs with probability $bp(t_1)p(t_2)$ where $p(t_1)$ is the probability that t_1 is selected from T_{i-1} and $p(t_2)$ is the probability that t_2 is selected from T_{i-1}. The parameter b controls the average size of the trees in T_i.

In the following the efficiency of a tree size estimating program will be measured by the product of the expected number of nodes examined (averaged over all trees in T_i) times the expected variance of the estimate of tree size (averaged over all trees in T_i). Since this is a measurement of work times error, low numbers indicate high efficiency. This quantity was selected because it is easy to calculate. Although other measures of efficiency might be better, the method of selecting m should apply with little change even with slightly different measures.

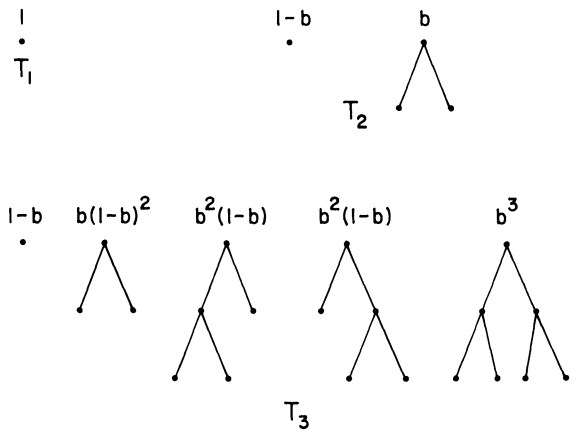


FIG. 2. *The trees in T_1, T_2, and T_3 with their associated probabilities.*

Let $p_{it}(n)$ be the probability that tree t has n nodes and that tree t is selected from T_i. Then $p_{it}(n)$ obeys the recurrence

$$p_{10}(n) = \delta_{n1},$$

$$p_{i0}(n) = (1-b)\delta_{n1} \quad \text{for } i > 1,$$

and

$$p_{i(t_1,t_2)}(n) = b\sum_m p_{i-1,t_1}(m)p_{i-1,t_2}(n-1-m) \quad \text{for } i > 1.$$

The expected number of nodes for a tree in T_i is

$$n_i = \sum_{n,t} np_{it}(n) = 1 + 2bn_{i-1} = \frac{(2b)^i - 1}{2b-1}.$$

The variance is given by

$$\begin{aligned} v_i &= \sum_{n,t} n^2 p_{it}(n) - n_i^2 \\ &= 1 + 8bn_{i-1} + 2bv_{i-1} \\ &= \frac{2(1-b)}{(2b-1)^3}[-2b - (2b-1)(2i-1)(2b)^i + (2b)^{2i}]. \end{aligned}$$

For most values of b and i the standard deviation is the same order of magnitude as the average.

When the basic algorithm selects m to be 1 with probability $1-p$ and to be 2 with probability p, it estimates the size of binary trees according to the recurrences

$$E(0)=1,$$

$$E(t_1, t_2)=1+\begin{cases}2E(t_1) & \text{with probability } (1-p)/2,\\ 2E(t_2) & \text{with probability } (1-p)/2,\\ E(t_1)+E(t_2) & \text{with probability } p.\end{cases}$$

Let $p_{it}(l)$ be the probability that l nodes are examined when estimating the size of t and that tree t is selected from T_i. Then

$$p_{i0}(l)=(1-b)\delta_{l1}$$

and

$$p_{i(t_1,t_2)}(l)=\frac{b}{2}(1-p)\Big[p_{i-1,t_1}(l-1)\sum_m p_{i-1,t_2}(m)+p_{i-1,t_2}(l-1)\sum_m p_{i-1,t_1}(m)$$
$$+bp\sum_m p_{i-1,t_1}(m)p_{i-1,t_2}(l-1-m) \quad \text{for } i>1.\Big]$$

The expected number of nodes examined (averaging over all trees in T_i and repeated runs of the algorithm) is

$$l_i=\sum_{l,t} lp_{it}(l)=1+b(1+p)l_{i-1}=\frac{[b(1+p)]^i-1}{b(1+p)-1}.$$

Let $p_{it}(e)$ be the probability that e is the estimate of the size of tree t and that tree t is selected from T_i. Then

$$p_{i0}(e)=(1-b)\delta_{e1}$$

and

$$p_i(t_1, t_2)=\frac{b}{2}(1-p)\Big[p_{i-1,t_1}\Big(\frac{e-1}{2}\Big)\sum_m p_{i-1,t_2}(m)+p_{i-1,t_2}\Big(\frac{e-1}{2}\Big)\sum_m p_{i-1,t_1}(m)\Big]$$

$$+pb\sum_m p_{i-1,t_1}(m)p_{i-1,t_2}(e-1-m)$$

The expected value of the estimate is

$$e_i=\sum_{e,t} ep_{it}(e)=1+2be_{i-1}=\frac{(2b)^i-1}{2b-1}.$$

The expected value of the square of the estimate is

$$s_i=\sum_{e,t} e^2p_{it}(e)=1+2b(2e_{i-1}+pe_{i-1}^2)+2b(2-p)s_{i-1}$$

$$=\frac{1}{(2b-1)^2}\Big[\frac{4b^4-2bp-1}{4b-2bp-1}+\frac{2(3-p)(1-b)(2b-1)^2}{(2b-2p-1)(p-1)(2b+p-2)}[2b(2-p)]^i$$
$$+\frac{2[2b-p-1]}{p-1}(2b)^i+\frac{p(4b^2)^i}{2b+p-2}\Big].$$

The total variance can be calculated from s_i. Of more interest, however, is the internal variance, which measures the expected variation of the estimate for the size of a tree about the expected value of the estimate of the size of that tree. The internal

variance is equal to the total variance minus the variance in the size of the various trees about the expected tree size. Furthermore, variance in the size of the trees is independent of p while the internal variance is zero for $p = 1$. Therefore, the internal variance is

$$v_i = s_i - s_i(p=1)$$
$$= \frac{2(1-b)}{(2b-1)^3}\left[\frac{4b^2(1-p)}{4b-2bp-1} + (2b-1)\left[\frac{1+p}{1-p}+2i\right](2b)^i + \frac{1-p}{2b+p-2}(4b^2)^i \right.$$
$$\left. + \frac{(2b-1)^3}{(4b-2bp-1)(1-p)(2b+p-2)}[2b(2-p)]^i\right].$$

The efficiency of the algorithm is indicated by $l_i v_i$, where a low value indicates high efficiency. For $\frac{1}{2} \leqq b \leqq 1$ and i large there are three values of p of interest. When p is small the term that grows like $[2b(2-p)]^i$ dominates the variance. The size of this term decreases with increasing p. The number of nodes examined is small if $b(1+p) \leqq 1$. Therefore, $p = b^{-1} - 1$ is one value of interest. The $[2b(2-p)]^i$ term continues to dominate as long as $4b^2 \leqq 2b(2-p)$. Therefore, $p = 2 - 2b$ is a second value of interest. Finally, for $p = 1$ the variance is zero. For $b^{-1} - 1 \leqq p \leqq 2 - 2b$ both extreme values for p produce local minima in $l_i v_i$, provided that one is above $\frac{1}{2}$ and the other is below $\frac{1}{2}$. If both extreme values are on the same side of $\frac{1}{2}$, then only the more distant one produces a local minimum. For $\frac{1}{2} \leqq b < \frac{1}{2}\sqrt{2}$, $p = 2 - 2b$ produces the lower value, while for $\frac{1}{2}\sqrt{2} < b \leqq 1$, $p = b^{-1} - 1$ produces the lower value. For $\frac{2}{3} \leqq b \leqq \frac{1}{2}\sqrt{2}$ one may want to set p to $b^{-1} - 1$ to avoid looking at the larger number of nodes required when $p = 2 - 2b$, just as one usually does not use $p = 1$ because it requires looking at a prohibitive number of nodes.

A constant value of p is often not best. Rather than making many runs with $p = b^{-1} - 1$ or $2 - 2b$, it is better to set $p = 1$ for several consecutive levels starting at the root and to set $p = b^{-1} - 1$ or $2 - 2b$ for the remaining levels. In practice one would want to make at least 3 runs since it is also important to know the variance of the estimate.

Now consider the modified algorithm. Let $p_{it}(l)$ be the probability that l nodes are examined when estimating the size of tree t and that tree t is selected from T_i. Then

$$p_{i0}(l) = (1-b)\delta_{l1}, \qquad p_{i(0,0)}(l) = b(1-b)^2\delta_{l3},$$
$$p_{i(t_1,0)}(l) = b(1-b)p_{i-1,t_1}(l-2) \quad \text{for } t_1 \neq 0,$$
$$p_{i(0,t_2)}(l) = b(1-b)p_{i-1,t_2}(l-2) \quad \text{for } t_2 \neq 0,$$

and

$$p_{i(t_1,t_2)}(l) = \frac{b}{2}(1-p)\left[p_{i-1,t_1}(l-2)\sum_m p_{i-1,t_2}(m) + p_{i-1,t_2}(l-2)\sum_m p_{i-1,t_1}(m)\right]$$
$$+ bp\sum_m p_{i-1,t_1}(m)p_{i-1,t_2}(l-1-m) \quad \text{for } t_1 \neq 0 \quad \text{and} \quad t_2 \neq 0.$$

The expected number of nodes examined is

$$l_i = \sum_{l,t} l p_{it}.$$

Therefore $l_1 = 1$, $l_2 = 1 + 2b$, and for $i > 2$

$$l_i = 1 + b^2(1+p) + (2b - b^2(1-p))l_{i-1} = \frac{1 + b^2(1-p) - 2b[b(2-b(1-p))]^{i-1}}{1 - 2b + b^2(1-p)}.$$

Let $p_{it}(e)$ be the probability that the modified method estimates that tree t is of size e and that tree t is selected from T_i. Then

$$p_{i0}(e)=(1-b)\delta_{e1}, \qquad p_{i(0,0)}(e)=b(1-b)^2\delta_{e3},$$
$$p_{i(t_1,0)}(e)=b(1-b)p_{i-1,t_1}(e-2) \quad \text{for } t_1\neq 0,$$
$$p_{i(0,t_2)}(e)=b(1-b)p_{i-1,t_2}(e-2) \quad \text{for } t_2\neq 0,$$

and

$$p_{i(t_1,t_2)}(e)=\frac{b}{2}(1-p)\left[p_{i-1,t_1}\left(\frac{e-1}{2}\right)\sum_m p_{i-1,t_2}(m)+p_{i-1,t_2}\left(\frac{e-1}{2}\right)\sum_m p_{i-1,t_1}(m)\right]$$
$$+bp\sum_m p_{i-1,t_1}(m)p_{i-1,t_2}(e-1-m) \quad \text{for } t_1\neq 0 \quad \text{and} \quad t_2\neq 0.$$

The expected value of the estimate is

$$e_i=\sum_{e,t} ep_{it}(e).$$

Therefore $e_1=1$, $e_2=1+2b$, and for $i>2$,

$$e_i=1+2be_{i-1}=\frac{(2b)^i-1}{2b-1}.$$

The expected value of the square of the estimate is $s_i=\sum_{e,t} e^2p_{it}(e)$. Therefore $s_1=1$, $s_2=1+8b$, and for $i>2$

$$s_i=1-2b(1-p)+2b^2(1-p)+4b[2-p-b(1-p)]e_{i-1}+2bpe_{i-1}^2$$
$$+2b(1+b(1-p))s_{i-1}$$
$$=\frac{1}{(2b-1)^3}\left[\frac{1+2b-2b^2(3-p)-8b^2(1-b)(1-p)}{1-2b-2b^2(1-p)}\right.$$
$$+\frac{4(2b-1)^2(b-1)(b-bp+2)}{(1-2b-2b^2(1-p))(1-b(1+p))(1-p)}[2b(1+b(1-p))]^{i-1}$$
$$\left.+\frac{4[2-b(5-3p)+2b^2(1-p)]}{1-p}(2b)^{i-1}-\frac{4b^2p}{1-b(1+p)}(4b^2)^{i-1}\right].$$

The expected internal variance is

$$v_i=s_i-s_i(p=1).$$

For $i>2$,

$$v_i=\frac{4(1-b)}{(2b-1)^3}\left[\frac{4b^4(1-p)}{-1+2b+2b^2(1-p)}+(2b-1)\left[\frac{2-3b(1-p)}{1-p}-2ib\right](2b)^{i-1}\right.$$
$$-\frac{b^2(1-p)}{1-b(1+p)}(4b^2)^{i-1}$$
$$\left.+\frac{(2b-1)^3(2+b-bp)}{(-1+2b+2b^2(1-p))(1-b(1+p))(1-p)}[2b+1+b(1-p))]^{i-1}\right].$$

Again the efficiency is indicated by l_iv_i. For $\frac{1}{2}\leqq b\leqq 1$ and large i there are three values of p of interest. These are $p=(b^{-1}-1)^2$, $p=b^{-1}-1$, and $p=1$. For $(b^{-1}-1)^2\leqq p\leqq b^{-1}-1$ both endpoints produce local minima if b is near $\frac{1}{2}\sqrt{2}$. For $\frac{1}{2}\leqq b<\frac{1}{2}\sqrt{2}$, $p=b^{-1}-1$ produces the lower local minimum whereas for $\frac{1}{2}\sqrt{2}<b\leqq 1$, $p=(b^{-1}-1)^2$

produces the lower local minimum. The basic and modified algorithms have about the same efficiency when p is set to the best value for each. The dominant exponential terms have the same base and exponent, and the coefficients do not differ greatly in value. The analysis indicates that the modified algorithm is favored for $b > .7325$, and the basic algorithm for $b < .7325$, but this conclusion is dependent on detailed assumptions made in the analysis. Also, the basic algorithm is easier to program.

4. Experimental results. To test the practical application of these methods, a number of measurements were made. The first set was made on a backtrack program that found ways to multiply 2 by 2 matrices with 7 multiplications. The program was not very sophisticated and it produced a 106,283,567 node binary tree of height 84 (more details are given in the Appendix). Level 48 had the largest number of nodes (16,077,754). The results of these runs are given in Tables 1 and 2. Figures 3 and 4 show the efficiency of each method as a function of p, the probability that both branches of the binary tree are investigated. Figures 3 and 4 also have a least square fit of the theory for T_i trees to the data. The value of chi-square is 98.7 for Fig. 3 and 21.8 for Fig. 4. The large values of chi-square are probably caused by using measured standard deviations rather than exact values. For comparison Fig. 3 also has the curve for the parameters that give the best fit to the data in Fig. 4. The occasional large standard deviations result from the non-Gaussian nature of the distributions generated by the estimating process. The experimental results show that values of $p \neq 0$ can produce a considerable improvement in the efficiency of the estimation process. They also show that the theory for T_i trees provides a practical guide for selecting p.

The second set of runs was made with a slightly improved program which looked for methods to multiply 3 by 3 matrices using 22 multiplications. This program generated a gigantic tree of height 594 with about 10^{42} nodes. The level with the largest number of nodes is near level 300, which has 4×10^{41} nodes. By carefully

TABLE 1

Results of the basic method with various values of p. Numbers after the ± signs are standard deviations. Not all runs were useful for obtaining the standard errors for the last column.

p ($\times \frac{1}{16}$)	Runs	Nodes examined ($\times 10^6$)	Estimate ($\times 10^8$)	Nodes × variance ($\times 10^{21}$)
0	4.89×10^7	2.000	0.56 ± 0.18	68 ± 26
1	3.97×10^7	2.000	0.80 ± 0.30	176 ± 161
2	3.15×10^7	2.000	1.49 ± 0.74	1099 ± 1086
3	2.39×10^7	2.000	0.757 ± 0.055	6.02 ± 0.66
4	1.66×10^7	2.000	0.998 ± 0.072	10.4 ± 5.5
5	9.69×10^6	2.000	0.991 ± 0.027	1.49 ± 0.22
6	4.29×10^6	2.000	1.058 ± 0.036	2.65 ± 1.13
7	1.39×10^6	2.000	1.064 ± 0.030	1.83 ± 0.82
8	548355	3.000	1.064 ± 0.019	1.16 ± 0.46
9	85854	2.000	1.047 ± 0.018	0.70 ± 0.11
10	31541	3.322	1.049 ± 0.016	0.86 ± 0.08
11	5959	2.905	1.029 ± 0.026	1.95 ± 0.50
12	860	2.005	1.128 ± 0.057	6.4 ± 1.2
13	195	2.009	1.035 ± 0.074	11.0 ± 2.4
14	40	2.045	1.09 ± 0.10	21.1 ± 5.7
15	10	2.389	1.21 ± 0.22	119 ± 36
16	1	1.063	1.06283576	0

TABLE 2

Results of the modified method with various values of p. Not all runs were useful for obtaining the standard errors for the last column.

p ($\times\frac{1}{16}$)	Runs	Nodes examined ($\times 10^6$)	Estimate ($\times 10^8$)	Nodes × variance ($\times 10^{21}$)
0	3.85×10^6	2.000	0.999 ± 0.030	1.76 ± 0.60
1	2.63×10^6	2.000	1.046 ± 0.021	0.87 ± 0.11
2	1.61×10^6	2.000	1.064 ± 0.014	0.40 ± 0.28
3	860153	2.000	1.123 ± 0.053	5.6 ± 4.9
4	397856	2.000	1.064 ± 0.016	0.49 ± 0.11
5	161623	2.000	1.060 ± 0.019	0.76 ± 0.30
6	60744	2.000	1.077 ± 0.015	0.47 ± 0.04
7	21507	2.000	1.055 ± 0.019	0.73 ± 0.18
8	3778	1.000	1.094 ± 0.058	3.4 ± 2.2
9	1265	1.001	1.083 ± 0.044	1.97 ± 0.25
10	444	1.005	0.99 ± 0.13	4.5 ± 1.3
11	150	1.004	1.044 ± 0.075	6.0 ± 1.6
12	64	1.021	0.93 ± 0.13	17.6 ± 7.3
13	21	1.018	1.05 ± 0.14	19.8 ± 4.7
14	9	1.354	0.58 ± 0.38	199 ± 48
15	5	1.833	1.04 ± 0.15	89 ± 48

selecting a value of p for each level it was possible to estimate a size of 7.14×10^{42} with a standard deviation of 4.79×10^{42}. This required examining 7.8×10^8 nodes. A single value of p was unsuitable because of the variation in the best value at different levels in the tree. With previous methods it would have been impossible to obtain even a rough estimate of the size of this tree.

The experimental data was collected on a TI980 minicomputer, which could test about 10^8 nodes per day.

Partial backtracking results in an exponential improvement in Knuth's algorithm for estimating tree size. The effect is particularly important for trees with a lot of dead-end branches (b near $\frac{1}{2}$ in the analysis) and for tall trees. The analysis suggests that good results can be obtained by choosing the number of branches to investigate so that each level, from the root down to the levels with the bulk of the nodes, is examined at about the same frequency. Further improvement is obtained by looking at all branches near the root, although one is usually limited by the number of nodes one can afford to look at.

Appendix. Strassen [5] discovered that matrix multiplication can be done in less than $O(n^3)$ operations. He showed that 2×2 matrix multiplication can be done with 7 multiplications. Finding a method of multiplying N by N matrices with L multiplications is equivalent to finding a solution to the non-linear equations

$$\sum_{1\leqq t\leqq L} A_{ij1t}A_{kl2t}A_{mn3t} = \delta_{jk}\delta_{lm}\delta_{ni} \quad \text{for } 1\leqq i, j, k, l, m, n\leqq N.$$

This is given by Laderman [6] and Brent [7]. The backtrack program which was used as an example for this paper attempted to solve these equations (modulo 2) by guessing values for the A_{ijvt}, proceeding in lexicographical order on the index $ijvt$.

The backtrack program uses two tests at each node. The first test is that each equation must be true. The second test relates to the symmetry of the equations under a permutation of the t index. This symmetry is removed by requiring that the vector

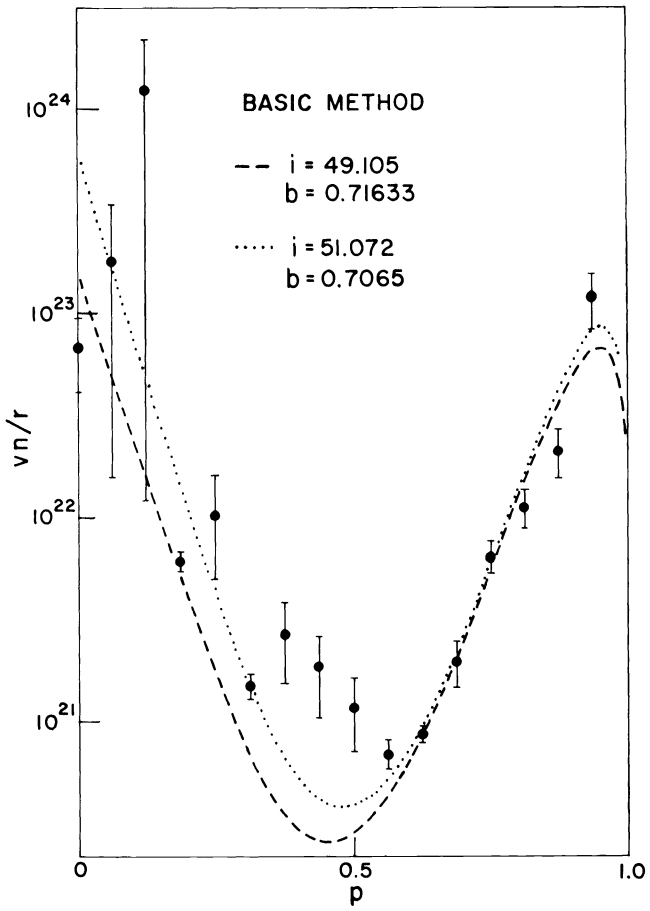


FIG. 3. *The efficiency in nodes per run times variance of estimate of the basic partial backtrack algorithm as a function of the probability, p, that both branches of the tree are followed. The error bars indicate one standard deviation. A least-squares fit of the theory for T_i trees is shown. Also, for comparison, the dotted curve is for the parameters that best fit the data in Fig.* 4.

$(A_{000t}, \cdots, A_{NN3t})$, where the first three indices increase in lexicographical order, is lexicographically greater than or equal to $(A_{000t'}, \cdots, A_{NN3t'})$ when $t > t'$.

Two additional tests are done whenever $t = L$. For the first test consider the vector $(A_{ijv1}A_{jlv'1}, \cdots, A_{ijvL}A_{jlv'L})$ where $v' = (v \bmod 3)+1$. Each set of such vectors formed by varying i and l must be linearly independent. For the second test consider any matrix $\{B_{ij}\}$ and form a vector $(\sum_{i,j} B_{ij}A_{ijv1}, \cdots, \sum_{i,j} B_{ij}A_{ijvL})$. This vector must have at least N times the rank of $\{B_{ij}\}$ nonzero components.

There are a number of additional tests that would be useful, but which the author did not include in this particular backtrack program. These include using the symmetries of Hopcroft, Kerr, and Musinski [8], [9] and using the techniques of Brockett and Dobkin [10]. Unfortunately the author and his coworkers have not yet found a set of tests that will result in a quick solution to the problem of how to best multiply 3×3 matrices.

Acknowledgment. The author wishes to thank Dr. Cynthia A. Brown and David Seaman, who developed programs which led to this work.

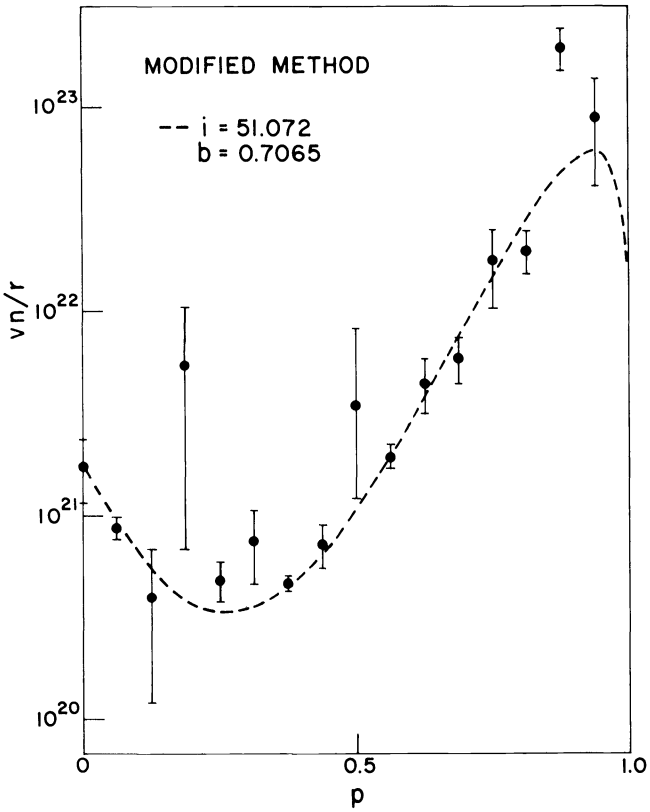


FIG. 4. *The efficiency vs. p for the modified partial backtrack algorithm along with a least-square fit to the theory of T_i trees.*

REFERENCES

[1] D. E. KNUTH, *Estimating the efficiency of backtrack programs*, Math. Comput., 29 (1975), pp. 121–236.

[2] J. R. BITNER AND E. M. REINGOLD, *Backtrack programming techniques*, Comm. ACM, 18 (1975), pp. 651–655.

[3] C. T. FAN, M. E. MULLER AND IVAN REZUCHA, *Development of sampling plans by using sequential (item by item) selecting techniques and digital computers*, J. Amer. Statist. Assoc., 57 (1962), pp. 387–402.

[4] T. G. JONES, *A note on sampling a tape-file*, Comm. ACM, 5 (1962), p. 343.

[5] VOLKER STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[6] JULIAN D. LADERMAN, *A Noncommutative algorithm for multiplying* 3×3 *matrices using* 23 *multiplications*, Bull. Amer. Math. Soc., 82 (1976), pp. 126–128.†

[7] R. BRENT, *Algorithms for matrix multiplication*, Technical Report STAN-CS-70-157, Dept. of Computer Science, Stanford Univ., 1970.

[8] J. E. HOPCROFT AND L. R. KERR, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–36.

[9] J. HOPCROFT AND J. MUSINSKI, *Quality applied to the complexity of matrix multiplication and other bilinear forms*, SIAM J. Comput., 2 (1973), pp. 159–173.

[10] ROGER W. BROCKETT AND DAVID DOBKIN, *On the optimal evaluation of a set of bilinear forms*, Proceedings of Fifth Annual ACM Symposium on Theory of Computing 1973, pp. 88–95. Major revision made in 1976.

† The following notational change has been used $A_{ji1t} \equiv z_{ikt}$, $A_{kl2t} \equiv x_{ght}$, and $A_{mn3t} \equiv Y_{rst}$.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0009 $01.00/0

RANKING AND LISTING ALGORITHMS FOR k-ARY TREES*

ANTHONY E. TROJANOWSKI†

Abstract. The problem of ranking a finite set X may be defined as follows: if $|X| = N$, define a linear order on X and find the order isomorphism $\varphi: X \to \{0, 1, \cdots, N-1\}$, and its inverse φ^{-1}. In this paper, X is the set of k-ary trees on n vertices, $k \geqq 2$, $n \geqq 0$; the linear order is the lexicographic order on a set of permutations used to represent the trees. The representation of k-ary trees by permutations leads to efficient computation of φ and φ^{-1}. One result of this investigation is a generalization of binomial coefficients. The problem of listing all k-ary trees on n vertices is also addressed; an algorithm which, excluding input-output, is linear in the number of such trees is presented.

Key words. algorithm, binary trees, binomial coefficient, linear order, k-ary tree, permutation, ranking function, recursion

1. Introduction. The main problem of classical combinatorics may be described as follows: given a finite set X, evaluate $N = |X|$. Evaluation of N implicitly defines $N!$ set isomorphisms

$$\varphi: X \to \{0, 1, \cdots, N-1\}.$$

Any choice of such a φ induces a total order $<_\varphi$ on X,

$$x <_\varphi y \text{ iff } \varphi(x) < \varphi(y).$$

φ is thus an order isomorphism between the ordered sets $(X, <_\varphi)$ and $\{0, 1, \cdots, N-1\}$ with the usual ordering.

With X and N as as above, a somewhat different problem may be defined: impose a total order $<_X$ on X and construct an order isomorphism

$$\varphi: (X, <_X) \to \{0, 1, \cdots, N-1\}.$$

Such a function is a ranking function on X with respect to $<_X$. Clearly φ is unique, and also, for $x \in X$

$$\varphi(x) = |\{y \in X : y <_X x\}|.$$

A converse problem may be defined: given X, $<_X$, N as above, compute φ^{-1}.

In this paper, X will be the set of k-ary trees on n vertices, $k \geqq 2$, $n \geqq 0$.

DEFINITION 1.1 [*k-ary tree*]. Let k be an integer, $k \geqq 2$.

(i) ϕ, the empty tree, is a k-ary tree with no root and with vertex set $V(\phi) = \phi$;

(ii) if $T_1, \cdots T_k$ are k-ary trees, then structure produced by joining a new root r to the root of each nonempty T_j, $1 \leqq j \leqq k$, is a k-ary tree with root r and vertex set $\{r\} \cup V(T_1) \cup \cdots \cup V(T_k)$;

(iii) there are no other k-ary trees.

DEFINITION 1.2 [*subtree rooted at v*]. Let T be a nonempty k-ary tree with $T_1, \cdots, T_k$ as in Definition 1.1 (ii). Let $v \in V(t)$. (i) If $v = r$, the subtree of T rooted at v is T; (ii) Otherwise, $v \in V(T_j)$ for some j, $1 \leqq j \leqq k$, and the subtree of T rooted at v is the subtree of T_j rooted at v.

DEFINITION 1.3 [*parent, child, etc.*]. Let T be a nonempty k-ary tree, $v \in V(T)$, T_v the subtree of T rooted at v, with $T_{v1}, \ldots T_{vk}$ as in Definition 1.1 (ii). The root of

* Received by the editors June 3, 1977, and in revised form February 10, 1978. This research was supported in part by the National Science foundation under Grant Numbers DCR 74-02774 and MCS 73-03408; this research was completed while the author was a visitor at the Department of Computer Science, University of Illinois, Urbana, IL 61801.

† Department of Mathematics, Illinois State University, Normal, Illinois 61761.

each nonempty T_{vj} is child of v, and v is the parent of such a root. A vertex $w \in T_v$ is a descendant of v, and v is an ancestor of w. If $w \neq v$, the descendant and ancestor relations are proper. A vertex with no proper descendants is a leaf. A vertex which is not a leaf is internal.

DEFINITION 1.4 [*isomorphism*]. Let T_1, T_2 be k-ary trees.

(i) If $T_1 = T_2 = \phi$, T_1 and T_2 are isomorphic.

(ii) If T_1 and T_2 are nonempty and $T_{11}, \cdots, T_{1k}, T_{21}, \cdots T_{2k}$ are as in Definition 1.1 (ii), T_1 and T_2 are isomorphic iff T_{1j} and T_{2j} are isomorphic, $1 \leqq j \leqq k$.

DEFINITION 1.5 [*full k-ary tree*]. A k-ary tree T is a full k-ary tree iff

(i) T is nonempty; and

(ii) each internal vertex of T has exactly k children.

There is a relationship between k-ary trees and full k-ary trees given by the following lemma.

LEMMA 1.6. *The set of k-ary trees and the set of full k-ary trees are in* 1-1 *correspondence.*

Proof. The correspondence is defined as follows:

(i) ϕ corresponds to the full k-ary tree with one vertex;

(ii) A nonempty k-ary tree T corresponds to the full k-ary tree T' obtained from T by adding leaves to T so that every vertex of T has exactly k children in T'.

It is clear that this correspondence is 1-1. Q.E.D.

This correspondence is an obvious extension of the special case $k = 2$ which is presented on p. 559 of [6].

DEFINITION 1.7 [*m-inorder labeling*]. Let T be a nonempty k-ary tree with root r; for $v \in V(T)$, let T_v be the subtree of T rooted at v and let $T_{v1}, \cdots T_{vk}$ be as in Definition 1.1 (ii). If $T_{vj} \neq \phi$, let r_j be the root of T_{vj}. Let $0 \leqq m \leqq k$. The following algorithm labels T in m-inorder.

```
begin
  procedure inorder (v, m);
  begin
    for j: = 1 until m do
    if T_vj ≠ φ then inorder (r_j, m);
    label [v]: = i: = i + 1;
    for j: = m + 1 until k do
    if T_vj = φ then inorder (r_j, m);
  end inorder (v, m);
  i: = 0;
  inorder (r, m);
end.
```

A search which visits the vertices of T in the same order in which they are numbered by an m-inorder labeling is an m-inorder search.

The definition of m-inorder labeling may be extended to arbitrary ordered trees, but this extension is not necessary in this paper and so the present definition is framed in terms of k-ary trees.

The cases $m = 0$ and $m = k$ are the usual pre-order and post-order labelings of T respectively; the case $k = 2$ and $m = 1$ is the usual in-order labeling of a binary tree. See [1] for standard definitions of these labelings.

Finally, to formalize the remarks at the beginning of this paper, the next definition is made.

DEFINITION 1.8 [*ranking function*]. Let $(X, <)$ be a finite, nonempty ordered set with $|X| = N$. The unique order isomorphism φ: $(X, <) \to \{0, 1, \cdots, N-1\}$ is the ranking function for X with respect to the total order $<$.

2. Lexicographic ranking of binary trees. Let $\mathbf{B}(n)$ be the set of binary trees on n vertices; let $B(n) = |\mathbf{B}(n)|$.

DEFINITION 2.1 [*permutation*]. Let n be a positive integer. A permutation σ of $1, 2, \cdots, n$ is a finite sequence $s_1\, s_2 \cdots s_n$ satisfying

$$\{s_1, \cdots, s_n\} = \{1, \cdots, n\}.$$

The set of all permutations of $1, 2, \cdots, n$ is denoted by $\mathbf{S}_n$.

There is much more structure which can be associated with $\mathbf{S}_n$ but it will not be necessary for the present development.

DEFINITION 2.2 [$\sigma(j_1, \cdots, j_m)$]. Let n and m be positive integers with $n \geqq m$. Let $\{j_1, \cdots, j_m\} \subseteq \{1, \cdots, n\}$; let $\sigma \in \mathbf{S}_n$. $\sigma(j_1, \cdots, j_m)$ is the finite sequence obtained by deleting $j_1, \cdots, j_m$ from σ.

DEFINITION 2.3 [$\mathbf{P}(n)$]. Let n be a positive integer. $\mathbf{P}(n) \subseteq \mathbf{S}_n$ is defined recursively as follows:

(i) $\mathbf{P}(1) = \mathbf{S}_1$;

(ii) if $n > 1$, $\sigma \in \mathbf{P}(n)$ iff $\sigma(n) \in \mathbf{P}(n-1)$ and $s_j = n$, $s_k = n-1$ implies $j \geq k-1$.

The following theorem gives the relationship between $\mathbf{P}(n)$ and $\mathbf{B}(n)$.

THEOREM 2.4. *There is a* 1-1 *correspondence between* $\mathbf{P}(n)$ *and* $\mathbf{B}(n)$, $n \geqq 1$.

Proof. Let $T \in \mathbf{B}(n)$; generate a permutation $\sigma \in \mathbf{S}_n$ according to the following scheme: label the vertices of T in pre-order, and read off the labels in in-order. It must be shown that $\sigma \in \mathbf{P}(n)$. The proof is by induction on n.

For $n = 1, 2$, the result is immediate. Let $n > 2$, $T_1 \in \mathbf{B}(n_1)$, $T_2 \in \mathbf{B}(n_2)$ as in Definition 1.1, and assume that the result is true for $1, 2, \cdots, n-1$.

Let $\sigma \in \mathbf{P}(n_1)$ be the permutation generated from T_1, and $\tau \in \mathbf{P}(n_2)$ be the permutation generated from T_2; if $n_1 = 0$ or $n_2 = 0$, the corresponding permutation is defined to be the empty sequence. Let $\sigma = s_1\, s_2 \cdots s_{n_1}$, $\tau = t_1 t_2 \cdots t_{n_2}$. Let $s'_j = s_j + 1$; let $t'_j = t_j + n_1 + 1$. Then it is clear that the permutation ρ generated from T is given by

$$\rho = s'_1 s'_2 \cdots s'_{n_1}\, 1\, t'_1 t'_2 \cdots t'_{n_2}. \tag{2.5}$$

It follows that $s'_i = n$, $t'_j = n-1$ is impossible, and since $\sigma \in \mathbf{P}(n_1)$, $\tau \in \mathbf{P}(n_2)$, the second condition of Definition 2.3 (ii) must be satisfied. Let $\rho = r_1 r_2 \cdots r_n$, with $r_j = n$. Let $v \in V(T)$ be the vertex labeled n. If $T' \in \mathbf{B}(n-1)$ is the binary tree obtained by deleting v from T, the permutation generated from T' is $\rho(n)$, and by the induction hypothesis, $\rho(n) \in \mathbf{P}(n-1)$; hence the first condition of Definition 2.3 (ii) is satisfied. Thus the proof that $\sigma \in \mathbf{P}(n)$ is complete.

So far, the existence of a 1-1 map from $\mathbf{B}(n)$ into $\mathbf{P}(n)$ has been proven; it remains to show that this map is onto. This will be accomplished by constructing an inverse to the map just defined.

Let $a_1\, a_2 \cdots a_n$ be a finite sequence of distinct positive integers. An unique binary tree $T(a_1\, a_2 \cdots a_n) \in \mathbf{B}(n)$ may be constructed as follows:

(I) If $n = 1$, $T(a_1)$ is the binary tree with one vertex;

(II) if $n > 1$, let $a_j = \min\{a_1, \cdots, a_n\}$. Then $T(a_1, a_2 \cdots a_m)$ is the binary tree formed by joining $T(a_1\, a_2 \cdots a_{j-1})$ and $T(a_{j+1}\, a_{j+2} \cdots a_n)$ to a root r as in Definition 1.1 (ii); if either sequence is empty, so is the corresponding tree.

The claim is that $\rho \mapsto T(\rho)$ is the inverse of the map previously constructed; the proof is by induction on n.

For $n = 1, 2$, the result is obvious. Suppose that $n > 2$, and that the result holds for $1, 2, \cdots, n-1$. Let $\sigma \in \mathbf{P}(n_1)$; $\tau \in \mathbf{P}(n_2)$, s_i, t_j be as before. Referring to Equation 2.5, it is easy to see that $T(s'_1 \cdots s'_{n_1}) = T(\sigma)$ and $T(t'_1 \cdots t'_{n_2}) = T(\tau)$; thus $T(\rho)$ is the binary tree formed by joining $T(\sigma)$ and $T(\tau)$ to a new root; but by the induction hypothesis, $T(\sigma) = T_1$, $T(\tau) = T_2$. Hence $T(\rho) = T$. Q.E.D.

This same result, using a different (although equivalent) definition of $\mathbf{P}(n)$ is derived in Exercises 2.2.1–5 and 2.3.1–6 of [6]. This derivation goes by way of stacks, while the above theorem obtains the result directly.

Example 2.6. For $n = 1, 2$, $\mathbf{P}(n) = \mathbf{S}_n$. For $n = 3$, $\mathbf{P}(n) = \mathbf{S}_n - \{312\}$. $4321 \in \mathbf{P}(4)$ while $4312 \notin \mathbf{P}(4)$. The permutation generated from the tree in Fig. 2.7 (i) is 32415; the permutation generated from the tree in Fig. 2.7 (ii) is 231546.

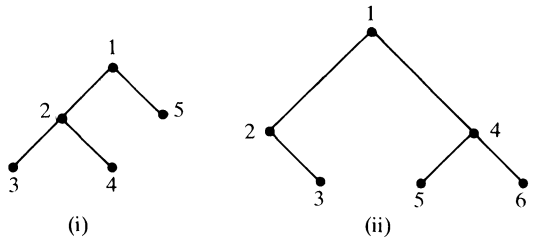


FIG. 2.7

DEFINITION 2.8 [*lexicographic ordering on* $\mathbf{B}(n)$]. Let n be a positive integer; let T_1, $T_2 \in \mathbf{B}(n)$; T_1 is lexicographically less than T_2 iff the permutation $\sigma_1 \in \mathbf{P}(n)$ corresponding to T_1 is lexicographically less than $\sigma_2 \in \mathbf{P}(n)$ corresponding to T_2.

The remainder of this section will be concerned with ranking and unranking algorithms on $\mathbf{P}(n)$ with respect to the lexicographic ordering.

DEFINITION 2.9 [$\mathbf{P}(n, m)$, $P(n, m)$]. Let n be a positive, m a nonnegative integer; define

$$\mathbf{P}(n, m) = \{\sigma \in \mathbf{P}(n) : s_m = n\};$$

and

$$P(n, m) = |\mathbf{P}(n, m)|.$$

Clearly $P(n, m) = 0$ unless $1 \leqq m \leqq n$.

From this point on, the notion of concatenation of lists or strings will be important; the symbol $\|$ will be used to denote the operation of concatenation of lists or strings.

DEFINITION 2.10 [*Direct insertion order on* $\mathbf{S}_n$, $\mathbf{P}(n)$]. Let n be a positive integer. The direct insertion order is defined on $\mathbf{S}_n$ as follows:

(i) $\mathbf{S}_1$ is in direct insertion order;

(ii) let $L_{n-1} = (\sigma_1, \sigma_2, \cdots, \sigma_{(n-1)!})$ be a listing of $\mathbf{S}_{n-1}$ in direct insertion order. For $1 \leqq i \leqq (n-1)!$, create a list $L_{n,i}$ by inserting the character n into each possible position in σ_i, starting at the right and working leftwards. The listing of $\mathbf{S}_n$ in direct insertion order is

$$L_{n,1} \| L_{n,2} \| \cdots \| L_{n,(n-1)!}.$$

The definition of direct insertion order on $\mathbf{P}(n)$ is similar except that in part (ii), if $\sigma_i \in \mathbf{P}(n-1, m)$, the insertion process halts with the insertion of n to the immediate left of s_m.

	$n=1$	2	3	4
$\mathbf{S}_n$	1	12	123	1234
		21	132	1243
			312	1423
			213	4123
			231	1324
			321	1342
				1432
				4132
				3124
				3142
				3412
				4312
				.
				.
				.
				etc.
$\mathbf{P}(n)$	1	12	123	1234
		21	132	1243
			213	1324
			231	1342
			321	1432
				2134
				2143
				2314
				2341
				2431
				3214
				3241
				3421
				4321

FIG. 2.11. *Generation of* $\mathbf{S}_n$ *and* $\mathbf{P}(n)$ *by direct insertion.*

Figure 2.11 depicts the generation of $\mathbf{S}_n$ and $\mathbf{P}(n)$ by direct insertion, for $n = 1, 2, 3, 4$.

Note that while lexicographic order and direct insertion order differ on $\mathbf{S}_n$ for $n = 3, 4$, they coincide on $\mathbf{P}(n)$ for $n = 1, 2, 3, 4$. This is not accidental.

LEMMA 2.12. *The lexicographic and direct insertion orders on* $\mathbf{P}(n)$ *coincide.*

Proof. Induction on n; from Fig. 2.11 the result is true for $n = 1, 2, 3, 4$.

Suppose the result is true for $\mathbf{P}(n-1)$; let $L_{n,i}$ be the list generated from σ_i, $1 \leqq i \leqq B(n-1)$. Clearly each $L_{n,i}$ is in lexicographic order; so it need only be shown that the last element of $L_{n,i}$ is lexicographically less than the first element $L_{n,i+1}$, $1 \leqq i \leqq B(n-1)-1$.

Let $\sigma_i = s_{i,1}\, s_{i,2} \cdots s_{i,n-1}$, with $\sigma_i \in \mathbf{P}(n-1, m)$; and let $\sigma_{i+1} = s_{i+1,1}\, s_{i+1,2} \cdots s_{i+1,n-1}$. Then the last element of $L_{n,i}$ is $\rho = s_{i,1} \cdots s_{i,m-1}(n)\, s_{i,m} \cdots s_{i,n-1}$; and the first element of $L_{n,i+1}$ is $\tau = s_{i+1,1} \cdots s_{i+1,n-1}(n)$.

First note that since it is assumed that $i \leqq B(n-1)-1$, it must be that $2 \leqq m \leqq n-1$, for the only element of $P(n-1, 1)$ is $(n-1)\,(n-2) \cdots (2)\,(1) = \sigma_{B(n-1)}$.

Suppose that $\tau < \rho$ lexicographically in $\mathbf{P}(n)$. Since $\sigma_i < \sigma_{i+1}$ lexicographically in $\mathbf{P}(n-1)$, it follows that $s_{i,j} = s_{i+1,j}$ for $1 \leqq j \leqq m-1$, and $s_{i,m} \leqq s_{i+1,m} < n$. But since ρ is the lexicographically largest element of $L_{n,i}$, it follows that $s_{i,m} = n-1$; hence $s_{i+1,m} = n-1$. But σ_i and σ_{i+1} are successive elements in the list of $\mathbf{S}_{n-1}$ in direct insertion order; hence $s_{i,m} = s_{i+1,m} = n-1$ is impossible. Q.E.D.

Before proceeding with the development of the ranking and unranking algorithms, it is worthwhile to evaluate $P(n, m)$ for $1 \leqq m \leqq n$. Considering the construction of $\mathbf{P}(n)$ form $\mathbf{P}(n-1)$ by direct insertion, it is easy to see that for $m \geqq 1$, $n \geqq 2$,

$$P(n, m) = \sum_{\mu=1}^{m} P(n-1, \mu). \tag{2.13}$$

As was remarked in the proof of Lemma 2.12,

$$P(n, 1) = 1, \quad n \geqq 1. \tag{2.14}$$

Equations (2.13) and (2.14) may be used to obtain the following recursion with boundary conditions:

$$\begin{aligned} &P(n, m) = P(n, m-1) + P(n-1, m), \qquad n \geqq 2, \ m \geqq 1; \\ &P(n, 1) = 1, \ n \geqq 1; \qquad P(n, m) = 0, \qquad n < m. \end{aligned} \tag{2.15}$$

LEMMA 2.16. *For* $2 \leqq m \leqq n$,

$$P(n, m) = \frac{(n-m+1)}{(m-1)!} \prod_{\mu=1}^{m-2} (n+\mu), \tag{2.17}$$

where the product is defined to be 1 *for* $m = 2$.

Proof. The proof is a routine calculation. Q.E.D.

For $\sigma \in \mathbf{P}(n)$, denote by *lexrank* (σ) the rank of σ in the lexicographic ordering of $\mathbf{P}(n)$. It will be seen in the next lemma that the $P(i, j)$ are intimately involved in the computation of *lexrank* (σ).

LEMMA 2.18. *Let* $\sigma = s_1 s_2 \cdots s_n \in \mathbf{P}(n)$, *let* $0 \leqq j < i < n$, *and suppose that* σ *satisfies*

$$s_l = l, \ n-j+1 \leqq l \leqq n;$$

$$s_{n-i} = n-j.$$

Let $\tau = s_1 s_2 \cdots s_{n-i-1} s_{n-i+1} \cdots s_{n-j}(n-j)(n-j+1) \cdots (n)$. *Then lexrank* (σ) − *lexrank* $(\tau) = P(i+1, j+2)$.

Proof. By induction on i, j. If $j = 0$, $s_n \neq n$, and clearly *lexrank* (σ) − *lexrank* $(\tau) = i = P(i+1, 2)$.

Suppose that $j > 0$ and $i = j+1$. Then

$$\sigma = s_1 \cdots s_{n-j-2}(n-j)s_{n-j}(n-j+1) \cdots (n);$$

$$\tau = s_1 \cdots s_{n-j-2}s_{n-j}(n-j)(n-j+1) \cdots (n).$$

Let

$$\sigma' = s_1 \cdots s_{n-j-2}s_{n-j}(n)(n-1) \cdots (n-j+1)(n-j).$$

Considering the insertion process by which σ' and σ are constructed from $s_1 \cdots s_{n-j-2}s_{n-j} \in \mathbf{P}(n-j-1)$, it is seen that

$$\textit{lexrank } (\sigma) = \textit{lexrank } (\sigma') + 1.$$

Thus

$$\begin{aligned} \textit{lexrank } (\sigma) - \textit{lexrank } (\tau) = 1 &+ \textit{lexrank } (\sigma') \\ &- \textit{lexrank } (\tau); \end{aligned}$$

but $\{\rho : \tau \leqq \rho < \sigma \text{ lexicographically}\}$ is order isomorphic to $\mathbf{P}(j+1)$. Hence

$$\begin{aligned} \textit{lexrank}\ (\sigma) - \textit{lexrank}\ (\tau) &= 1 + P(j+1) - 1 \\ &= P(j+1) \\ &= P(j+2, j+2). \end{aligned}$$

Now let $j > 0$, $i > j+1$, and suppose inductively that the result is true for $i-1$ and j; and that the result is true for $j-1$ and all i' satisfying $j-1 < i' < n$. Let

$$\sigma = s_1 s_2 \cdots s_{n-j-1}(n-j)s_{n-i+1} \cdots s_{n-j}(n-j+1) \cdots (n);$$

then

$$\tau = s_1 s_2 \cdots s_{n-i-1}s_{n-i+1} \cdots s_{n-j}(n-j)(n-j+1) \cdots (n).$$

Let

$$\sigma' = s_1 s_2 \cdots s_{n-i-1}s_{n-i+1}(n)(n-1) \cdots (n-j)s_{n-i+2} \cdots s_{n-j};$$

$$\sigma'' = s_1 s_2 \cdots s_{n-i-1}s_{n-i+1}(n-j)s_{n-i+1} \cdots s_{n-j}(n-j+1) \cdots (n).$$

As before,

$$\textit{lexrank}\ (\sigma) - \textit{lexrank}\ (\sigma') = 1;$$

thus

$$\begin{aligned} \textit{lexrank}\ (\sigma) - \textit{lexrank}\ (\tau) &= 1 + \textit{lexrank}\ (\sigma') - \textit{lexrank}\ (\tau) \\ &= 1 + \textit{lexrank}\ (\sigma') - \textit{lexrank}\ (\sigma'') \\ &\quad + \textit{lexrank}\ (\sigma'') - \textit{lexrank}\ (\tau). \end{aligned}$$

But

$$\textit{lexrank}\ (\sigma'') - \textit{lexrank}\ (\tau) = P(i, j+2)$$

by the induction hypothesis on i; and

$$\textit{lexrank}\ (\sigma') - \textit{lexrank}\ (\sigma'') = \sum_{\nu=0}^{j-1} P(i, \nu+2)$$

by the induction hypothesis on j. Hence

$$\begin{aligned} \textit{lexrank}\ (\sigma) - \textit{lexrank}\ (\tau) &= 1 + \sum_{\nu=0}^{j-1} P(i, \nu+2) + P(i, j+2) \\ &= \sum_{\nu=-1}^{j} P(i, \nu+2) \\ &= P(i+1, j+2) \end{aligned}$$

by Equation 2.13. Q.E.D.

The result of Lemma 2.18 is exploited by the following algorithm, which computes the lexicographic rank of $\sigma \in \mathbf{P}(n)$. In this algorithm σ is represented by a linked list, although the representation and linked list manipulations are suppressed in the description presented here; the array $p[1:n, 1:n]$ is used to store the values of $P(i, j)$; $p[i, j] = P(i, j)$. Discussion of the method of computing the $p[i, j]$ is deferred until after the description of the algorithm.

ALGORITHM 2.19. Input: $\sigma \in \mathbf{P}(n)$, represented by a linked list. Output: The lexicographic rank of σ in $\mathbf{P}(n)$.

```
begin
  procedure lexrank (σ, n, l);
  begin
    temp: = 0;
    for j: = l until n − 2 do
    if s_{n−j}¬ = n − j then
    for i: = 1 until n − 1 do
    if s_{n−i} = n − j then
    begin
      temp: = p[i + 1, j + 2]
              + lexrank (σ(n − j, ···, n)‖(n − j)‖ ··· ‖(n), n, j + 1);
      i: = n;
      j: = n − 1;
    end;
    lexrank: = temp;
  end lexrank (σ, n, l);
  compute p[1 : n, 1 : n];
  lexrank (σ, n, 0);
end.
```

THEOREM 2.20. *Algorithm* 2.19 *is* $O(n^2)$ *time-bounded; the algorithm can be implemented so that it is* $O(n)$ *space-bounded.*

Proof. The procedure call *lexrank* (σ, n, l) requires $O(n-1)$ steps to execute the loops on i and j in the worst case; so the worst-case time-complexity for *lexrank* is given by the recursion

$$t(\textit{lexrank}\ (\sigma, n, 0)) \leqq 0(n) + t(\textit{lexrank}\ (\sigma(n)\|(n), n, 1))$$

since computation of $\sigma(n-j, \cdots, n)\|(n-j)\| \cdots \| \cdots \|(n)$ is constant time by linked-list operations. But the time required by *lexrank* $(\sigma(n)\|(n), n, 1)$ is the same as the time required by *lexrank* $(\sigma(n), n-1, 0)$. Hence

$$t(\textit{lexrank}\ (\sigma, n, 0)) = O(n^2).$$

Computation of $p[1:n, 1:n]$ by Equations 2.15 requires $O(n^2)$ time. It appears at first storing the $P(i, j)$ in the array $p[1:n, 1:n]$ (as opposed to direct computation from Equation 2.17, which also leads to an $O(n^2)$ time-bound) requires $O(n^2)$ storage. However, any particular column of $p[1: n, 1:n]$ is accessed at most once during the execution of Algorithm 2.20; and those columns which are accessed in ascending order on j. Furthermore, if the jth column of $p[1:n, 1:n]$ is known, the $j+1$st may be computed using Equations 2.17. Hence no more than two complete columns of $p[1:n, 1:n]$ need be kept in storage at any time; hence the space bound. Q.E.D.

The essential steps of Algorithm 2.19 are illustrated in the following example.

EXAMPLE 2.21. Compute the lexicographic rank of $\sigma = 35421$ in $\mathbf{P}(5)$. According to Lemma 2.19,

$$\begin{aligned} \textit{lexrank}\ (35421) &= \textit{lexrank}\ (34215) + P(4,2) \\ &= \textit{lexrank}\ (32145) + P(4,3) + P(4,2) \\ &= \textit{lexrank}\ (21345) + P(5,4) + P(4,3) + P(4,2) \\ &= \textit{lexrank}\ (12345) + P(5,5) + P(5,4) + P(4,3) + P(4,2). \end{aligned}$$

But $\textit{lexrank}\ (12345) = 0$; and evaluation of the appropriate $P(i, j)$ yields

$$\begin{aligned} \textit{lexrank}\ (35421) &= 14 + 14 + 5 + 3 \\ &= 36. \end{aligned}$$

Lemma 2.19 also leads to an unranking algorithm.

ALGORITHM 2.22. Input: integers $n \geqq 1$, $0 \leqq r \leqq B(n) - 1$. Output: the rank r element of $\mathbf{P}(n)$, represented as a linked list.

```
begin
  procedure lexrecover (r, n, j);
  begin
    if j ≧ 0 then
    begin
      find largest i such that P[i+1, j+2] ≦ r;
      σ: = s_1 ··· s_{n-i-1}(n-j)s_{n-i} ··· s_{n-j-1};
      lexrecover [r-p[i+1, j+2], n, j-1);
    end;
  end lexrecover (r, n, j);
  compute p[1:n, 1:n];
  σ: = 1;
  lexrecover (r, n, n-2);
end.
```

In this algorithm, as in Algorithm 2.19, the linked-list operations have been suppressed.

THEOREM 2.23. *Algorithm 2.22 is $O(n^2)$ time-bounded and $O(n)$ space bounded.*

Proof. In the procedure call *lexrecover* (r, n, j), the time required to find the appropriate $P(i, j)$ is at most $O(n - j)$; hence the time required for *lexrecover* (r, n, j) is given by

$$\begin{aligned} t(\textit{lexrecover}\ (r, n, j)) \leqq\ & O(n-j) \\ & + t(\textit{lexrecover}\ (r - P[i+1, j+2], n, j-1). \end{aligned}$$

(It follows that the call *lexrecover* $(r, n, n-2)$ requires at most $O(n^2)$ time. On the other hand, it can be seen that *lexrecover* $(B(n)-1, n, n-2)$ requires $O(n^2)$ time.

Computation of $P[1:n, 1:n]$ requires $O(n^2)$ time using Equations 2.15.

The $O(n)$ space requirement is demonstrated in a fashion similar to the proof of Theorem 2.20 in this case, the columns of $P[1:n, 1:n]$ are accessed in decreasing order of j; if the jth column is known, the $j-1$st can be computed using Equations 2.15. Q.E.D.

Example 2.24. Given $n=5$, $r=36$, compute the rank r element of $\mathbf{P}(n)$ in lexicographic order. At the outset, set $\sigma=1$. The largest i such that $P(i+1, 5) \leqq 36$ is 4: $P(5, 5)=14$. Hence 2 is inserted immediately to the left of $s_{5-4}=s_1$ in σ i.e. $\sigma=21$; r is decremented to 22. The largest i such that $P(i+1, 4) \leqq 22$ is 4: $P(5, 4)=14$. Hence 3 is inserted into the first position in σ, i.e. $\sigma=321$, and r is decremented to 8. At the next step, $j=1$ and i is found to be 3; so σ becomes 3421 and r is decremented to 3; at the last step, $j=0$, and i is found to be 3; so σ becomes 35421 and r is decremented to 0. The algorithm halts at this point.

In the next section, the ranking and unranking of k-ary trees will be treated in a similar fashion.

3. Lexicographic ranking of full k-ary trees. Recall from Lemma 1.6 the correspondence between k-ary trees and full k-ary trees; from this correspondence, it follows that to rank k-ary trees on n vertices, it suffices to rank full k-ary trees on $kn+1$ vertices. The general strategy will be to relate full k-ary trees on $kn+1$ vertices to a subset of $\mathbf{B}(nk+1)$, which subset will be ranked in a manner similar to the lexicographic ranking of $\mathbf{B}(n)$ in § 2.

Using the standard definition of (ordered) tree and forest (see § 2.3.2 of [6]), the following definition is made.

DEFINITION 3.1 [*operations on ordered forests*]. Let $F=(T_1, \cdots, T_n)$ $F'=(T'_1, \cdots, T'_m)$ be ordered forests; then FF' is the ordered forest $(T_1, \cdots, T_n, T'_1, \cdots, T'_m)$; $\hat{F}$ is the ordered forest (T), where T is the tree obtained by joining the root of each T_i to a new root r. In particular, if F is the empty ordered forest, $\hat{F}$ is the ordered forest on one vertex.

Definition 3.1 allows a concise statement of the correspondence between ordered forests on n vertices and elements of $\mathbf{B}(n)$.

DEFINITION 3.2 [*correspondence between binary trees and ordered forests*].

(i) The empty binary tree corresponds to the empty ordered forest;

(ii) If T_1 and T_2 are binary trees corresponding to the ordered forests F_1 and F_2 respectively, the binary tree T which has T_1 and T_2 as its left and right subtrees corresponds to $F_1\hat{F}_2$.

The correspondence in § 2.3.2 of [6] has the binary tree T corresponding to $\hat{F}_1F_2$; Definition 3.2 (ii) is preferable for the present purposes. That the correspondence in Definition 3.2 is 1-1 follows from the fact that the correspondence in [6] is 1-1.

DEFINITION 3.3 [U_k]. Let $k \geqq 2$; $U_k \in \mathbf{B}(k)$ is the unique binary tree on k vertices such that no vertex of U_k has a right child.

DEFINITION 3.4. [$\mathbf{F}(nk+1, k)$]. Let $n \geqq 0$, $k \geqq 2$.

(i) If $n=0$, $\mathbf{F}(1, k)=\mathbf{B}(1)$;

(ii) If $n \geqq 1$, let $n_1+\cdots+n_k=n-1$, let $T_i \in \mathbf{F}(n_ik+1, k)$, $1 \leqq i \leqq k$, and let T_i have root r_i; form a binary tree $T \in \mathbf{B}(nk+1)$ as follows:

(a) the root of T is r; r has no left child, and the right child of r is r_1;

(b) for $1 \leqq i \leqq k$, the left child of r_i is r_{i+1}, if r_{i+1} exists; the right subtree of r_i is the right subtree of r_i in T_i. Then $T \in \mathbf{F}(nk+1, k)$.

(iii) $\mathbf{F}(nk+1, k)$ has no other elements.

Example 3.5. Figure 3.6 depicts the sets $\mathbf{F}(nk+1, k)$ for $k=2, 3$, and $n=0, 1, 2$.

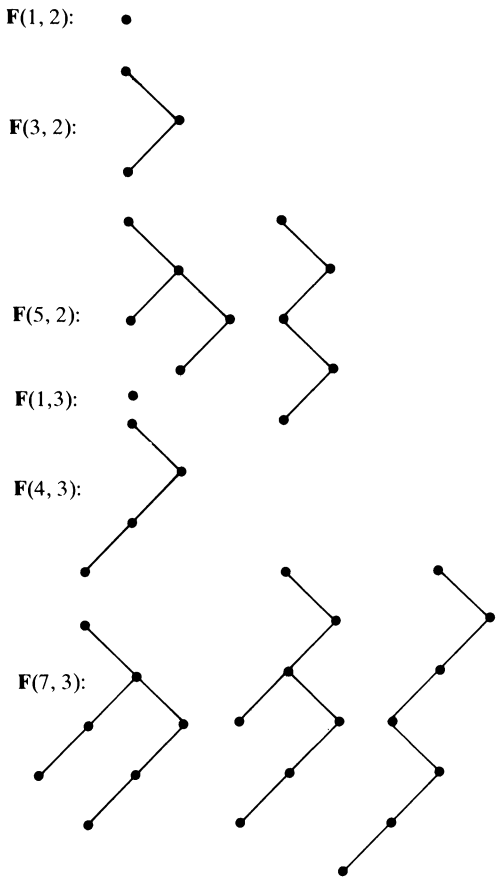


FIG. 3.6. $\mathbf{F}(nk+1, k)$ *for* $k = 2, 3$, $n = 0, 1, 2$.

LEMMA 3.7. *Let* $n \geqq 0$, $k \geqq 2$. *There is a* 1-1 *correspondence between* $\mathbf{F}(nk+1, k)$ *and the set of full* k*-ary trees on* $nk+1$ *vertices.*

Proof. The 1-1 correspondence is the restriction to $\mathbf{F}(nk+1, k)$ of the 1-1 correspondence in Definition 3.2. The proof is by induction on n.

The case $n = 0$ follows immediately from Definition 3.2 (ii) applied to the binary tree on one vertex.

Suppose the result is true for $0, 1, \cdots, n-1$. Let $T \in \mathbf{F}(nk+1, k)$ be formed from $T_i \in \mathbf{F}(n_i k+1, k)$, $1 \leqq i \leqq k$ as in Definition 3.4. Let S_i be the full k-ary tree on $n_i k+1$ vertices corresponding to T_i. It follows from Definition 3.2 that the ordered forest corresponding to T is the tree S obtained by joining the root of each T_i to a new root r; hence S is a full k-ary tree on $nk+1$ vertices. The argument is completely reversible. Q.E.D.

Thus, to rank the set of full k-ary trees on $nk+1$ vertices, it suffices to rank $\mathbf{F}(nk+1, k)$. This will be accomplished by considering the subset of $\mathbf{P}(nk+1)$ generated by $\mathbf{F}(nk+1, k)$ using the method of Theorem 2.4.

DEFINITION 3.8 [$\mathbf{P}(nk+1; k)$]. $\mathbf{P}(nk+1; k) \subseteq \mathbf{S}_{nk+1}$ is the set of permutations generated from $\mathbf{F}(nk+1, k)$ by the method of Theorem 2.4.

By what has already been said, $\mathbf{P}(1; k) = \mathbf{S}_1$; the next result characterizes $\mathbf{P}(nk+1; k)$ for $n \geqq 1$.

LEMMA 3.9. *Let* $n \geqq 1$, $k \geqq 2$, $\sigma \in \mathbf{S}_{nk+1}$. *Then* $\sigma \in \mathbf{P}(nk+1; k)$ *iff*

(i) $\sigma((n-1)k+2, (n-1)k+3, \cdots, nk, nk+1) \in \mathbf{P}((n-1)k+1, k)$;

(ii) $(nk+1)(nk)(nk-1)\cdots((n-1)k+2)$ *is a substring of* σ;

(iii) *if* $s_m = nk+1$ *and* $s_{m'} = (n-1)k+1$, *then* $m > m'$.

Proof. By induction on n; it is easily seen that $\mathbf{P}(k+1; k)$ consists of the single permutation

$$\sigma = (1)(k+1)(k)\cdots(2),$$

which satisfies the conditions; and no other $\tau \in \mathbf{S}_{k+1}$ satisfies these conditions.

Let $n > 1$, and suppose that the result is true for $n-1$. Let $T \in \mathbf{F}(nk+1, k)$, and label the vertices of T in pre-order. It follows from Definition 3.4 that the vertices labeled $(n-1)k+2, \cdots, nk+1$ form a subtree rooted at the vertex w labeled $(n-1)k+2$, and this subtree is U_k. Let v be the parent of w; then *label* $[v] \leqq (n-1)k+1$. Since a descendant of v is labeled $nk+1$, it must be that the vertex labeled $(n-1)k+1$ is a descendant of v; furthermore, this vertex must equal v or lie in left subtree of v, since it can be seen that the right subtree of v is precisely the U_k already mentioned.

If the right subtree of v is deleted, the result is an element of $\mathbf{F}((n-1)k+1, k)$ labeled in pre-order. By the induction hypothesis, the permutation generated from this tree is an element σ of $\mathbf{P}((n-1)k+1; k)$. By the structural properties of T discussed above, it follows the permutation generated by T may be obtained from σ by inserting the sequence $(nk+1)(nk)\cdots((n-1)k+2)$ immediately to the right of *label* $[v]$ in σ; but either *label* $[v] = (n-1)k+1$ or $(n-1)k+1$ lies to the left of *label* $[v]$ in σ; hence the permutation generated by T satisfies (i), (ii), and (iii).

Let $\sigma \in \mathbf{S}_{nk+1}$, $n \geqq 2$, satisfy (i), (ii), (iii); let $T \in \mathbf{B}(nk+1)$ be the binary tree generated from σ as in Theorem 2.4. By the induction hypothesis, the binary tree T' generated from $\sigma((n-1)k+2, \cdots, nk+1)$ is an element of $\mathbf{F}((n-1)k+1, k)$; also, the subtree of T corresponding to $(nk+1)(nk)\cdots((n-1)k+2)$ is U_k; so it remains to be shown that this U_k is the right subtree of some vertex v in T'. Let v be the parent of u, where *label* $[u] = (n-1)k+2$. Suppose $u = lchild\ [v]$; now, *label* $[v] \leqq (n-1)k+1$, and v is a vertex in a subtree U_k of T'; if v is not the leaf of U_k, it is obvious that $u = rchild\ [v]$; hence v is the leaf of the subtree U_k. Hence *label* $[v] = n'k+1$ for some $n' < n$; also, $nk+1$ occurs to the left of $n'k+1$ in σ, from the construction of the inverse map in Theorem 2.4, but by (iii), $n'k+1$ occurs to the left of $(n-1)k+1$; hence $nk+1$ occurs to the left of $(n-1)k+1$, which contradicts the assumption about σ. It follows that $T \in \mathbf{F}(nk+1, k)$. Q.E.D.

Lemma 3.9 leads to a lexicographic ranking scheme for $P(nk+1; k)$; the first step is to obtain a result analogous to Lemma 2.12.

DEFINITION 3.10 [$\mathbf{P}(nk+1, m; k)$]. Let $n \geqq 0$, $k \geqq 2$, $1 \leqq m \leqq nk+1$. $\mathbf{P}(nk+1, m; k) = \{\sigma \in \mathbf{P}(nk+1; k): s_m = nk+1\}$.

DEFINITION 3.11 [*direct insertion order on* $\mathbf{P}(nk+1; k)$]. The direct insertion order on $\mathbf{P}(nk+1; k)$ is defined analogously to the direct insertion order on $\boldsymbol{P}(n)$; i.e. for $\sigma \in \mathbf{P}((n-1)k+1, m; k)$, the string $(nk+1)(nk)\cdots((n-1)k+2)$ is into σ starting at the right and proceeding leftwards until insertion between s_m and s_{m+1}.

LEMMA 3.12. *On* $\mathbf{P}(nk+1; k)$, *the lexicographic order and direct insertion order coincide.*

Proof. By induction on n; the result is trivial for $n = 0$, 1 since $P(1; k) = P(k+1; k) = 1$. Let $n > 1$ and suppose the result holds for $n-1$. As in Lemma 2.12, it suffices to show that for $\sigma < \tau$ lexicographically consecutive in $\mathbf{P}((n-1)k+1; \sigma')$, the lexicographically largest element of $\mathbf{P}(nk+1; k)$ formed from σ by direct insertion, is

lexicographically less than τ', the lexicographically least element of $\mathbf{P}(nk+1; k)$ formed from τ by direct insertion.

Let $\sigma = s_1 \cdots s_{(n-1)k+1}$, $\tau = t_1, \cdots, t_{(n-1k+1}$. By the induction hypothesis and the properties of $\mathbf{P}((n-1)k+1; k)$ there is an $l > 1$ such that

$$s_i = t_i,\ 1 \leqq i \leqq l; \qquad s_{l+1} < t_{l+1}.$$

Suppose that $\tau' < \sigma'$; it follows that $\sigma' \in \mathbf{P}(nk+1, m; k)$ for $m \leqq l+1$. By the definition of direct insertion order on $\mathbf{P}(nk+1; k)$, $s_{m-1} = (n-1)k+1$; hence $t_{m-1} = (n-1)k+1$. But this is impossible since $\mathbf{P}((n-1)k+1; k)$ is in direct insertion order. Q.E.D.

The final step in preparation for the ranking and unranking algorithms on $\mathbf{P}(nk+1; k)$ is the definition of some auxiliary coefficients.

DEFINITION 3.13 [$C(i, j, k)$]. Let $k \geqq 2$, $j \geqq 0$, $i \geqq (j+1)k-1$; let $n \geqq \max(j+2)k, i)$. Let $\sigma = s_1 \cdots s_{nk+1} \in \mathbf{P}(nk+1; k)$ satisfy

$$s_{(n-l-1)k+2} = (n-l)k+1, \qquad 0 \leqq l \leqq j-1;$$

let

$$s_{nk-i+1} = (n-j)k+1.$$

Let $\tau = t_1 \cdots t_{nk+1}$ be obtained from $\sigma((n-j-1)k+2, \cdots, (n-j)k+1)$ by inserting the substring $((n-j)k+1)((n-j)k) \cdots (n-j-1)k+2$ so that $t_{(n-j-1)k+2} = (n-j)k+1$.

Define

$$C(i, j, k) = lexrank\ (\sigma) - lexrank\ (\tau),$$

where *lexrank* is computed with respect to $\mathbf{P}(kn+1; k)$. If $0 \leqq i < (j+1)k-1$, define

$$C(i, j, k) = 0.$$

The definition of $C(i, j, k)$ and the properties derived therefrom in [9] may be summarized in the following recursion:

$$C(i, j, k) = \begin{cases} 0, & 1 \leqq i \leqq j(k+1)-1; \\ i-k+1, & j=0, \quad i \geqq k; \\ C(i-1, j-1, k) + C(i-1, j, k), & \text{otherwise.} \end{cases} \tag{3.14}$$

Thus the $C(i, j, k)$ satisfy Pascal's identity, but with different boundary conditions.

THEOREM 3.15. *Let* $j \geqq 0$, $i \geqq (j+1)k-1$; *then*

$$C(i, j, k) = \binom{i}{j+1} - (k-1)\binom{i}{j}.$$

Proof. By induction on i, j. The case $j = 0$, $i \geqq k-1$ is the left boundary condition of Recursion 3.14.

Suppose that $j>0$ and that the result is true for $j-1$. Let $i=(j+1)k-1$. Then

$$\begin{aligned}
&\binom{(j+1)k-1}{j+1}-(k-1)\binom{(j+1)k-1}{j}\\
&\quad=\binom{(j+1)k-1}{j+1}-(k-1)\frac{((j+1)k-1)\cdots((j+1)(k-1)+1}{j!}\\
&\quad=\binom{(j+1)k-1}{j+1}-\frac{((j+1)k-1)\cdots((j+1)(k-1))}{(j+1)!}\\
&\quad=0\\
&\quad=C((j+1)k-2,j,k).
\end{aligned}$$

The rest follows from Pascal's identity. Q.E.D.

It is worth noting at this point that, although the $C(i, j, k)$ have been defined only for $k\geqq 2$, if the result of Theorem 3.15 is extended to the case $k=1$, then

$$C(i,j,1)=\binom{i}{j+1},\qquad j\geqq 0,\quad i\geqq j.$$

Thus the $C(i, j, k)$ extended to the case $k=1$ are a generalization of ordinary binomial coefficients, at least over their common range of definition.

From their definition, the $C(i, j, k)$ will play the same role in lexicographic ranking algorithm for $\mathbf{P}(kn+1; k)$ as did the $P(i, j)$ for lexicographic ranking of $\mathbf{P}(n)$. In this algorithm, the $C(i, j, k)$ are stored in the array $ck\,[1:n*k, 0:n-2]$; as before the details of computing this array are omitted from the algorithm. The permutation $\sigma\in\mathbf{P}(nk+1; k)$ is represented by an array $s[1: n*k+1]$.

ALGORITHM 3.16. Input: $\sigma\in\mathbf{P}(nk+1; k)$ represented as an array $s[1:n*k+1]$. Output: the lexicographic rank of σ in $\mathbf{P}(nk+1; k)$.

```
begin
  compute ck[1:n*k, 0:n-2];
  lexrank: =0;
  j: =0;
  for i: =k-1 until n*k-2 do
  if s[n*k-i+1]=(n-j)*k+1 then
  begin
    lexrank: =lexrank+ck[i, j];
    j: =j+1;
  end;
end.
```

Note that this algorithm is not recursive, and in fact does not explicitly construct the permutation τ of Definition 3.13. This is because the position of $((n-l)k+1)$, $l>j$, is unaffected by the movement of the substring $((n-j)k+$

$1)((n-j)k)\cdots((n-j-1)k+2)$ to the right. Thus the statement of the algorithm is particularly simple. This property of σ also leads to a nonrecursive unranking algorithm based on the $C(i, j, k)$. The unranking algorithm employs a stack for temporary storage.

ALGORITHM 3.17. Input: integers $n \geqq 0$, $0 \leqq r \leqq \mathbf{P}(nk+1; k)-1$. Output: the rank r element of $\mathbf{P}(nk+1; k)$ represented as an array $s[1: n * k+1]$.

```
begin
  compute ck[1: n * k, 0: n-2];
  for j: = n-2 step -1 until 0 do
  begin
    choose maximum i such that ck[i, j] ≦ r;
    a[n-j]: = i;
    r: = r-ck[i, j];
  end
  s[1]: = 1;
  j: = 2;
  for i = 2 until k+1 do add i to stack;
  for l: = 2 until n * k+1 do
  begin
    if n * k-a[j]+1 = l then
    begin
    for q: = j * (n-1)+2 until j * n+1 do add q to stack;
    j: = j+1;
  end;
  s[l]: = top of stack;
    delete top of stack;
  end;
end.
```

THEOREM 3.18. *Algorithms* 3.16 *and* 3.17 *are* $O(nk)$ *space-bounded; and with the exception of computing ck* $[1: n * k, 0: n-2]$ *they are* $O(nk)$ *time-bounded.*

Proof. The proof of the space bounds is similar to those in Theorems 2.20 and 2.23.

The time bound on Algorithm 3.16 is clear since the algorithm makes a single pass through $s[1: n * k+1]$ from right to left.

To establish the time-bound for Algorithm 3.17 it must be shown that the array $a[2: n]$ can be constructed in $O(nk)$ time. Let $0 \leqq j < n-2$; suppose that $a[j] = i$. Then in the rank r permutation $\sigma \in \mathbf{P}(nk+1; k)$,

$$s_{nk-i+1} = jk+1.$$

By the definition of $\mathbf{P}(nk+1; k)$,

$$s_{nk-i'+1} = (j+1)k+1$$

for some $i' > i$. Hence the entries of $a[2:n]$ are strictly increasing, and $a[2:n]$ can be constructed by a sequential scan of the integers $1, \cdots, nk$. Q.E.D.

As a consequence of Theorem 3.18, it is possible to implement Algorithms 3.16 and 3.17 in linear time, by precomputing the factorials of the integers $0, 1, \cdots, nk$ and storing them in an array; the evaluations of $ck[i, j]$ in Algorithms 3.16 and 3.17 may be accomplished by using the result of Theorem 3.15. Since at most nk such evaluations are required by the algorithms, the result is a linear time-bound. This approach is, however, not feasible in practice, due to the rapid growth of $n!$ as a function of n.

4. Listing algorithms.

procedure *next* (σ, n, m);
  begin
  comment $\sigma \in \mathbf{P}(n, m)$ *for* $m > 1$;
  find least j *such that* $s_{m+j} \neg = n-j$ *or* $m+j > n$;
  if $j = 1$ **then**
    begin
    m: $= m-1$;
    σ: $= s_1 \cdots s_{m-2}(n)s_{m-1} \cdots s_n$;
    end;
  else
    begin
    comment $\sigma = s_1 s_2 \cdots s_{m-1}(n)(n-1) \cdots (n-j+1)s_{m+j} \cdots s_n$
      where $s_{m+j} \cdots s_n$ *may be the empty string*;
    m: $= n$;
    σ: $= s_1 s_2 \cdots s_{m-2}(n-j+1)s_{m-1}s_{m+j} \cdots s_n(n-j+2) \cdots (n)$;
    end;
  end *next*;

Clearly, setting $\sigma = 1\,2 \cdots n$ and iterating *next* will generate all binary trees in lexicographic order. The cost of each iteration is $O(j)$; for a particular j in the range $1 \leqq j \leqq n-2$, the number of strings in $\mathbf{P}(n)$ for which $O(j)$ time is required is

$$B(n)-B(n-1), \qquad j=1;$$

$$\sum_{n=2}^{n-j+1} P(n-j+1, \mu) = B(n-j+1)-1, \qquad 2 \leqq j \leqq n-2.$$

Hence the total time required to generate all elements of $\mathbf{P}(n)$ by iteration of *next*

is bounded by a constant multiple of

$$B(n)-B(n-1)+\sum_{j=2}^{n-2} jB(n-j+1)<B(n)+\sum_{j=0}^{n-4}(j+2)B(n-j-1).$$

Now, for $j\geqq 0$, $j+2\leqq 3B(j)$; hence the time required is bounded by a constant multiple of

$$\begin{aligned} &B(n)+3\sum_{j=0}^{n-4} B(j)B(n-j-1)\\ &\quad<B(n)+3\sum_{j=0}^{n-1} B(j)B(n-j-1)\\ &\quad=4B(n). \end{aligned}$$

Hence the expected time required by each iteration of *next* is bounded by a constant.

A similar analysis applies to $\mathbf{P}(nk+1, k)$.

5. Final remarks. The notion of ranking combinatorial configurations was first presented in [7]; since then a considerable amount of work has been done on the subject; see [2], [3], [4], [5], [9], [10], [11], [12].

The problem of ranking binary trees is also considered in [4] and [5], and ranking k-ary trees is considered in [8]. It is shown in [9] that the method of [5], when generalized to k-ary trees, leads to algorithms which are exponential in k.

In [4], a full binary tree is represented by the sequence of level numbers of its leaves; from this, a number $T(n, k)$ is defined which is related to the $P(n, m)$ of § 2 by the equation

$$P(n, m)=T(n, n-m+1).$$

The results of [4] are generalized to the k-ary case in [8]; however, there does not appear to be any relation between the coefficients used in [8] and the $C(i, j, k)$ defined here.

The ranking algorithms presented here differ from the usual procedure for ranking permutations and n-tuples, as defined in [10], [11]. This procedure may be described as follows: to rank $s_1\cdots s_n$ lexicographically, let $A_i=\{t_1\cdots t_n : t_j=s_j 1\leqq j\leqq i-1 \text{ and } t_i<s_i\}$. The rank of $s_1\cdots s_n$ is simply $\sum_{i=1}^{n}|A_i|$. This idea may be represented graphically by binomial grids [10] or reduction diagrams [11].

If the methods of [10] or [11] are applied to Recurrence 2.15, the following linear order is defined on $\mathbf{P}(n)$: $\sigma<\tau$ in $\mathbf{P}(n)$ iff

(i) $\sigma\in\mathbf{P}(n, m)$, $\tau\in\mathbf{P}(n, m')$ and $m<m'$; or

(ii) $\sigma, \tau\in\mathbf{P}(n, m)$ and $\sigma(n)<\tau(n)$ in $\mathbf{P}(n-1)$.

The corresponding ranking and unranking algorithms are $O(n^2)$ time and $O(n)$ space bounded. A similar remark applies to $\mathbf{P}(nk+1; k)$.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1973.

[2] E. Calabi and H. Wilf, *On the sequential and random selection of subspaces over a finite field*, J. Combinatorial Theory Ser. A, 22 (1977), pp. 107–109.

[3] J. FILLMORE AND S. G. WILLIAMSON, *Ranking algorithms: The symmetries and colorations of the n-cube*, this Journal, 5 (1976), pp. 297–304.
[4] F. RUSKEY AND T. C. HU, *Generating binary trees lexicographically*, this Journal, 6 (1977), pp. 745–758.
[5] G. D. KNOTT, *A numbering system for binary trees*, Comm. ACM, 20 (1977), pp. 113–115.
[6] D. E. KNUTH, *The Art of Computer Programming*, vol. I, Addison-Wesley, Reading, MA, 1973.
[7] D. H. LEHMER, *The machine tools of combinatorics*, Applied Combinatorial Mathematics, E. F. Beckenbach ed., Wiley, New York, (1964), pp. 5–31.
[8] F. RUSKEY, *Generating t-ary trees lexicographically*, this Journal, to appear.
[9] A. E. TROJANOWSKI, *On the ordering, enumeration, and ranking of k-ary trees*, Technical Report Number UIUCDCS-R-77-850, Department of Computer Science, University of Illinois, February, 1977.
[10] H. S. WILF, *A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects*, preprint, University of Pennsylvania.
[11] S. G. WILLIAMSON, *On the ordering, ranking, and random generation of basic combinatorial sets*, Proceedings of the Table Ronde, Combinatoire et Representation du Groupe Symmetrique (Strasbourg, France, April 26–30, 1976), to appear.
[12] ———, *Ranking algorithms for lists of partitions*, this Journal, 5 (1976), pp. 602–617.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0010 $01.00/0

GENERALIZED CONNECTORS*

NICHOLAS PIPPENGER†

Abstract. An n-connector is an acyclic directed graph having n inputs and n outputs and satisfying the following condition: given any one-to-one correspondence between inputs and distinct outputs, there exists a set of vertex-disjoint paths that join each input to the corresponding output. It is known that the minimum possible number of edges in an n-connector lies between lower and upper bounds that are asymptotic to $3n \log_3 n$ and $6n \log_3 n$ respectively. A generalized n-connector satisfies the following stronger condition: given any one-to-many correspondence between inputs and disjoint sets of outputs, there exists a set of vertex-disjoint trees that join each input to the corresponding set of outputs. It is shown that the minimum number of edges in a generalized n-connector is asymptotic to the minimum number in an n-connector.

Imagine an information transmission network intended to mediate between n sources of information and n users of this information. At any time, any of the users may wish to be connected with any of the sources; a user can be connected with only one source at a time, but many users may wish to be connected with the same source. This paper deals with an idealized version of the problem of designing a network capable of providing any such pattern of simultaneous connections.

An *(n, m)-graph* is an acyclic directed graph with a set of n distinguished vertices called *inputs* and a disjoint set of m distinguished vertices called *outputs*. An n-graph is an (n, n)-graph.

An *n-connector* is an n-graph satisfying the following condition: given any one-to-one correspondence between inputs and distinct outputs, there exists a set of vertex-disjoint paths that join each input to the corresponding output. (A *path* joining an input to an output is a directed path whose origin is the input and whose destination is the output.) Let $c(n)$ denote the minimum possible number of edges in an n-connector; it is known that

$$3n \log_3 n \leqq c(n) \leqq 6n \log_3 n + O(n)$$

(see Pippenger and Valiant [4, Remark 2.2.6]).

A *generalized n-connector* is an n-graph satisfying the following stronger condition: given any one-to-many correspondence between inputs and disjoint sets of outputs, there exists a set of vertex-disjoint trees that join each input to the corresponding set of outputs. (A *tree* joining an input to a set of outputs is a directed tree whose root is the input and whose leaves are the outputs.) Let $d(n)$ denote the minimum possible number of edges in a generalized n-connector; that

$$d(n) \leqq 10n \log_2 n + O(n)$$

for n a power of 2 is implicit in the work of Ofman [1]. Thompson [5] has recently shown that

$$d(n) \leqq 12n \log_3 n + O(n)$$

for n a power of 3.

The object of this note is to show that

$$d(n) = c(n) + O(n),$$

* Received by the editors May 13, 1977.

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

and thus that

$$d(n) \sim c(n).$$

It is clear that

$$d(n) \geqq c(n);$$

thus it will suffice to show that

$$d(n) \leqq c(n) + O(n). \tag{1}$$

This will be done by means of a new type of graph which will be called a generalizer. An *n-generalizer* is an n-graph that satisfies the following condition: given any correspondence between inputs and nonnegative integers that sum to at most n, there exists a set of vertex-disjoint trees that join each input to the corresponding number of distinct outputs. Let $g(n)$ denote the minimum possible number of edges in an n-generalizer; it will be shown below that

$$g(n) \leqq 120n + O((\log n)^2), \tag{2}$$

so that in particular

$$g(n) = O(n).$$

A generalized n-connector can be obtained from an n-generalizer and an n-connector by identifying the outputs of the generalizer with the inputs of the connector, as shown in Fig. 1. It is obvious that this yields a generalized n-connector: the generalizer provides the appropriate number of copies of each input, and the connector joins these copies to the appropriate outputs. Thus

$$\begin{aligned} d(n) &\leqq c(n) + g(n) \\ &\leqq c(n) + O(n), \end{aligned}$$

which completes the proof of (1).

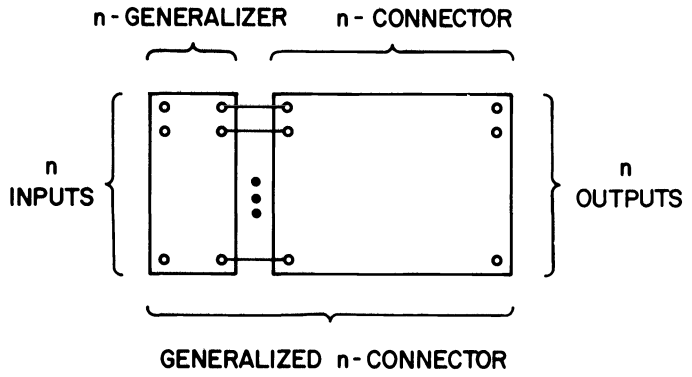


FIG. 1.

It remains to prove (2). To do this, two more types of graphs, called concentrators and superconcentrators, will be needed.

An n-superconcentrator is an n-graph that satisfies the following condition: given any set of inputs and any equinumerous set of outputs, there exists a set of vertex-disjoint paths that join the given inputs in a one-to-one fashion to the given outputs. Let $s(n)$ denote the minimum possible number of edges in an n-superconcentrator; that

$$s(n) \leqq 234n$$

was shown by Valiant [6], who first defined superconcentrators. Pippenger [3] subsequently showed that

$$s(n) \leqq 39n + O(\log n).$$

An (n, m)-*concentrator* is an (n, m)-graph that satisfies the following condition: given any set of m or fewer inputs, there exists a set of vertex-disjoint paths that join the given inputs in a one-to-one fashion to distinct outputs. Let $r(n, m)$ denote the minimum possible number of edges in an (n, m)-concentrator; that

$$r(n, m) \leqq 29n$$

was shown by Pinsker [2], who first defined concentrators. It will now be shown that

$$r(n, \lfloor n/2 \rfloor) \leqq 20n + O(\log n), \tag{3}$$

where $\lfloor \cdots \rfloor$ denotes "the greatest integer less than or equal to ...".

A $(n, \lfloor n/2 \rfloor)$-concentrator can be obtained by combining $\lfloor n/2 \rfloor$ edges with an $\lceil n/2 \rceil$-superconcentrator (where $\lceil \cdots \rceil$ denotes "the least integer greater than or equal to ..."), as shown in Fig. 2. It is obvious that this yields an $(n, \lfloor n/2 \rfloor)$-

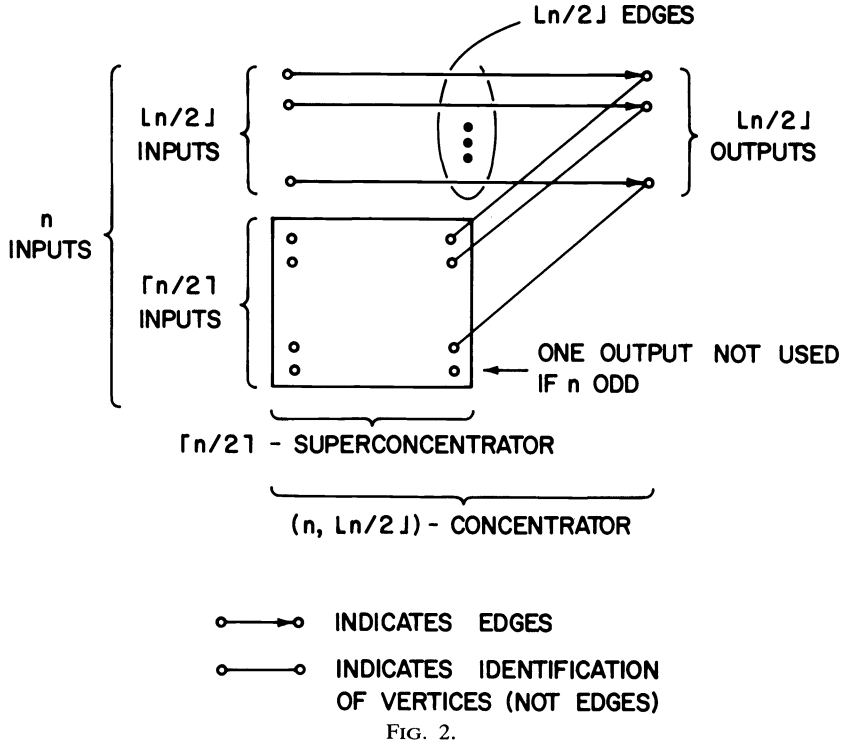


FIG. 2.

concentrator: those of the given inputs that lie among the upper $\lfloor n/2 \rfloor$ inputs can be joined to distinct outputs through the edges; those that lie among the lower $\lceil n/2 \rceil$ can be joined to other distinct outputs through the superconcentrator. Thus

$$\begin{aligned} r(n, \lfloor n/2 \rfloor) &\leqq \lfloor n/2 \rfloor + s(\lceil n/2 \rceil) \\ &\leqq \lfloor n/2 \rfloor + 39\lceil n/2 \rceil + O(\log \lceil n/2 \rceil) \\ &\leqq 20n + O(\log n), \end{aligned}$$

which completes the proof of (3).

It still remains to prove (2). This will be done by means of a recursive construction: an n-generalizer can be obtained by combining an $(n, \lfloor n/2 \rfloor)$-concentrator, an $\lfloor n/2 \rfloor$-generalizer, $2\lfloor n/2 \rfloor$ edges, and an n-superconcentrator, as shown in Fig. 3. This can be seen to yield an n-generalizer as follows. If an input is to be joined to x

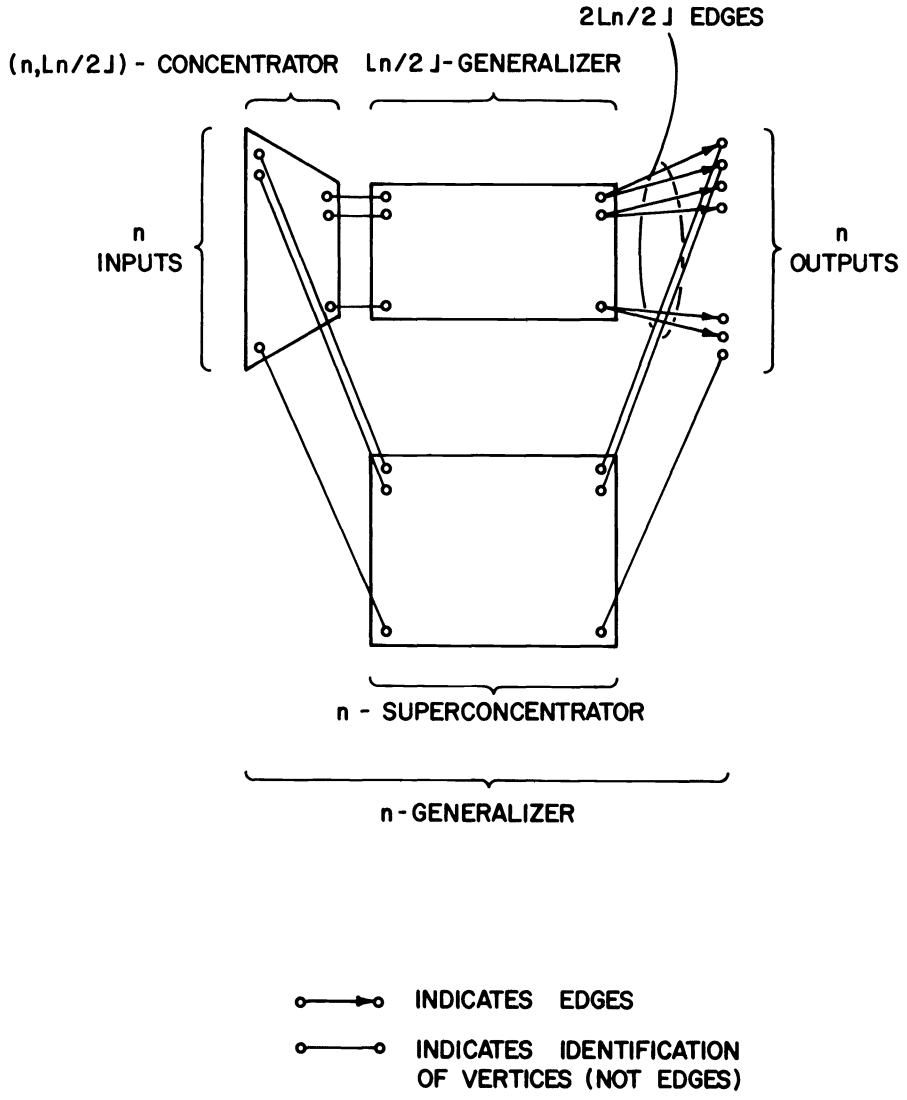


FIG. 3.

distinct outputs, one can write $x = 2y + z$, where y is a nonnegative integer and z is either 0 or 1. Since the x's sum to at most n, there can be at most $\lfloor n/2 \rfloor$ inputs for which y is greater than 0. Each of these inputs can therefore be joined to a distinct output of the concentrator, thence to y distinct outputs of the $\lfloor n/2 \rfloor$-generalizer, and finally to $2y$ distinct outputs of the n-generalizer. All that remains is to join the inputs for which z is 1 to other distinct outputs; this can be done through the superconcentrator. Thus

$$\begin{aligned} g(n) &\leqq g(\lfloor n/2 \rfloor) + r(n, \lfloor n/2 \rfloor) + 2\lfloor n/2 \rfloor + s(n) \\ &\leqq g(\lfloor n/2 \rfloor) + 20n + O(\log n) + 2\lfloor n/2 \rfloor + 39n + O(\log n) \\ &\leqq g(\lfloor n/2 \rfloor) + 60n + O(\log n) \\ &\leqq 120n + O((\log n)^2), \end{aligned}$$

which completes the proof of (2).

The result of this note is satisfying from a theoretical point of view: information-theoretic considerations suggest that since

$$\log n^n = \log n! + O(n)$$

one should have

$$d(n) = c(n) + O(n),$$

as has indeed been shown to be the case. The proof technique used in this note, however, does not endow the result with any practical significance: $120n$ exceeds $6n \log_3 n$ until n exceeds $3^{20} = 3{,}486{,}784{,}401$.

Acknowledgment. The author is indebted to Clark Thompson for suggesting the possibility of proving the existence of linear generalizers.

REFERENCES

[1] YU. P. OFMAN, *Universalnyi avtomat*, Trudy Moskov. Mat. Obšč., 14 (1965), pp. 186–199 = *A universal automaton*, Trans. Moscow Math. Soc., 14 (1965), pp. 200–215.

[2] M. S. PINSKER, *On the complexity of a concentrator*, Proc. 7th Internat. Teletraffic Conf., Stockholm, 1973, pp. 318/1–318/4.

[3] N. J. PIPPENGER, *Superconcentrators*, this Journal, 6, (1977), pp. 298–304.

[4] N. J. PIPPENGER AND L. G. VALIANT, *Shifting graphs and their applications* J. Assoc. Comput. Mach., 23 (1976), pp. 423–432.

[5] C. D. THOMPSON, *Generalized connection networks for parallel processor interconnection*, Carnegie-Mellon Univ. Tech. Rep., Pittsburgh, May 1977.

[6] L. G. VALIANT, *On non-linear lower bounds in computational complexity*, Proc. 7th Ann. ACM Symp. on Theory of Computing, Albuquerque, 1975, Assoc. Comput. Mach., New York, pp. 45–53.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0011 $01.00/0

ON THE LOOP SWITCHING ADDRESSING PROBLEM

ANDREW CHI-CHIH YAO

Abstract. The following graph addressing problem was studied by Graham and Pollak in devising a routing scheme for Pierce's loop switching network. Let G be a graph with n vertices. It is desired to assign to each vertex v_i an address in $\{0, 1, *\}^l$, such that the Hamming distance between the addresses of any two vertices agrees with their distance in G. Let $N(G)$ be the minimum length l for which an assignment is possible. It was shown by Graham and Pollak that $N(G) \leqq m_G(n-1)$, where m_G is the diameter of G. In the present paper, we shall prove that $N(G) \leqq 1.09(\lg m_G)n + 8n$ by an explicit construction. This shows in particular that any graph has an addressing scheme of length $O(n \log n)$.

Key words. addressing scheme, binary tree, graph, Hamming distance, loop switching network

1. Introduction. An interesting routing scheme to Pierce's loop switching network [7] was proposed by Graham and Pollak [3], [4] (see also [1]). In this scheme, Pierce's network is represented by a graph where vertices stand for the loops, and edges stand for the contacts between loops in the network. The scheme calls for assigning a sequence of ternary symbols to each vertex such that the distances between vertices in the graph are faithfully represented. The combinatorial problem is described below; for a detailed discussion of the connection between Pierce's network and this combinatorial problem, as well as further information on the subject, see references [1], [3], [4], [7].

Throughout our discussion, $G = (V, E)$ will be a connected graph with a set V of vertices, and a set E of undirected edges. A *path of length t* in G from a vertex v_i to a vertex v_j is a sequence of vertices $v_{k_0}, v_{k_1}, \cdots, v_{k_t}$ such that $v_{k_0} = v_i$, $v_{k_t} = v_j$, and $\{v_{k_{s-1}}, v_{k_s}\} \in E$ for $s = 1, 2, \cdots, t$. The *distance* $d_G(v_i, v_j)$ between vertices v_i and v_j is the minimum length t for which a path of length t from v_i to v_j exists. The diameter of G, denoted by m_G, is the largest distance between any two vertices in G. That is, $m_G = \max \{d_G(v_i, v_j) | v_i, v_j \in V\}$.

Let Σ be the ternary symbol set $\{0, 1, *\}$. (The character "*" is a "don't-care" symbol.) The *Hamming distance* H between elements in Σ is defined by $H(1, 0) = H(0, 1) = 1$, and $H(a, b) = 0$ for all other pairs of a, b in Σ. For two sequences $\alpha = a_1 a_2 \cdots a_l$ and $\beta = b_1 b_2 \cdots b_l$ in Σ^l, where $l > 0$, their *Hamming distance* is given by

$$H(\alpha, \beta) = \sum_{1 \leqq i \leqq l} H(a_i, b_i).$$

An *addressing scheme* for a graph $G = (V, E)$ with n vertices is an assignment of a sequence $c(v_i) \in \Sigma^l$ to each vertex v_i such that $H(c(v_i), c(v_j)) = d_G(v_i, v_j)$ for all v_i, v_j in V. The positive integer l is called the *length* of the addressing scheme, and the sequence $c(v_i)$ the *address* of vertex v_i. It is desired to find addressing schemes with small length. Let $N(G)$ be the minimum l for which an addressing scheme of length l exists for G. In [3], it was proved that an addressing scheme always exists (i.e., $N(G) < \infty$), and furthermore, $N(G) \leqq m_G(n-1)$. We shall improve this bound by explicitly constructing an addressing scheme. The main results are as follows: (We shall use λ to denote the constant $(\frac{1}{3} \lg 3 + \frac{2}{3} \lg \frac{3}{2})^{-1} \approx 1.09$.)

* Received by the editors October 20, 1977, and in final revised form February 27, 1978. This research was supported in part by the National Science Foundation under Grant MCS 72-03752 A03.

† Computer Science Department, Stanford University, Stanford, California 94305.

THEOREM 1. *For a graph G with n vertices,*

$$N(G) \leqq \lambda n \lg n + 2n.$$

THEOREM 2. *For a graph G with n vertices,*

$$N(G) \leqq \lambda n (\lg m_G) + 8n.$$

Note: lg means logarithm to the base 2.

2. Definitions and preliminaries. Let $G=(V, E)$ be a (connected, undirected) graph. A path $v_{k_0}, v_{k_1}, \cdots, v_{k_t}$ in G is *simple* if all the vertices v_{k_s} for $0 \leqq s \leqq t$ are distinct, except possibly for $v_{k_0} = v_{k_t}$. A graph $G'=(V', E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G'=(V', E')$ is said to be a *tree* in G if G' is connected and there is no simple path of length >0 in G' from any vertex $v \in V'$ to itself. A tree $G'=(V', E')$ in G is a *spanning tree* for G if $V'=V$. For any subset of vertices $V' \subseteq V$, the diameter of V', written $\mathrm{diam}_G(V')$, is $\max\{d_G(v_i, v_j) \mid v_i, v_j \in V'\}$. In particular, $\mathrm{diam}_G(V) = m_G$. The *distance* $d_G(v_i, V')$ between a vertex $v_i \in V$ and a subset $V' \subseteq V$ is defined as $d_G(v_i, V') = \min\{d_G(v_i, v_j) \mid v_j \in V'\}$.

We shall make use of *binary trees* in our design process. (See for example Knuth [5] for basic definitions regarding binary trees.) Let T be a binary tree with n leaves. Assume the nodes of T are numbered arbitrarily from 1 to $2n-1$. The node with number k will be denoted by r_k. We will also use the notation $\boxed{i}$ for a leaf numbered i, and (j) for an internal node numbered j. For a node r_k, let $R(k)$ be the subset of leaves in T which are descendants of r_k. The size of $R(k)$ is called the *weight* of r_k, denoted by $w(k)$. For example, we have $R(1)=\{r_8, r_6, r_9\}$, $R(2)=\{r_2\}$, and $w(1)=3$, $w(2)=1$ in Fig. 1. The *external path length* $P(T)$ is defined by the following equation:

$$P(T) = \sum_{\text{internal node } r_k} w(k). \tag{1}$$

The quantity $P(T)$ can alternatively be described as the sum of the distances from the leaves to the root [5]. If r_i and r_j are respectively the leftson and the rightson of r_k, we shall write $i = \mathrm{leftson}(k)$, $j = \mathrm{rightson}(k)$; $k = \mathrm{father}(i) = \mathrm{father}(j)$; and $j = \mathrm{brother}(i)$, $i = \mathrm{brother}(j)$. As a shorthand, we shall use $\bar{k}$ for $\mathrm{father}(k)$ and k' for $\mathrm{brother}(k)$. A binary tree T is said to be *weight-balanced* if for each internal node r_k,

$$\begin{aligned} &\tfrac{1}{3} w(k) \leqq w(\mathrm{leftson}(k)) \leqq \tfrac{2}{3} w(k), \\ &\tfrac{1}{3} w(k) \leqq w(\mathrm{rightson}(k)) \leqq \tfrac{2}{3} w(k). \end{aligned} \tag{2}$$

The following result is from [6, Thm. 2].

LEMMA 1 [Nievergelt and Wong]. *If T is a weight-balanced binary tree with n leaves, then the external path length of T satisfies*

$$P(T) \leqq \lambda n \lg n \approx 1.09 n \lg n.$$

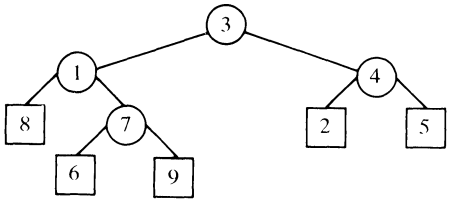


FIG. 1

In a binary tree T, if a leaf $\boxed{i}$ precedes another leaf $\boxed{j}$ in post-order [5], we shall say that $\boxed{i}$ is to the left of $\boxed{j}$ (or $\boxed{j}$ is to the right of $\boxed{i}$), and write $\boxed{i} \prec \boxed{j}$ (or equivalently $\boxed{j} \succ \boxed{i}$). We further extend the relation so that

$$\boxed{i} \prec r_k \quad \text{if} \quad \boxed{i} \prec \boxed{j} \quad \text{for all descendants } \boxed{j} \text{ of } r_k,$$

$$\boxed{i} \succ r_k \quad \text{if} \quad \boxed{i} \succ \boxed{j} \quad \text{for all descendants } \boxed{j} \text{ of } r_k.$$

Clearly, for any leaf $\boxed{i}$ and node r_k, either $\boxed{i} \prec r_k$, $\boxed{i} \succ r_k$, or $\boxed{i}$ is a descendant of r_k; and exactly one of the three relations holds. In Fig. 1, we have $\boxed{6} \prec \boxed{2}$, $\boxed{6} \prec ④$, and $\boxed{2} \succ ⑦$.

3. The construction of a length $O(n \lg n)$ addressing scheme.

3.1. The design tree. The key to obtaining an $O(n \lg n)$ scheme is by using a hierarchical design. A *design tree* M is a pair (T, f) where T is a binary tree with n leaves, and f is a one-to-one mapping from the leaves of T to the vertices of G. For notational convenience, we shall number the nodes of T in such a way that the leaves receive numbers 1 to n and leaf $\boxed{i}$ is associated with vertex v_i under f. The root of T will be labeled with $2n-1$; and the remaining internal nodes with $n+1$ through $2n-2$ (their actual numbering will be unimportant for M).

We now describe an addressing scheme $Z(M)$ corresponding to a given design tree M. Every address $c(v_i)$ in $Z(M)$ will consist of $2n-2$ blocks of code, where the kth block has length l_k (to be defined later) and is conceptually associated with the node r_k of T. (Note that r_k cannot be the root since $k \neq 2n-1$.) Thus we shall write for $1 \leqq i \leqq n$,

$$c(v_i) = c_{i1}c_{i2} \cdots c_{i,2n-2} \quad \text{where} \quad |c_{ik}| = l_k. \tag{3}$$

By definition, the Hamming distance between two addresses $c(v_i)$ and $c(v_j)$ is equal to the sum of the Hamming distances between corresponding blocks. That is,

$$H(c(v_i), c(v_j)) = \sum_{k=1}^{2n-2} H(c_{ik}, c_{jk}). \tag{4}$$

We shall design the code in such a way that in (4), only a few terms will contribute to the sum, other terms being zero. For example, consider the design tree M shown in Fig. 2. We shall in fact have

$$H(c(v_3), c(v_2)) = H(c_{3,10}, c_{2,10}) + H(c_{3,11}, c_{2,11}) + H(c_{3,2}, c_{2,2}) \tag{5}$$

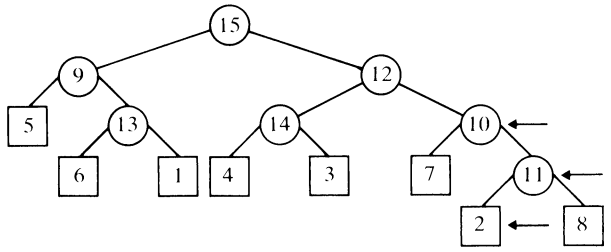


FIG. 2. *A design tree M with a marked path.*

and $H(c_{3,k}, c_{2,k}) = 0$ for $k \notin \{10, 11, 2\}$. The trick to achieve $H(c(v_3), c(v_2)) = d_G(v_3, v_2)$ is as follows. Define $S(k) = \{f(r_i) \mid r_i \in R(k)\}$ i.e., $S(k)$ is the set of vertices

associated with the leaf descendants of r_k. We shall require that,

$$
\begin{aligned}
&H(c_{3,10}, c_{2,10}) = d_G(v_3, S(10)),\\
&H(c_{3,10}, c_{2,10}) + H(c_{3,11}, c_{2,11}) = d_G(v_3, S(11)),\\
&H(c_{3,10}, c_{2,10}) + H(c_{3,11}, c_{2,11}) + H(c_{3,2}, c_{2,2}) = d_G(v_3, S(2)).
\end{aligned} \tag{6}
$$

We can view (6) in the following way. Starting at the *lowest common ancestor* (lca) of $\boxed{3}$ and $\boxed{2}$ (i.e., the common ancestor of $\boxed{3}$ and $\boxed{2}$ farthest from the root), which is r_{12}, we move down the path r_{10}, r_{11}, to the leaf r_2. Each node r_k encountered along the path, excluding the lca, will add a block of code which creates enough Hamming distance to bring the total up to $d(v_3, S(k))$. An equivalent form of (6) is

$$H(c_{3,k}, c_{2,k}) = d_G(v_3, S(k)) - d_G(v_3, S(\bar{k})) \tag{7}$$

for $k = 10, 11, 2$, and $\bar{k} =$ father (k). In general, we want to achieve the following. For $\boxed{i} \lessdot \boxed{j}$, let node h_0 be the lowest common ancestor of $\boxed{i}$ and $\boxed{j}$, and $h_0, h_1, \cdots, h_t = j$ be the path from node h_0 to $\boxed{j}$, then

$$
\begin{aligned}
&H(c_{i,k}, c_{j,k}) = d(v_i, S(k)) - d(v_i, S(\bar{k})) \quad \text{for } k = h_1, h_2, \cdots, h_t, \text{ and } \bar{k} = \text{father } (k);\\
&H(c_{i,k}, c_{j,k}) = 0 \quad \text{for all other } k.
\end{aligned} \tag{8}
$$

It is easy to verify that (8), if true for all $\boxed{i} \lessdot \boxed{j}$, will be sufficient to guarantee that $Z(M) = \{c(v_i) \mid 1 \leqq i \leqq n\}$, as given by (3), is an addressing scheme. That is, $d_G(v_i, v_j) = H(c(v_i), c(v_j))$ for all i, j. We now describe a construction of the c_{ik}'s that satisfy (8).

$Z(M)$: *The addressing scheme induced by M.* For each k, $1 \leqq k \leqq 2n-2$, let

$$l_k = \max_{1 \leqq i \leqq n} [d_G(v_i, S(k)) - d_G(v_i, S(\bar{k}))]. \tag{9}$$

The block c_{ik}, for $1 \leqq i \leqq n$, has length l_k and is given by

$$
c_{ik} = \begin{cases}
000 \cdots \cdots 0 & \text{if } \boxed{i} \text{ is a descendant of } r_k,\\
*** \cdots \cdots * & \text{if } \boxed{i} \succ r_k,\\
\underbrace{111 \cdots 1}_{\delta} *** \cdots * & \text{with } \delta = d_G(v_i, S(k)) - d_G(v_i, S(\bar{k})),\\
 & \text{if } \boxed{i} \lessdot r_k.
\end{cases} \tag{10}
$$

Finally, form $Z(M) = \{c(v_i) \mid 1 \leqq i \leqq n\}$ according to (3). The length of $Z(M)$, denoted by $\tau(M)$, is

$$\tau(M) = \sum_{1 \leqq k \leqq 2n-2} l_k. \tag{11}$$

To see that $Z(M)$ is actually an addressing scheme, we need only show that (8) is satisfied. For $\boxed{i} < \boxed{j}$, we see from (10) that $H(c_{ik}, c_{jk}) = 0$ unless $\boxed{i} \lessdot r_k$ and $\boxed{j}$ is a descendant of r_k; in the latter case, $H(c_{ik}, c_{jk}) = d_G(v_i, S(k)) - d_G(v_i, S(\bar{k}))$. But this is exactly as required by (8). Q.E.D.

3.2. Criteria for a good design tree. Let us find out what sort of design tree M will generate a short addressing scheme. Notice that for any $1 \leqq i \leqq n$, $1 \leqq k \leqq 2n-2$, we have

$$d_G(v_i, S(k)) - d_G(v_i, S(\bar{k})) \leqq \operatorname{diam}_G (S(\bar{k})). \tag{12}$$

Inequality (12) is valid, since we can concatenate a path from v_i to the nearest point in

$S(\bar{k})$, with a path of length at most $\text{diam}_G(S(\bar{k}))$, to reach a vertex in $S(k)$. This tells us that

$$l_k \leqq \text{diam}_G(S(\bar{k})). \tag{13}$$

An upper bound to $\tau(M)$ is therefore

$$\tau(M) \leqq \sum_{1 \leqq k \leqq 2n-2} \text{diam}_G(S(\bar{k})) = 2 \sum_{n+1 \leqq k \leqq 2n-1} \text{diam}_G(S(k)), \tag{14}$$

every internal node being the father of two nodes. This upper bound will in general be $O(n^2)$, as the subset $S(k)$ may have diameter $O(n)$ for many k. However, if we insist on two conditions

(i) no two points in $S(k)$ are far apart compared to its size $|S(k)|$, specifically, $\text{diam}_G(S(k)) \leqq |S(k)|$; and

(ii) the binary tree is weight-balanced,

then (14) would give

$$\tau(M) \leqq 2 \sum_{n+1 \leqq k \leqq 2n-1} |S(k)| = 2 \cdot P(T) \leqq 2\lambda n \lg n \tag{15}$$

by Lemma 1.

To achieve conditions (i) and (ii), we use the following idea. Let us think of $M = (T, f)$ as a tree built topdown by successively breaking V into smaller parts. From this viewpoint, the tree in Fig. 2 is obtained by first dividing (at node 15) $\{v_1, v_2, \cdots, v_8\}$ into $\{v_5, v_6, v_1\}$ and $\{v_4, v_3, v_7, v_2, v_8\}$; each of the two resulting parts are further divided into $\{v_5\}$, $\{v_6, v_1\}$ at node 9, and $\{v_4, v_3\}$, $\{v_7, v_2, v_8\}$ at node 12, respectively. This process is repeated until we have only the singleton sets $\{v_i\}$.

We shall see that in building M in this fashion, it is possible to keep the points in each part close together (condition (i)), and also make the two parts more or less equal in size (condition (ii)) on each decomposition. We shall describe such a method next, and then perform a finer analysis improving the bound given by (15).

3.3. Constructing M from a spanning tree. We shall construct a design tree M with the properties (i) and (ii) given in § 3.2. Choose any spanning tree with edge set A for the graph G. Let us create a new vertex v_0 and a new edge $\{v_0, v_1\}$. We now define a one-to-one mapping φ between the edge set of the augmented spanning tree $A' = A \cup \{\{v_0, v_1\}\}$ and the vertex set V (without v_0). The mapping φ is obtained by regarding $(V \cup \{v_0\}, A')$ as a rooted tree with root v_0, and mapping each edge onto its "lower" end point. We shall then number the edges e_i in A' so that $\varphi(e_i) = v_i$. An example of this process is shown in Fig. 3.

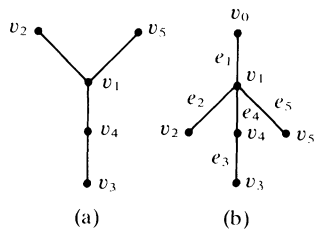


FIG. 3. (a) *A spanning tree on* $V = \{v_1, v_2, v_3, v_4, v_5\}$, *and* (b) *the labeling of its edges after augmentation.*

Our plan is to construct a binary tree Q by "suitably" splitting the edge set A' into two disjoint subsets, and repeat the process until only one edge remains in each

subset. Figure 4(a) shows the binary tree Q that may result from this process when applied to the spanning tree in Fig. 3(b). Although the tree Q so constructed is not a design tree on the vertex set, we can easily obtain such a design tree M_Q from Q in a natural way via the mapping φ. We shall transform Q into M_Q simply by identifying e_i with v_i in the tree Q. The design tree M_Q obtained from the Q in Fig. 4(a) is shown in Fig. 4(b).

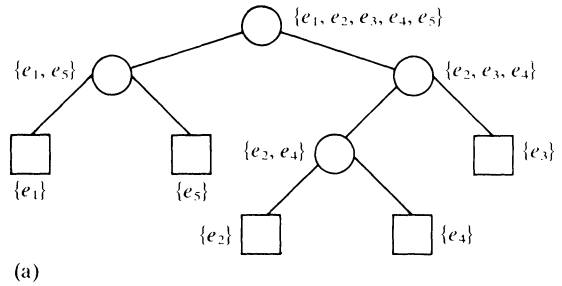


FIG. 4. (a) *A binary tree Q, and* (b) *its associated* M_Q.

We can now complete our task in two steps, (1) describe the topdown construction of a Q for which M_Q would satisfy conditions (i) and (ii), and (2) analyze the addressing scheme induced by such an M_Q.

(1) *Constructing Q.* A set of edges B in G is called a *tree set* if B is the edge set of some tree in G. Two tree sets B_1 and B_2 is said to form a *decomposition* of the tree set B if $B_1 \cap B_2 = \varnothing$ and $B_1 \cup B_2 = B$. Note that, in such a decomposition, there is a unique vertex v_s that is incident to both B_1 and B_2. For example, in Fig. 3(b), $B = \{e_2, e_4, e_5\}$ is a tree set. We can decompose B into $\{e_2\}$ and $\{e_4, e_5\}$ with v_1 being the unique vertex v_s.

A decomposition of B into B_1 and B_2 is *balanced* if $\frac{1}{3}|B| \leqq B_i \leqq \frac{2}{3}|B|$ for $i = 1, 2$. The following lemma is implicit in [2].

LEMMA 2 [Chung and Graham]. *Any tree set B with* $|B| \geqq 2$ *has a balanced decomposition into two tree sets.*

Let us now construct Q by breaking the augmented spanning tree A' into parts successively, using a balanced decomposition at each step. For example, the tree Q shown in Figure 4(a) can be obtained this way from A' in Figure 3(b). Once Q is constructed, we transform it into a design tree M_Q for the vertex set as described previously. It remains to analyze the address length obtained from this tree M_Q. To avoid confusion, we use $S(k)$ for the set of vertices associated with node r_k in M_Q, and use $B(k)$ to denote the tree set at the corresponding node in Q. Clearly, if $S(k) = \{v_{i_1}, v_{i_2}, \cdots v_{i_t}\}$, then $B(k) = \{e_{i_1}, e_{i_2}, \cdots, e_{i_t}\}$.

(2) *Analysis.* There are two simple properties of the design tree M_Q. Firstly, M_Q is weight-balanced by construction. Secondly, at any node r_k of M_Q, $\text{diam}_G(S(k)) \leqq |S(k)|$. This is true since any two vertices in $S(k)$ can be connected through at most $|S(k)|$ edges in the tree set $B(k)$. Thus, the two conditions (i) and (ii) in § 3.2 are satisfied, which implies $\tau(M) \leqq 2\lambda n \lg n$. A stronger bound can be obtained, however, by using the following lemma.

LEMMA 3. *For each node r_k in M_Q, and $1 \leqq i \leqq n$,*

$$d_G(v_i, S(k)) - d_G(v_i, S(\bar{k})) \leqq 1 + |S(k')|, \quad \textit{where } k' = \text{brother}(k). \tag{16}$$

Proof. Let v_j be a vertex in $S(\bar{k})$ closest to v_i, i.e.,

$$d_G(v_i, v_j) = d_G(v_i, S(\bar{k})). \tag{17}$$

If $v_j \in S(k)$, then $d_G(v_i, S(k)) = d_G(v_i, S(\bar{k}))$, and (16) is true. So we can assume that $v_j \in S(k')$.

Let v_s be the unique vertex that is incident to both an edge in $B(k')$ and an edge in $B(k)$. This implies that

$$d_G(v_j, v_s) \leqq |B(k')| = |S(k')|. \tag{18}$$

Now, let $\{v_s, v_t\}$ be an edge in $B(k)$ incident with v_s (see Fig. 5).

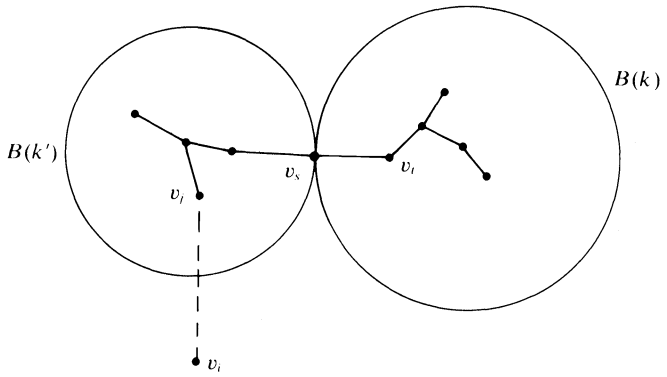


FIG. 5

Then,

$$d_G(v_i, v_s) \leqq d_G(v_i, v_j) + d_G(v_j, v_s) \leqq d_G(v_i, S(\bar{k})) + |S(k')|, \tag{19}$$

$$d_G(v_i, v_t) \leqq d_G(v_i, v_s) + 1 \leqq d_G(v_i, S(\bar{k})) + 1 + |S(k')|. \tag{20}$$

Since $\{v_s, v_t\} \in B(k)$, either v_s or v_t must be in $S(k)$. Therefore,

$$d_G(v_i, S(k)) \leqq \max\{d_G(v_i, v_s), d_G(v_i, v_t)\}. \tag{21}$$

Formula (16) follows from (19), (20), and (21). Q.E.D.

Lemma 3 implies that

$$l_k = \max_i \{d_G(v_i, S(k)) - d_G(v_i, S(\bar{k}))\} \leqq 1 + |S(k')|.$$

Therefore

$$\tau(M_Q) = \sum_{1 \leqq k \leqq 2n-2} l_k \leqq \sum_{1 \leqq k \leqq 2n-2} (1 + |S(k')|) = 2n - 2 + \sum_{1 \leqq k \leqq 2n-2} |S(k)|. \tag{22}$$

Making use of the fact that M_Q is weight-balanced and Lemma 1, we obtain after simplification,

$$\tau(M) \leqq \lambda n \lg n + 2n.$$

This proves Theorem 1.

3.4. Proof of Theorem 2. When m_G, the diameter of G, is substantially smaller than $n-1$, the addressing scheme we have constructed is better than the bound in Theorem 1 indicates. The key observation is that l_k is always no greater than m_G, because $l_k \leqq \max_i d_G(v_i, S(k)) \leqq m_G$. In the analysis of $\tau(M_Q) = \sum l_k$, we can thus use m_G to bound l_k, instead of $1 + |S(k')|$, for some of the nodes r_k.

Let X be the set of nodes r_k in M_Q such that $|S(k)| \leqq m_G$, and $|S(\bar{k})| > m_G$. For each $r_k \in X$, let $J_k = \{r_j | r_j$ is descendant of $r_k, r_j \neq r_k\}$. Let $J = \cup_{r_k \in X} J_k$. In Fig. 6, assume $m_G = 4$, the set X then consists of the nodes marked by arrows, and J is the set of shaded nodes. We shall use inequality $l_k \leqq 1 + |S(k')|$ for the nodes $r_k \in J$, and use $l_k \leqq m_G$ for the remaining nodes in deriving a bound for $\tau(M_Q)$.

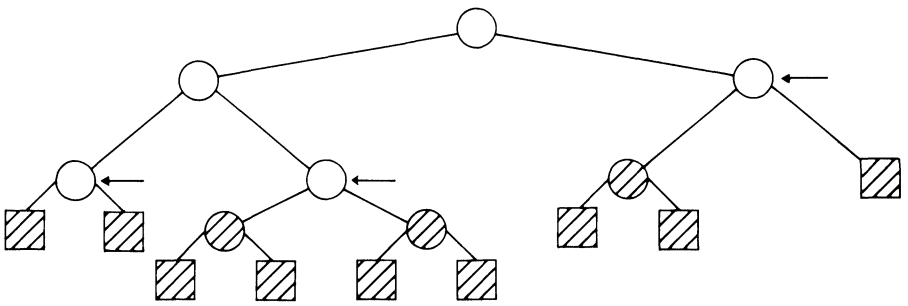


FIG. 6. *The set of shaded nodes is J.*

The following facts will be used in the calculation.

FACT 1. *Let q be the number of nodes not in J, then* $q < 6n/m_G$.

Proof. $|J| = \sum_{r_k \in X} (2 \cdot |S(k)| - 2) = 2(\sum_{r_k \in X} |S(k)|) - 2|X| = 2n - 2|X|$. Hence $q = 2n - 1 - |J| = 2|X| - 1$. Since $|S(k)| \geqq \frac{1}{3}|S(\bar{k})| \geqq \frac{1}{3} m_G$ for $r_k \in X$, we have $|X| \leqq n/(\frac{1}{3} m_G)$. Thus, $q < 2|X| \leq 6n/m_G$. Q.E.D.

FACT 2. *Let* $r_k \in X$, *then* $\sum_{r_j \in J_k} |S(j')| \leqq \lambda |S(k)| \lg |S(k)|$.

Proof. $\sum_{r_j \in J_k} |S(j')| = \sum_{r_j \in J_k} |S(j)|$. Fact 2 then follows from the fact that the subtree of M_Q rooted at r_k is weight-balanced. Q.E.D.

We can now prove the desired bound as follows:

$$\begin{aligned} \tau(M_Q) &= \sum_{r_k \notin J} l_k + \sum_{r_k \in J} l_k \leqq \sum_{r_k \notin J} m_G + \sum_{r_k \in J} (1 + |S(k')|) \\ &= q m_G + |J| + \sum_{r_k \in X} \sum_{r_j \in J_k} |S(j')| \\ &\leqq \frac{6n}{m_G} \cdot m_G + 2n + \lambda \sum_{r_k \in X} |S(k)| \lg |S(k)| \end{aligned} \tag{23}$$

where we have used Facts 1 and 2 in the last step.

Equation (23) leads by using $|S(k)| \leqq m_G$, to

$$\begin{aligned} \tau(M_Q) &\leqq 8n + \lambda (\lg m_G) \sum_{r_k \in X} |S(k)| \\ &= 8n + \lambda (\lg m_G) n. \end{aligned}$$

This completes the proof of Theorem 2. Q.E.D.

4. Remarks. In this paper we have given an algorithm which, for a graph with n vertices, constructs an addressing scheme of length $O(n \log n)$. The algorithm can be implemented straightforwardly, and has a $O(n^3)$ running time on a random access machine.

Fan Chung [private communication] pointed out that the bounds in Theorems 1 and 2 can be improved to $\frac{1}{2}\lambda n \lg n + 2n$ and $\frac{1}{2}\lambda n(\lg m_G) + 3.52n$ respectively, by a slight modification of the present constructions. However, we do not know of any construction that is guaranteed to give an address of length less than $O(n \lg n)$. The very attractive conjecture $N(G) \leqq n-1$ of Graham and Pollak [3], [4] thus still remains an open problem.

Acknowledgment. I wish to thank Ronald L. Graham for introducing this problem to me in a stimulating conversation on this subject.

REFERENCES

[1] L. H. BRANDENBURG, B. GOPINATH AND R. P. KURSHAN, *On the addressing problem of loop switching*, Bell System Tech. J., 51 (1972), pp. 1445–1469.

[2] F. R. K. CHUNG AND R. L. GRAHAM, *On graphs which contain all small trees*, J. Combinatorial Theory Ser. B, 24 (1978), pp. 14–23.

[3] R. L. GRAHAM AND H. O. POLLAK, *On the addressing problem for loop switching*, Bell System Tech. J., 50 (1971), pp. 2495–2519.

[4] ———, *On embedding graphs in squashed cubes*, Graph Theory and Applications, Lecture Notes in Mathematics, no. 303, Springer-Verlag (Proc. of a conference held at Western Michigan University, May 10–13, 1972).

[5] D. E. KNUTH, *Fundamental Algorithms*, The Art of Computer Programming, vol. 1, 2nd Ed., Addison-Wesley, Reading, MA, 1975.

[6] J. NIEVERGELT AND C. K. WONG, *Upper bounds for the total path length of binary trees*, J. Assoc. Comput. Mach., 20 (1973), pp. 1–6.

[7] J. R. PIERCE, *Network for block switching of data*, Bell System Tech. J., 51 (1972), pp. 1133–1145.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0012 $01.00/0

THE UNSOLVABILITY OF THE EQUIVALENCE PROBLEM FOR ε-FREE NGSM'S WITH UNARY INPUT (OUTPUT) ALPHABET AND APPLICATIONS*

OSCAR H. IBARRA†

Abstract. It is shown that the equivalence problem is unsolvable for ε-free nondeterministic generalized sequential machines whose input/output are restricted to unary/binary (binary/unary) alphabets. This strengthens a known result of Griffiths. Applications to some decision problems concerning right-linear grammars and directed graphs are also given.

Key words. unsolvability, equivalence problem, ε-free nondeterministic generalized sequential machines, right-linear grammars, directed graphs

1. Introduction. The equivalence problem for deterministic generalized sequential machines is decidable. (In fact, the equivalence problem is solvable for deterministic sequential transducers [1], [3].) It is also obvious that the equivalence problem for complete nondeterministic generalized sequential machines is decidable. (These are machines which output exactly one symbol per move.) However, the equivalence problem for ε-free (not having the null string ε as output) nondeterministic generalized sequential machines is unsolvable. This result was shown by Griffiths [4] who also observed (as a corollary) that the equivalence problem for c-finite languages [3] is undecidable. In [2], the result was used to show the unsolvability of the equivalence problem for sentential forms of context-free grammars.

In this paper, we strengthen Griffiths's result. Specifically, we show that the equivalence problem for ε-free nondeterministic generalized sequential machines is unsolvable even if we restrict the input/output to unary/binary (respectively, binary/unary) alphabets. This result which is somewhat surprising clearly demonstrates the complexity that nondeterminism can introduce even in very simple computing devices. Applications to some decision problems concerning right-linear grammars and directed graphs are briefly discussed.

The proofs are facilitated by considering a more general type of machine which we now define.

DEFINITION. An *ε-free nondeterministic generalized sequential machine with accepting states* (*EFNGSMA*) over $\Sigma \times \Delta$ is a 6-tuple $M = \langle K, \Sigma, \Delta, \delta, q_0, F\rangle$, where K, Σ, and Δ are finite nonempty sets called the *state set*, *input alphabet*, and *output alphabet*, respectively. δ is a function from $K \times \Sigma$ into the finite subsets of $K \times \Delta^+$,[1] q_0 in K is the *initial state*, and $F \subseteq K$ is a set of *accepting states*.

If $F = K$ (i.e., all states are accepting), M is called simply an *EFNGSM*. In this case, $F(=K)$ is not included in the specification.

The function δ is extended to $K \times \Sigma^+$ as follows: For q in K, x_1, x_2 in Σ^+, $\delta(q, x_1x_2) = \{(p, y_1y_2) |$ for some p', (p', y_1) is in $\delta(q, x_1)$ and (p, y_2) is in $\delta(p', x_2)\}$. For x in Σ^+, let $M(x) = \{y|(p, y)$ is in $\delta(q_0, x)$ for some p in $F\}$. Let $R(M) = \{(x, y)|x$ in Σ^+, y in $M(x)\}$. A relation $R \subseteq \Sigma^+ \times \Delta^+$ is called an EFNGSMA (respectively, EFNGSM) relation over $\Sigma \times \Delta$ if we can find an EFNGSMA (respectively, EFNGSM) M such that $R(M) = R$.

* Received by the editors May 31, 1977, and in revised form January 25, 1978. This research was supported by the National Science Foundation under Grant No. DCR75-17090.

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

[1] Δ^+ denotes the set of all nonnull finite-length strings of symbols in Δ.

For convenience, we will sometimes represent an EFNGSMA $M = \langle K, \Sigma, \Delta, \delta, q_0, F\rangle$ by a directed labeled graph where the nodes represent states and the labeled edges represent transitions. If $\delta(q, a)$ contains (p, y), then there is an edge from node q to node p with label a/y. For example, Fig. 1 shows an EFNGSMA, where $K = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $\Delta = \{0, 1\}$, q_0 is the initial state, and $F = \{q_0, q_2, q_3\}$.

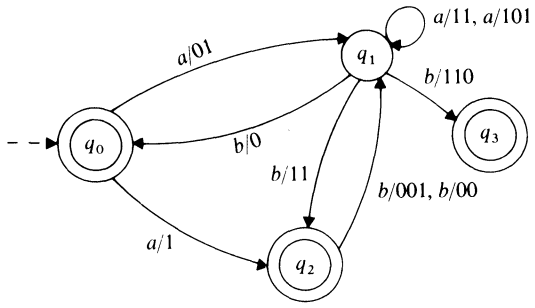


FIG. 1. *An EFNGSMA.*

The equivalence problem for EFNGSMA (respectively, EFNGSM) relations over $\Sigma \times \Delta$ is the problem of deciding for arbitrary EFNGSMA's (respectively, EFNGSM's) M_1 and M_2 over $\Sigma \times \Delta$ whether $R(M_1) = R(M_2)$.

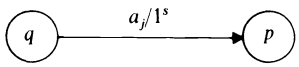
2. Unsolvability of the equivalence problem for EFNGSM relations over $\{0, 1\} \times \{1\}$. First, we prove the following lemma.

LEMMA 1. *The following statements are equivalent:*

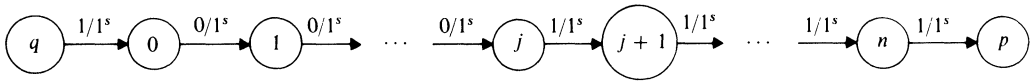
(a) *The equivalence problem for EFNGSMA relations over $\Sigma \times \{1\}$ is solvable for any Σ containing at least 2 elements.*

(b) *The equivalence problem for EFNGSM relations over $\{0, 1\} \times \{1\}$ is solvable.*

Proof. Clearly, (a) implies (b). To prove the converse, consider 2 EFNGSMA's N_1 and N_2 over $\Sigma = \{a_1, \cdots, a_n\}$, $n \geqq 2$ and $\Delta = \{1\}$. By encoding each a_j $(1 \leqq j \leqq n)$ as string $10^j1^{n-j}1$ (of length $n+2$),[2] we can construct 2 new EFNGSMA's M_1 and M_2 such that $R(M_i) = \{(10^{j_1}1^{n-j_1}110^{j_2}1^{n-j_2}1 \cdots 10^{j_k}1^{n-j_k}1,\ 1^{r(n+2)}) | (a_{j_1}a_{j_2} \cdots a_{j_k},\ 1^r)$ is in $R(N_i)\}$, $i = 1, 2$. The construction of M_i from N_i is straightforward. For example, if in N_i there is a transition of the form shown in Fig. 2(a), then the "encoded" sequence of transitions in M_i has the form shown in Fig. 2(b), where the intermediates states numbered $0, 1, \cdots, n$ are new.



(a) *Transition in N_i.*



(b) *Equivalent sequence of transitions in M_i (states $0, 1, \cdots, n$ are new).*

FIG. 2

Clearly, $R(M_1) = R(M_2)$ if and only if $R(N_1) = R(N_2)$. Now let $M_i = \langle K_i, \{0, 1\}, \{1\}, \delta_i, q_{0i}, F_i\rangle$, $i = 1, 2$. We shall construct 2 EFNGSM's M nd M' from M_1 and M_2.

[2] If x is a string, x^j is the string x concatenated with itself j times.

Assume that $K_1 \cap K_2 = \emptyset$, and let $q_0, p_1, \cdots, p_{n+2}$ be new states not in $K_1 \cup K_2$. Let $M = \langle K_1 \cup K_2 \cup \{q_0, p_1, \cdots, p_{n+2}\}, \{0, 1\}, \{1\}, \delta, q_0\rangle$, where δ is defined as follows:

(1) For each a in $\{0, 1\}$, let $\delta(q_0, a) = \delta_1(q_{01}, a) \cup \delta_2(q_{02}, a)$.

(2) For each q in $K_1 \cup K_2$ and a in $\{0, 1\}$, let $\delta(q, a) = \delta_1(q, a) \cup \delta_2(q, a)$.

(3) For $1 \leqq i < n+2$, let $\delta(p_i, 1) = \{(p_{i+1}, 11)\}$.

(4) For each q in F_1, also let $(p_1, 11)$ be in $\delta(q, 1)$.

M' is defined like M except that (4) is replaced by:

(4′) For each q in F_2, also let $(p_1, 11)$ be in $\delta(q, 1)$.

Clearly, $R(M_1) = R(M_2)$ implies $R(M) = R(M')$. Now suppose $R(M) = R(M')$. Consider (x, y) in $R(M_1)$. Then $(x1^{n+2}, y1^{2(n+2)})$ is in $R(M)$ and, hence, also in $R(M')$. But from the construction of M_1, M_2, M, and M' it is clear that the only way $(x1^{n+2}, y1^{2(n+2)})$ can be in $R(M')$ is for (x, y) to be in $R(M_2)$. Hence $R(M_1) \subseteq R(M_2)$. By symmetry, $R(M_2) \subseteq R(M_1)$. Thus, $R(M) = R(M')$ if and only if $R(M_1) = R(M_2)$, and if and only if $R(N_1) = R(N_2)$. It follows that (b) implies (a). □

Notation. For any input alphabet Σ, define the one-state EFNGSMA $M_\Sigma = \langle\{q\}, \Sigma, \{1\}, \delta, q, \{q\}\rangle$, where $\delta(q, a) = \{(q, 1^k) | k = 1, 2, 3\}$ for each a in Σ. Clearly, $R(M_\Sigma) = \{(x, 1^r) |$ for some x_1, x_2, x_3 in Σ^*, $x = x_1x_2x_3 \neq \varepsilon$ and $r = |x_1| + 2|x_2| + 3|x_3|\}$.[3]

THEOREM 1. *It is recursively unsolvable to determine for arbitrary input alphabet Σ and EFNGSMA M over $\Sigma \times \{1\}$ whether $R(M) = R(M_\Sigma)$.*

Proof. The proof involves a reduction of the halting problem for single-tape Turing machines to the problem at hand. We show how we can construct for a given single-tape Turing machine Z an EFNGSMA M over $\Sigma \times \{1\}$ (for some Σ) such that $R(M) = R(M_\Sigma)$ if and only if Z does not halt on an initially blank tape. Since the halting problem for Turing machines is unsolvable [5], the result would follow. The construction of M uses some ideas developed in the proof of Theorem 6.3 of [6].

Let Z be a single-tape Turing machine and K be its set of states. Assume without loss of generality that Z's tape alphabet consists of 0, 1 and b (for blank). We may also assume that Z never overwrites a symbol by a blank. Hence, any configuration of Z can be written as $bxqyb$, where x, y are strings of 0's and 1's, and q is in K. The initial configuration is bq_0b, where we assume that q_0, the initial state, is not a halting state. The EFNGSMA M we shall construct has input alphabet $\Sigma = \{0, 1, b, \#\} \cup K$, where $\#$ is a new symbol.

Let $L_Z = \{x | x = \#ID_1\# \cdots \#ID_k\#,\ k \geqq 2,\ ID_1, \cdots, ID_k$ are configurations of Z, ID_1 is the initial configuration, and ID_k is a halting configuration$\}$. Clearly, L_Z is a regular set and finite automata [7] N_1 and N_2 can be constructed to accept L_Z and $\Sigma^+ - L_Z$, respectively. The EFNGSMA M is constructed from 4 EFNGSMA's $M_1, \cdots, M_4$ such that $R(M) = R(M_1) \cup \cdots \cup R(M_4)$. Since EFNGSMA relations are obviously effectively closed under union, we need only describe the construction of $M_1, \cdots, M_4$.

(1) Let $R_1 = \{(x, 1^r) | (x, 1^r)$ in $R(M_\Sigma)$, x in $\Sigma^+ - L_Z\}$. (See notation above.) Clearly, an EFNGSMA M_1 can be constructed from M_Σ and finite automaton N_2 so that $R(M_1) = R_1$.

(2) M_2 and M_3 are shown in Fig. 3. It is easy to verify that $R(M_2) = \{(x, 1^r) | (x, 1^r)$ in $R(M_\Sigma)$, $r > 2|x|\}$ and $R(M_3) = \{(x, 1^r) | (x, 1^r)$ in $R(M_\Sigma)$, $r < 2|x|\}$.

(3) Now let $R_4 = \{(x, 1^r) | (x, 1^r)$ in $R(M_\Sigma)$, $x = \#ID_1\# \cdots \#ID_k\#$ in L_Z and either $r \neq 2|x|$ or $r = 2|x|$ and for some ID_i, $1 \leqq i < k$, ID_{i+1} is not a proper successor of $ID_i\}$. We shall construct an EFNGSMA M_4 such that $R(M_4) = R_4$. Since the finite

[3] $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$, where ε denotes the null string (i.e., the string of 0 length). $|x|$ denotes the length of x.

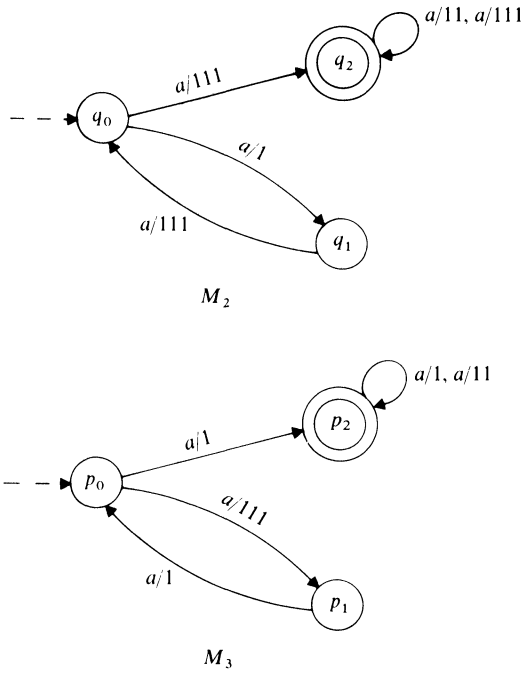


FIG. 3. *a/x represents several transitions, one for each a in* Σ.

automaton N_1 (accepting L_Z) can easily be built into the finite-state control of M_4, we may assume that the inputs to M_4 come from the language L_Z.

M_4 may (nondeterministically) choose to simulate either M_2 or M_3, or perform the following operations on input $x = \#ID_1\# \cdots \# ID_i \# ID_{i+1} \# \cdots \# ID_k \#$ (see Fig. 4): M_4 moves right emitting 2 ones/move until it reaches the # immediately to the left of some ID_i, $1 \leqq i < k$ (ID_i is chosen nondeterministically.)

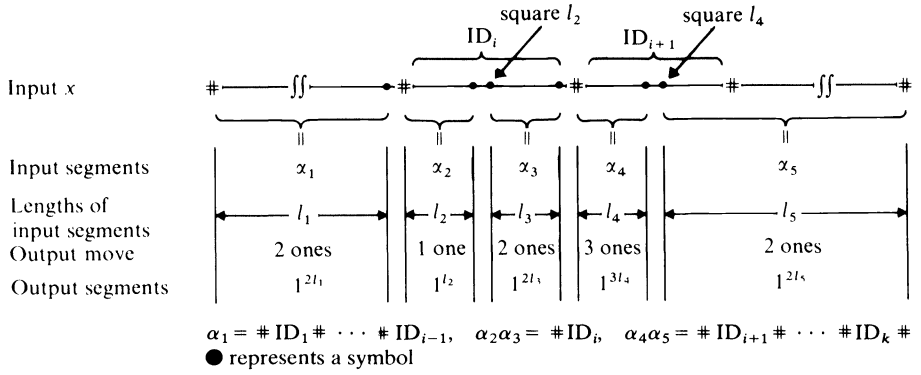


FIG. 4

Then M_4 moves right emitting 1 one/move until it reaches some number $l_2 \geqq 1$ (chosen nondeterministically) of squares to the right of # and guesses that an "error" occurs in position l_2, l_2+1, or l_2+2 of ID_i and ID_{i+1}. M_4 uses its finite-state control to remember these symbols of ID_i as it moves right (of square l_2) emitting 2 ones/move until it reaches the next #. Then M_4 moves right (of #) emitting 3 ones/move. At some point, M_4 guesses that the number l_4 ($\geqq 1$) of squares it has crossed from the last # is equal to l_2. It then moves right (of square l_4) emitting 2 ones/move and checks

whether the symbols at positions l_4, l_4+1, and l_4+2 are appropriate for the successor of ID_i if $l_2=l_4$. If they are appropriate (respectively, not appropriate), M_4 enters a nonaccepting state (respectively, accepting state) and remains in this state emitting 2 ones/move as it advances to the right. The formal construction of M_4 from our description of its operation is straightforward but tedious and is therefore omitted.

Now suppose $(x, 1^r)$ is in $R(M_4)$. Then for some l_1, l_2, l_3, l_4, l_5, $|x|=l_1+l_2+l_3+l_4+l_5$ (see Fig. 4) and $r=2l_1+l_2+2l_3+3l_4+2l_5$. Clearly, $r=2|x|$ if and only if $l_2=l_4$. It follows that $R(M_4)=R_4$.

Let M be an EFNGSMA such that $R(M)=R(M_1)\cup\cdots\cup R(M_4)$. Then $R(M)=R(M_\Sigma)$ if and only if the Turing machine Z does not halt. □

COROLLARY 1. *There is no algorithm P to construct for a given EFNGSMA $M=\langle K, \Sigma, \{1\}, \delta, q_0, F\rangle$ a state-minimal EFNGSMA $M'=\langle K', \Sigma, \{1\}, \delta', q_0', F'\rangle$ such that $R(M)=R(M')$.*

Proof. Suppose an algorithm P exists. Let Z be a single-tape Turing machine and M be the associated EFNGSMA constructed in the proof of Theorem 1. Using P, construct a state-minimal machine M' equivalent to M, i.e., $R(M)=R(M')$. If M' has only 1 state, then $R(M)=R(M')=R(M_\Sigma)$ if and only if M' and M_Σ are identical, and this is trivially decidable. If M' has more than 1 state, then $R(M)=R(M')\neq R(M_\Sigma)$ since M_Σ has only 1 state. Hence, we can decide if $R(M)=R(M')=R(M_\Sigma)$. But from the proof of Theorem 1, $R(M)=R(M_\Sigma)$ if and only if Z does not halt. The result follows. □

From the constructions in Lemma 1 and Theorem 1, we have one of our main results:

THEOREM 2. *Let $\mathcal{G}_1$ be the class of EFNGSM's $M=\langle K, \{0, 1\}, \{1\}, \delta, q_0\rangle$, where δ satisfies the property that for each q in K and a in $\{0, 1\}$, $(p, 1^k)$ in $\delta(q, a)$ implies $k=1, 2, 3$. Then the equivalence problem for $\mathcal{G}_1$ is unsolvable.*

3. Unsolvability of the equivalence problem for EFNGSM relations over $\{1\}\times\{0, 1\}$. We begin by showing that the equivalence problem for EFNGSMA's over $\{1\}\times\Delta$ is unsolvable. We shall present two statements of this result. The first one (Theorem 3) is stronger in that we can restrict one of the machines to be a fixed one-state machine M_Δ. The second statement (Theorem 3') is weaker but the proof is a lot simpler. We give both proofs since they illustrate two different techniques.

Notation. For any output alphabet Δ, define the one-state EFNGSMA $M_\Delta=\langle\{q\}, \{1\}, \Delta, \delta, q, \{q\}\rangle$, where $\delta(q, 1)=\{(q, y)|y \text{ in } \Delta^+, |y|=2, 3, \text{ or } 6\}$. Clearly, $R(M_\Delta)=\{(1^r, y)|$for some integers $r_1, r_2, r_3\geqq 0$, $r=r_1+r_2+r_3\neq 0$, y in Δ^+ and $|y|=2r_1+3r_2+6r_3\}$.

THEOREM 3. *It is recursively unsolvable to determine for arbitrary output alphabet Δ and EFNGSMA M over $\{1\}\times\Delta$ whether $R(M)=R(M_\Delta)$.*

Proof. Let Z be a single-tape Turing machine with state set K and tape alphabet consisting of 0, 1, and b. As before, we assume that the initial state q_0 is not a halting state and Z does not overwrite a symbol by a blank. Let $\Delta=\{0, 1, b, \#\}\cup K$. We shall construct an EFNGSMA M over $\{1\}\times\Delta$ such that $R(M)=R(M_\Delta)$ if and only if Z does not halt on an initially blank tape.

Let h be a homomorphism on Δ^* defined by $h(a)=aaaaaa$ for each a in Δ. Let $Q_Z=\{h(x)|x=\#ID_1\#\cdots\#ID_k\#, k\geqq 2, ID_1, \cdots, ID_k$ are configurations of Z, ID_1 is the initial configuration, and ID_k is a halting configuration$\}$. (Thus, $h(x)$ is just like x except that each symbol is written 6 times.) We can construct finite automata N_1 and N_2 to accept Q_Z and Δ^+-Q_Z, respectively. Now define 4 EFNGSMA's $M_1, \cdots, M_4$ over $\{1\}\times\Delta$ as follows:

(1) M_1 is such that $R(M_1)=\{(1^r, y)|(1^r, y)$ in $R(M_\Delta)$, y in $\Delta^+ - Q_Z\}$. Clearly, M_1 can be constructed from M_Δ and finite automaton N_2.

(2) M_2 and M_3 are shown in Fig. 5. It is easy to check that $R(M_2)=\{(1^r, y)|(1^r, y)$ in $R(M_\Delta)$, $|y|>3r\}$ and $R(M_3)=\{(1^r, y)|(1^r, y)$ in $R(M_\Delta)$, $|y|<3r\}$.

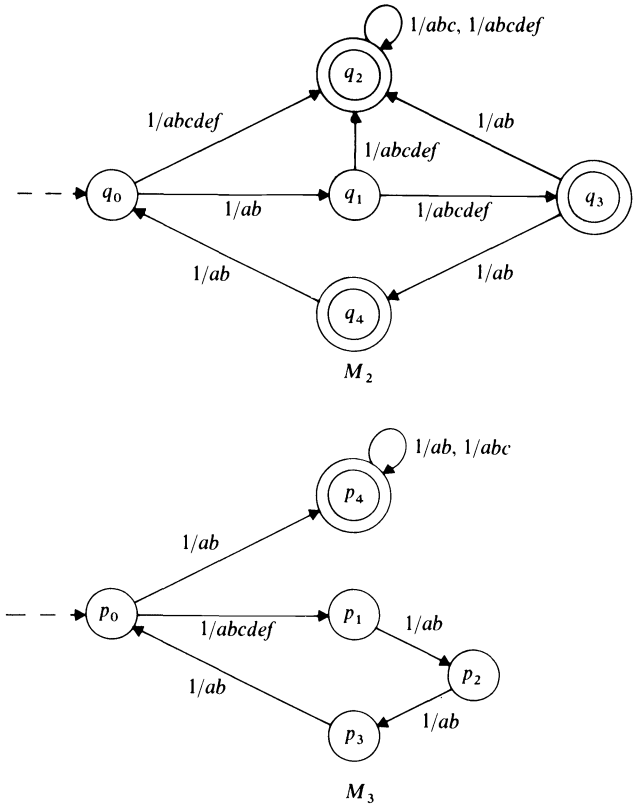


FIG. 5. *a, b, c, d, e, f represent symbols in* Δ. $1/ab$, *e.g., represents several transitions, one for each choice of a and b in* Δ.

(3) Let $R_4=\{(1^r, y)|(1^r, y)$ in $R(M_\Delta)$, $y=h(\#ID_1\#\cdots\#ID_k\#)$ in Q_Z and either $|y|\neq 3r$ or $|y|=3r$ and for some ID_i, $1\leqq i<k$, ID_{i+1} is not a proper successor of $ID_i\}$. We shall construct an EFNGSMA M_4 such that $R(M_4)=R_4$. Since Q_Z is a regular set, we may assume that in any successful computation of M_4, the output string generated is an Q_Z. M_4 may (nondeterministically) choose to simulate either M_2 or M_3, or perform the following operations (see Fig. 6):

Given input 1^r, M_4 nondeterministically decomposes it into 5 segments, $1^r = 1^{2l_1}1^{l_2}1^{2l_3}1^{3l_4}1^{2l_5}$, and generates the different output segments as follows: M_4 generates $h(\alpha_1)$ at the rate of 3 symbols/move while reading the first $2l_1$ ones. Then M_4 scans the next l_2 ones and generates $h(\alpha_2)$ at the rate of 6 symbols/move. The next output segment $h(\alpha_3)$ is generated at the rate of 3 symbols/move while $h(\alpha_4)$ is generated at the rate of 2 symbols/move. Finally, for the last $2l_5$ ones, M_4 generates $h(\alpha_5)$ at the rate of 3 symbols/move. As in the proof of Theorem 1, M_4 has to guess that an "error" occurs after generating $h(\alpha_2)$ and checks this condition after generating $h(\alpha_4)$.

Now $r=2l_1+l_2+2l_3+3l_4+2l_5$ and $|y|=6(l_1+l_2+l_3+l_4+l_5)$. Clearly, $|y|=3r$ if and only if $6l_2=6l_4$. Hence, M_4 can be constructed so that $R(M_4)=R_4$. Construct an

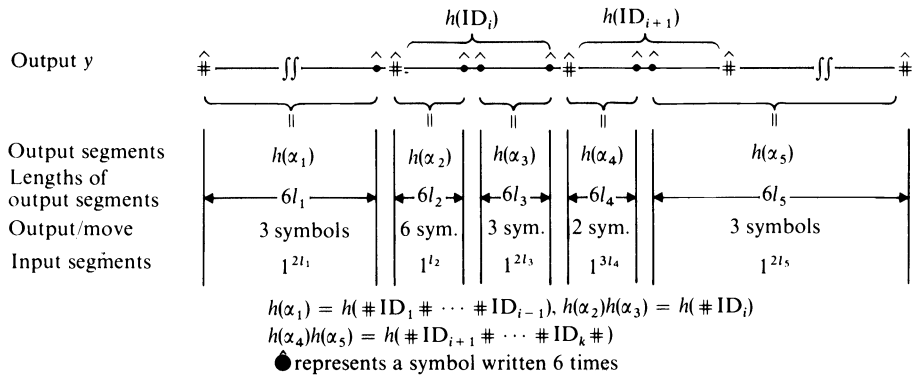


FIG. 6

EFNGSMA M such that $R(M)=R(M_1)\cup\cdots\cup R(M_4)$. Then $R(M)=R(M_\Delta)$ if and only if Z does not halt, completing the proof. □

COROLLARY 2. *There is no algorithm P to construct for a given EFNGSMA $M=\langle K, \{1\}, \Delta, \delta, q_0, F\rangle$ a state-minimal EFNGSMA $M'=\langle K', \{1\}, \Delta, \delta', q_0', F'\rangle$ such that $R(M)=R(M')$.*

Proof. Similar to that of Corollary 1, this time using Theorem 3. □

The next result is a weaker form of Theorem 3. The proof was suggested by the referee.

THEOREM 3′. *The equivalence problem for EFNGSMA's over $\{1\}\times\{0,1\}$ is undecidable even if the machines are restricted so that the output/move is of the form a^k, where a is in $\{0,1\}$ and $k=1,2,3$.*

Proof. The proof relies on Theorem 2. Let N_1 and N_2 be EFNGSMA's over $\{0,1\}\times\{1\}$ whose output/move is of the form 1^k, $k=1,2,3$. Construct from N_i an EFNGSMA M_i over $\{1\}\times\{0,1\}$ by making the substitutions indicated in Fig. 7.

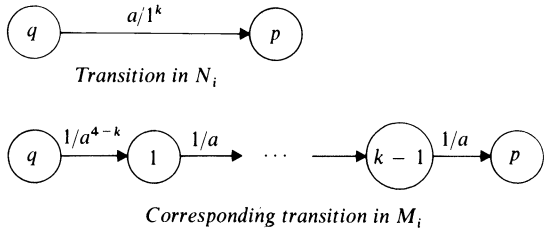


FIG. 7

The accepting states of M_i are the same as those of N_i. Let h be the homomorphism defined by: $h(0)=0^3$ and $h(1)=1^3$. Clearly, $R(M_i)=\{(1^r, h(x))|(x, 1^r)$ in $R(N_i)\}$. Hence, $R(M_1)=R(M_2)$ if and only if $R(N_1)=R(N_2)$. The result now follows from Theorem 2. □

THEOREM 4. *Let $\mathcal{G}_2$ be the class of EFNGSM's $M=\langle K, \{1\}, \{0,1\}, \delta, q_0\rangle$, where δ satisfies the property that for each q in K, (p, y) in $\delta(q, 1)$ implies $y=0^k$ or 1^k, $k=1,2,3$. Then the equivalence problem for $\mathcal{G}_2$ is undecidable.*

Proof. Let $M_i=\langle K_i, \{1\}, \{0,1\}, \delta_i, q_{0i}, F_i\rangle$, $1=1,2$, be the EFNGSMA's constructed in the proof Theorem 3′. Assume that $K_1\cap K_2=\varnothing$. Construct an EFNGSM $M=\langle K_1\cup K_2\cup\{q_0, p_1, p_2, p_3\}, \{1\}, \{0,1\}, \delta, q_0\rangle$, where q_0, p_1, p_2, p_3 are new states and

δ is defined as follows:

(1) $\delta(q_0, 1) = \delta_1(q_{01}, 1) \cup \delta_2(q_{02}, 1)$.

(2) For each q in $K_1 \cup K_2$, let $\delta(q, 1) = \delta_1(q, 1) \cup \delta_2(q, 1) \cup S$, where $S = \{(p_1, 1)\}$ if q is in F_1 and $S = \varnothing$, otherwise.

(3) $\delta(p_1, 1) = \{(p_2, 0)\}$.

(4) $\delta(p_2, 1) = \{(p_3, 1)\}$.

Construct another EFNGSM M' which is defined like M except that in (2), $S = \{(p_1, 1)\}$ if q is in F_2 and $S = \varnothing$, otherwise. It is straightforward to verify that $R(M) = R(M')$ if and only $R(M_1) = R(M_2)$, and the result follows. □

Remark. We have shown that the equivalence problem for EFNGSM's with unary input (respectively, output) alphabet is unsolvable. If we require both the input and output to have unary alphabets then the equivalence problem becomes solvable. In fact, it is decidable to determine for arbitrary (not necessarily ε-free) NGSMA's M_1 and M_2 satisfying $R(M_i) \subseteq w_1^* \cdots w_k^* \times z_1^* \cdots z_m^*$ for some k, $m \geqq 1$, nonnull strings $w_1, \cdots, w_k, z_1, \cdots, z_m$, whether $R(M_1) = R(M_2)$. This follows from the decidability of the equivalence problem for bounded context-free languages [3] and the observation that we can effectively construct linear grammars generating languages $L_i = \{x \# y^R | y \text{ in } M_i(x)\} \cup \{\#\}$, $i = 1, 2$, where $\#$ is a new symbol.[4] (See the next section.)

4. Applications. Let $M = \langle K, \Sigma, \Delta, \delta, q_0 \rangle$ be an EFNGSM. Assume that $K \cap (\Sigma \cup \Delta) = \varnothing$, and let c be a new symbol. We can construct a linear grammar [3] $G = \langle N, \Sigma \cup \Delta \cup \{c\}, P, S \rangle$, where $N = K$ is the set of *nonterminals*, $\Sigma \cup \Delta \cup \{c\}$ is the set of *terminals*, $S = q_0$, and P is the set of rewriting *rules* defined as follows: For each q in K and a in Σ, let $q \to apy^R$ be in P if $\delta(q, a)$ contains (p, y). Also let $q \to c$ be in P for each q in K. Clearly, $L(G) \subseteq \Sigma^* c \Delta^*$ and $L(G)$ has the following property which characterizes *c-finite languages* [3]: (a) for every x in Σ^*, the set $\{y | xcy \text{ in } L(G)\}$ is finite, and (b) for every y in Δ^*, the set $\{x | xcy \text{ in } L(G)\}$ is finite. G is called a *c-finite grammar.* Two c-finite grammars G_1 and G_2 are *equivalent* if $L(G_1) = L(G_2)$.

From Theorems 2 and 4, we have the following refinement of Griffiths's result concerning c-finite languages.

THEOREM 5. *The equivalence problem for c-finite grammars is unsolvable even if the rules are restricted to be of the form $A \to c$ or $A \to aB1^k$, where A and B are nonterminals, a is in $\{0, 1\}$, and $k = 1, 2, 3$. The result also holds for the case when the rules are restricted to be of the form $A \to c$ or $A \to 1Ba^k$, A, B, a and k have the same meaning.*

The next result concerns *right-linear grammars* (*RLG's*). A RLG over Σ is a linear grammar $G = \langle N, \Sigma, P, S \rangle$ where the rules are of the form $A \to xB$ or $A \to x$, A, B in N and x in Σ^+ [5]. (We shall only consider ε-free languages.) If x is in Σ (i.e., x is a single symbol) then G is *normalized.* It is obvious that for every RLG G_1 we can construct a normalized RLG G_2 *equivalent* to G_1, i.e., $L(G_1) = L(G_2)$.

Let y be in Σ^+ and $n \geqq 1$. If $S \overset{*}{\underset{G}{\Rightarrow}} y$ in an n-step derivation, then write $S \overset{n}{\underset{G}{\Rightarrow}} y$. Call 2 RLG's $G_i = \langle N_i, \Sigma, P_i, S_i \rangle$, $i = 1, 2$, *time-equivalent* if for every y in Σ^+ and $n \geqq 1$, $S_1 \overset{n}{\underset{G_1}{\Rightarrow}} y$ if and only if $S_2 \overset{n}{\underset{G_2}{\Rightarrow}} y$. Time-equivalence implies equivalence, but the converse is not true in general. Clearly, time-equivalence of normalized RLG's is decidable. Using Theorem 4, it is easy to show that this is not true for arbitrary RLG's:

[4] y^R is the reverse of string y.

THEOREM 6. *The time-equivalence problem for RLG's over* $\Sigma = \{0, 1\}$ *is undecidable even if the rules are restricted to be of the form* $A \to a^k B$ *or* $A \to a^k$, *where* A *and* B *are nonterminals,* a *is in* $\{0, 1\}$, *and* $k = 1, 2, 3$.

Open Problem: A RLG G is *nonterminal-minimal* if it has the least number of nonterminals among all RLG's equivalent to G. Is there an algorithm to construct for an arbitrary RLG G a nonterminal-minimal RLG G' equivalent to G? (Note that if we restrict our search to a normalized RLG G', such an algorithm exists.)

As a final example, we shall look at a decision problem concerning directed graphs. Let $G = \langle V, E, v_0, f, g\rangle$ be a directed graph, where V is a finite nonempty set of vertices, E is a finite nonempty set of ordered pairs $\langle u, v\rangle$ of distinct vertices called edges, v_0 is a distinguished vertex called the source vertex, and f and g are functions from E into $\{a, b\}$ and $\{1, 2, 3\}$, respectively. Let $R(G) = \{(x, c) | x = a_1 \cdots a_n,\ n \geqq 1$, each a_i in $\{a, b\}$, there exist edges $\langle u_1, u_2\rangle, \cdots, \langle u_n, u_{n+1}\rangle$ such that $u_1 = v_0$, $f(\langle u_i, u_{i+1}\rangle) = a_i$ for $1 \leqq i \leqq n$, and $c = \sum_1^n g(\langle u_i, u_{i+1}\rangle)\}$. Then, we have the following theorem whose proof is straightforward using Theorem 2.

THEOREM 7. *It is recursively unsolvable to determine for arbitrary directed graphs* $G_i = \langle V_i, E_i, v_{0i}, f_i, g_i\rangle$, $i = 1, 2$, *whether* $R(G_1) = R(G_2)$.

Open Problem: Using the notation above, define MAX $(G) = \{(x, \alpha) | \alpha =$ largest c such that (x, c) is in $R(G)\}$. Is there an algorithm to determine for arbitrary G_1 and G_2 whether MAX $(G_1) =$ MAX (G_2)? (We can also define MIN (G) and ask the same question.) Now, let SUM $(G) = \{(x, \alpha) | \alpha$ is the sum of all c's such that (x, c) is in $R(G)\}$. Clearly, we can construct for an arbitrary G a deterministic GSM M_G such that the relation defined by M_G, $R(M_G) = \{(x, 1^\alpha) | (x, \alpha)$ in SUM $(G)\}$. Since the equivalence problem for deterministic GSM's is decidable, it follows that equivalence of SUM (G)'s is decidable.

Acknowledgment. I would like to thank the referee for suggesting the proof of Theorem 3'.

REFERENCES

[1] M. BIRD, *The equivalence problem for deterministic two-tape automata*, J. Comput. System Sci., 7 (1973), pp. 218–236.

[2] M. BLATTNER, *The unsolvability of the equality problem for sentenial forms of context-free grammars*, Ibid., 7 (1973), pp. 463–468.

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

[4] T. V. GRIFFITHS, *The unsolvability of the equivalence problem for ε-free nondeterministic generalized machines*, J. Assoc. Comput. Mach., 15 (1968), pp. 409–413.

[5] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[6] O. H. IBARRA, *Reversal-bounded multicounter machines and their decision problems*, J. Assoc. Comput. Mach., 25 (1978), pp. 116–133.

[7] M. RABIN AND D. SCOTT, *Finite automata and their decision problems*, IBM J. Res. Develop., 3 (1959), pp. 114–125.

SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

0097-5397/78/0704-0013 $01.00/0

A TECHNIQUE FOR EXTENDING RAPID EXACT-MATCH STRING MATCHING TO ARRAYS OF MORE THAN ONE DIMENSION*

THEODORE P. BAKER†

Abstract. A class of algorithms is presented for very rapid on-line detection of occurrences of a fixed set of pattern arrays as embedded subarrays in an input array. By reducing the array problem to a string matching problem in a natural way, it is shown that efficient string matching algorithms may be applied to arrays. This is illustrated by use of the string-matching algorithm of Knuth, Morris and Pratt [7]. Depending on the data structure used for the preprocessed pattern graph, this algorithm may be made to run "real-time" or merely in linear time. Extensions can be made to nonrectangular arrays, multiple arrays of dissimilar sizes, and arrays of more than two dimensions. Possible applications are foreseen to problems such as detection of edges in digital pictures and detection of local conditions in board games.

Key words. array matching, pattern matching, pattern recognition, algorithms, real-time, on-line, subarray, analysis of algorithms, edge detection

Introduction. This paper deals with a conceptual framework that has been found useful for handling the problem of exact-match subarray identification in more than one dimension. (It does not claim to deal with any notion of "topological," "approximate," or "fuzzy" pattern recognition.) In the basic two-dimensional array matching problem, two rectangular arrays are given—a "pattern" and a "subject." The problem is to find all occurrences of the pattern as embedded subarrays of the subject. Such a problem occurs in some methods for detecting edges in digital pictures, where a set of "edge detector" arrays are matched against the picture. It also occurs in the detection of special local conditions in board games, such as "go."

More formally, if P is a u by v rectangular array of elements of alphabet Σ and S is an m by n array of the same type, the problem is to find all pairs (i, j) such that

$$S[i-u+k, j-v+l] = P[k, l]$$

for all k and l such that $1 \leqq k \leqq u$ and $1 \leqq l \leqq v$.

Karp, Miller and Rosenberg [6] have studied problems related to this. In particular, they present a method of finding all repeated occurrences of (all) square subarrays of an n by n square subject array in time $n^2 \cdot \log n$. Unfortunately, their methods are of no direct benefit for the present problem, due to the overhead of finding repeated occurrences of "unwanted" subarrays, and the restriction to square arrays.

The rectangular array matching problem may be viewed as a restricted case of a more general two-dimensional, or "two-level" string matching problem. Consider each row $P[i, 1] \cdots P[i, v]$ of P as a pattern string. Identify each of these rows with a character $\mathscr{S}_i$ in some new alphabet, Σ'. In this way, the whole array P may be viewed as a column of "row/characters," and recognizing P becomes a two-level string-matching process: first, recognize the component rows; second, recognize the column of these rows.

In this new form, it is possible to apply an extended version of the fast string-matching algorithm due to Morris and Pratt [10], yielding a new rapid method for recognition of matching subarrays. This method permits extension to arrays of irregular shape, multiple pattern arrays, and arrays of arbitrarily many

* Received by the editors September 17, 1976, and in final revised form February 13, 1978.

† Department of Computer Science, University of Iowa, Iowa City, Iowa 52242. This work was supported in part by the National Science Foundation under Grant DCR75-06340.

dimensions. Although these extensions exact a price in increased time, storage and algorithmic complexity (primarily during the phase of pattern preprocessing), they are useful and can be more efficient of time overall than more naïve algorithms for the same problems.

In this paper, we first present a general scheme for reducing the array problem to string problems, then a more detailed algorithm for the two-dimensional rectangular case, based on the Morris–Pratt string matching algorithm. Several possible extensions and generalizations are then briefly presented, including multiple pattern arrays, arrays of irregular shape, and "real-time" matching.

The basic array matching algorithm. When the pattern is a rectangular array, we may rely upon a very convenient property. The component rows are all of the same length. It follows that no two component rows may be proper suffixes, one of the other. So, for each position in a row of the subject string, at most one distinct row of the pattern may match in that location. (If two rows match at one location, they must be identical.)

Based on this observation, we present the following general algorithm for two-dimensional array matching.

ALGORITHM A. *General array matching.*

Input: $u \times v$ pattern array P and $m \times n$ subject array S.

Output: A list of all positions (i, j) where P "matches" S.

Phase I. *Preprocess the pattern array P.*

1. Identify the distinct rows of P and assign each a unique index. Let the distinct rows be $X_1, \cdots, X_q$.

2. Represent P by the column $p(1) \cdots p(u)$ in $\{1, \cdots, q\}^*$, such that

$$P = \begin{matrix} X_{p(1)} \\ \vdots \\ X_{p(u)}. \end{matrix}$$

Phase II. *Row matching.* Compute the array

$$Y = \begin{matrix} Y_{1,1} & \cdots & Y_{1,n} \\ \vdots & & \vdots \\ Y_{m,1} & \cdots & Y_{m,n} \end{matrix}$$

$$\text{in } (\{0, \cdots, q\}^*)^*$$

defined by: $Y_{i,j} = k$ if and only if X_k matches a suffix of $S_{i,1} \cdots S_{i,j}$,

$$\text{else } Y_{i,j} = 0.$$

Note that $Y_{i,j}$ is uniquely defined, because of the assumption that no suffix relationships exist between the $X_1, \cdots, X_q$, and that this assumption must be valid if $X_1, \cdots, X_q$ are rows of a rectangular array.

Phase III. *Column Matching.* Compute the array

$$Z = \begin{matrix} Z_{1,1} & \cdots & Z_{1,n} \\ \vdots & & \vdots \\ Z_{m,1} & \cdots & Z_{m,n} \end{matrix}$$

$$\text{in } (\{0, 1\}^*)^*$$

defined by: $Z_{i,j} = 1$ if and only if $p(1) \cdots p(u)$ matches $Y_{i,j} \cdots Y_{m,j}$ at position i, else $Z_{i,j} = 0$.

As an example of the functioning of this algorithm consider

$$P = \begin{array}{cccc} A & A & A = X_1 & p(1) = 1 \\ A & B & A = X_2 & p(2) = 2 \\ A & B & B = X_3 & p(3) = 3 \\ A & A & A = X_1 & p(4) = 1 \end{array}$$

$$S = \begin{array}{cccccccc} A & A & A & B & A & B & B & A \\ A & B & A & B & A & A & A & A \\ A & B & B & A & A & B & A & B \\ A & A & A & B & A & B & B & A \\ A & B & A & B & A & A & A & A \end{array}$$

$$Y = \begin{array}{cccccccc} 0 & 0 & 1 & 0 & 2 & 0 & 3 & 0 \\ 0 & 0 & 2 & 0 & 2 & 0 & 1 & 1 \\ 0 & 0 & 3 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 2 & 0 & 3 & 0 \\ 0 & 0 & 2 & 0 & 2 & 0 & 1 & 1 \end{array}$$

$$Z = \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Two matches are found for the pattern P, at positions (4, 3) and (5, 7).

In effect, we have reduced the two-dimensional matching problem to a collection of single-dimensional string matchings. Considerable work has been done in the efficient matching of strings [1], [2], [3], [4], [7], [8], [9], [10], [11]. In particular, we can apply a fairly simple extension of the Morris–Pratt algorithm [10], [6], [1].

The original M–P algorithm, stated in terms of a single pattern string ω, consists of two phases. In the first, a deterministic finite-state automaton that will accept the set $\Sigma^* \cdot \omega$ is constructed. Constructing this automaton is simplified by allowing it to perform state-transitions without advancing on the subject string ("ε-transitions"). In the second phase, this automaton is "run" on the subject string, a match being reported whenever the automaton advances into an accepting state.

Due to the particular way in which the matching automaton is constructed, it can be shown that for every ε-transition performed there must have been a corresponding non-ε-transition earlier, on which the input was advanced. It thus follows that the second phase of the M–P algorithm runs in time linear with respect to the length of the subject string, independent of the pattern string length. By a clever use of the partially-constructed matching automaton through a kind of "bootstrapping" process in which the pattern string is matched against itself, the initial phase can also be done in time linear with respect to the length of the pattern string.

In order to do multi-dimensional matching efficiently, it is necessary to extend the M–P algorithm to perform simultaneous matching of multiple pattern strings against a single subject string. This is given as an exercise in [1], and can be done in a fairly

straightforward fashion. The state transition diagram for the resulting finite automaton we call the "pattern graph." An example is shown in Fig. 1.

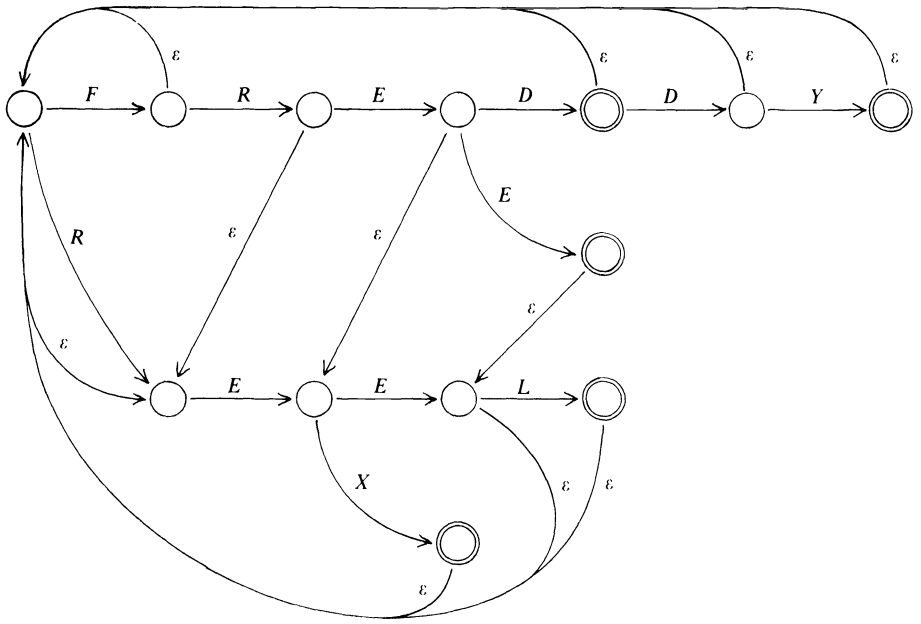


FIG. 1. *Pattern graph for set* {FRED, FREDDY, FREE, REEL, REX}.

The pattern graph for strings $P_1, \cdots, P_u$ may be constructed in time (and storage) bounded above by

$$k \cdot \sum_{i=1}^{u} |P_i|$$

where k is a constant that may depend on $|\Sigma|$ or $\log_2 |\Sigma|$, according to the choice of representation.

In addition to its linear running time, the M–P matching algorithm has another positive feature. It is "on line" in the sense that each position of the input is scanned only once, and that before the next character is read, all matches at the previous position have been found. Taking advantage of this characteristic of the M-P algorithm, our Algorithm A may be rewritten to perform Phases II and III in "parallel" as follows:

ALGORITHM B. *Array matching using M–P approach.*

Input: $m \times n$ pattern array P and $u \times v$ subject array S.

Output: a list of all positions (i, j) where P "matches" S.

Phase I.

1. Identify the distinct rows of P and assign each a unique index.[1] Let the distinct rows be $X_1, \cdots, X_q$.

2. Represent P by the column $p(1) \cdots p(u)$ in $\{1, \cdots, q\}^*$, such that

$$P = \begin{matrix} X_{p(1)} \\ \vdots \\ X_{p(u)}. \end{matrix}$$

[1] Note that the distinct rows may be identified, in linear time, while the pattern graph PAT1 is being built, by use of the partially built pattern graph.

3. Build a pattern graph with starting node PAT1 for $\{X_1, \cdots, X_q\}$.
4. Build a pattern graph with starting node PAT2 for $p(1) \cdots p(u)$.

Phase II. Let $P1$ be the current state of the row-match in the current row. Let $P2[J]$ be the current state of the column-match in the Jth column.

```
for J := 1 to n do P2[J] := PAT2;
for I := 1 to m do
begin
    P1 := PAT1;
    for J := 1 to n do
    begin
        {advance P1 on character S[I, J] as in M–P algorithm}
        C := S[I, J];
        while C-son (P1) = ∅ do P1 := ε-son (P1);
        P1 := C-son (P1);
        if P1 is final for X_t
        then begin
            {advance P2[J] on character t as in M–P algorithm}
            while t-son (P2[J]) = ∅ do P2[J] := ε-son (P2[J]);
            P2[J] := t-son (P2[J]);
            if P2[J] is final for P
            then report match at position (I, J);
            end
        else P2[J] := PAT2;
    end
end
```

The running time of this algorithm is linear in the size of S. Specifically, if S is an m by n matrix and P is a u by v matrix over Σ, with the nodes of the pattern graph represented in memory as vectors of pointers, the running time of matching is bounded above by

$$k_1 \cdot m \cdot n \cdot (\log_2 |\Sigma| + \log_2 u)$$

bit operations, where k_1 is a constant independent of P, S and Σ. In this case, the pattern building time (and hence storage space) will be bounded by

$$k_2 \cdot u \cdot v \cdot (|\Sigma| + \log_2 u),$$

where k_2 is another constant also independent of P, S and Σ. Of course, for "reasonable" values of u, and a register computer, $\log_2 u$ will be less than the register size, so this factor may be ignored in the running times. This is also true of $|\Sigma|$. By use of alternative data structures other time bounds may be obtained which might be preferable in certain applications, especially when Σ is large.

The extension of the Algorithms A and B to N-dimensional arrays is straightforward and is left as an exercise to the interested reader. Another straightforward generalization is to a set of pattern arrays with common row length. (This is the case for "edge detector" matrices in picture processing.) More difficult problems arise, however, if we wish to allow multiple pattern arrays of dissimilar sizes, or of nonrectangular shape. In these cases, more than one pattern row may match at a single position in the subject array, due to possible suffix relations between the pattern rows. If we wish to extend Algorithm A to handle these cases, we are forced to allow the

arrays Y and Z to take on *set* values, representing the set of strings and arrays, respectively, matching at each position of the subject, S. This is not conceptually difficult, but it causes trouble with our implementation B, which uses the M–P algorithm, since the M–P algorithm expects a single character at each position. To get around this, the sets that may appear in Y and Z may be given unique "names," or indices, in a fashion related to the manner in which the rows were given unique indices. Y and Z may then be treated as arrays of characters over larger alphabets. However, a new problem arises with the notion of "matching" which is probably best illustrated by an example.

Example. Consider the generalized arrays P and Q

$$P = \begin{matrix} & & \mathrm{A} = X_1 \\ & \mathrm{A} & \mathrm{A} = X_2, \\ \mathrm{A} & \mathrm{A} & \mathrm{A} = X_3 \end{matrix} \qquad Q = \begin{matrix} \mathrm{A} & \mathrm{A} = X_2 \\ \mathrm{A} & \mathrm{A} = X_2 \end{matrix}.$$

Notice that X_1 is a suffix of X_2 and X_3, and X_2 is a suffix of X_3. The sets that may occur in Y are $\varnothing$, $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$. Assign these indices

$$0, 1, 2, 3.$$

The sets that may appear in Z are $\varnothing$, $\{Q\}$, $\{P, Q\}$, with indices

$$0, 1, 2.$$

With the subject array S:

$$S = \begin{matrix} \mathrm{A} & \mathrm{A} & \mathrm{A} & \mathrm{A} \\ \mathrm{A} & \mathrm{A} & \mathrm{A} & \mathrm{A} \\ \mathrm{A} & \mathrm{A} & \mathrm{A} & \mathrm{A} \\ \mathrm{A} & \mathrm{A} & \mathrm{A} & \mathrm{A} \end{matrix}$$

an extended Algorithm A would define Y and Z:

$$Y = \begin{matrix} 1 & 2 & 3 & 3 \\ 1 & 2 & 3 & 3 \\ 1 & 2 & 3 & 3' \\ 1 & 2 & 3 & 3 \end{matrix} \qquad Z = \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 2 & 2 \\ 0 & 1 & 2 & 2 \end{matrix}$$

The problem is that all of the possible columns

$$\begin{matrix} 1 & 2 & 1 & 2 & 3 \\ 2 & 2 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 & 3 \end{matrix}$$

in Y may "match" $\{P, Q\}$ and that all of the columns

$$\begin{matrix} 1 & 2 & 1 & 2 & 3 \\ 2 & 2 & 3 & 3 & 3 \end{matrix}$$

may "match" $\{Q\}$. If the M–P algorithm is to be used to calculate Z from Y, the set of pattern columns must be expanded to include all of these. This may require a very large column pattern (exponentially large with respect to the number of occurrences of suffix-related rows appearing in the column patterns), and proportionally large preprocessing time, but the running time of the matching phase remains linear. This

extension has been implemented and tested on several examples in comparison runs against the "naïve" algorithm below.

NAÏVE ALGORITHM for generalized array matching. Let $p^{(1)}, \cdots, p^{(s)}$ be (irregular) pattern arrays and S be a subject array, where u is the number of rows in S and $v(i)$ is the length of the ith row.

```
for i := 1 to u do
    for j := 1 to v(i) do
        for t := 1 to S do
            if "p^(t) matches S at position (i, j)"
            then "report match"
```

The results of the comparison showed that:

1) examples of nonrectangular arrays and multiple arrays of dissimilar sizes with limited suffix relations between rows could be handled without memory size problems;
2) for all examples including some which pressed memory limits, total processing time, including pattern preprocessing, was less with the extended version of Algorithm B than it was with the naïve algorithm.

The examples tested were neither "random" nor specifically contrived to benefit one algorithm over the other; they consisted of several occurrences of the patterns arbitrarily placed in a blank space, with some overlapping.

Real-time matching. One advantage of Algorithm B is that it is "on-line." That is, the input is scanned only once, and, after scanning the character at any position of the input, before scanning further, it is possible to answer yes or no to whether any of the patterns match at that position. "Real-time" computability, introduced by Hartmanis and Stearns [5], and recently applied to string matching by Galil [4], is a further refinement of this notion. Intuitively, for an on-line algorithm to operate in real time, the delay between reading the ith input and reporting the appropriate output should be bounded above by a constant. It is possible to achieve this, on a register machine, for array matching by a modification of our techniques, if it is assumed that the number of distinct pattern rows is within the range of numbers that may be operated on in one register-operation.

The Morris–Pratt algorithm is essentially real-time already, except for the ε-transitions of the finite state matching automaton. These may be eliminated in a straightforward manner by specifying the correct state transition for *all* possible inputs, for all states. This is not difficult, and in work with examples we found that many of the "new" transitions added to replace the ε-transitions are either to the starting state or to one of its immediate successors. Recording these "informationless" transitions on the state diagram, or pattern graph, is not necessary. The modifications to the pattern graph for the pattern strings {FREDDY, FRED, FREE, REEL, REX} to allow real-time matching are shown in Fig. 2.

The wiggly arrows are the only new arcs. As can be seen the new graph actually has fewer arcs than the original one, which had ε-arcs.

In making a state transition on symbol C, one follows the arc labeled C, if one exists, from the current node. For any node that has no C-arc from it, one goes to the starting node and advances along the C-arc from it, if one exists. If neither of these approaches succeeds, then one remains at the starting node.

The real-time pattern graph may be constructed similarly to the pattern graph mentioned earlier, working breadth-first in the pattern tree and using the "backward"

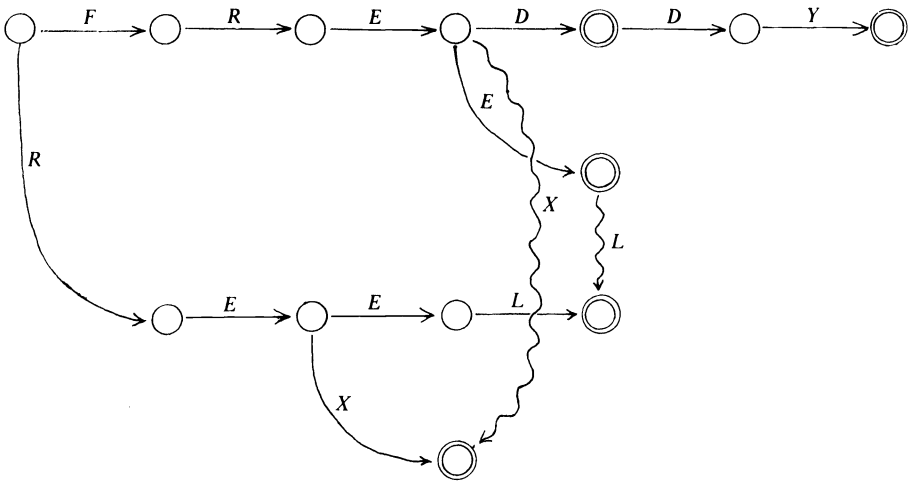


FIG. 2. *"Real time" pattern graph for set* {FREDDY, FRED, FREE, REEL, REX}.

arcs from the preceding level. In some cases, as in the example given, fewer arcs (and hence less storage) are needed for the real-time graph than for the graph with ε-arcs, but it is possible that the number of arcs may be greater by nearly a factor of $|\Sigma|$ (at least in certain pathological cases). However, where real-time operation might be advantageous, Algorithm B may be trivially altered to make use of this alternate form of pattern graph, thereby operating in "real-time."

Conclusions. The algorithms presented in this paper appear practical, at least for cases such as rectangular pattern arrays of identical size, and other cases where occurrences of suffix-related component rows are not numerous enough to cause difficulty. Unfortunately, the range of problems to which they apply appears limited to discrete exact-match situations.

Another approach to string matching, proposed by Weiner [11] and further developed by McCreight [9], is to preprocess the subject string (as opposed to preprocessing the pattern string), permitting the matching time to be reduced to linear in the length of the *pattern* string. This is particularly useful when the subject is likely to remain fixed for a long series of matches, but the patterns are not known in advance. It may be possible to extend this approach to arrays. Straightforward application, however, would appear to require that the set of permissible rows be strictly limited and fixed at the time the subject-array is preprocessed.

Note in revision. Since first writing of this paper, Boyer and Moore have published another interesting string matching algorithm [2], and improvements have been made on the original M–P algorithm [3]. Working from Algorithm A, it appears that these new techniques may be applied to improve on Algorithm B presented here.

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. S. BOYER AND J. S. MOORE, *A fast string searching algorithm*, Comm. ACM, 20 (1977), pp. 762–772.

[3] Z. GALIL AND J. SEIFERAS, *Saving space in fast string matching*, Conference record, IEEE 18th Annual Symposium on Foundations of Computer Science (1977), pp. 179–188.

[4] Z. GALIL, *Real time algorithms for string-matching and palindrome recognition*, Proceedings of the 8th Annual ACM Symposium on Theory of Computing (1976), Assoc. for Comput. Mach., New York, pp. 161–173.

[5] J. HARTMANIS AND R. E. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[6] R. M. KARP, R. E. MILLER AND A. L. ROSENBERG, *Rapid identification of repeated patterns in strings, trees and arrays*, Proceedings of the 4th Annual ACM Symposium on Theory of Computing (1972), Assoc. for Comput. Mach., New York, pp. 125–136.

[7] D. E. KNUTH, J. H. MORRIS AND V. R. PRATT, *Fast pattern matching in strings*, this Journal, 6 (1977), pp. 323–350.

[8] D. E. KNUTH, *The Art of Computer Programming*, vols. I and III, Addison-Wesley, Reading, MA, 1973.

[9] E. M. MCCREIGHT, *A space-economical suffix tree construction algorithm*, J. Assoc. Comput. Mach., 23 (1976), pp. 262–272.

[10] J. H. MORRIS AND V. PRATT, *A linear pattern matching algorithm*, Rep. 40, Computing Center, Univ. of California at Berkeley, 1970.

[11] P. WEINER, *Linear pattern matching algorithms*, Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory (1973), pp. 1–11.