A Fast Selection Algorithm and the Problem of Optimum Distribution of Effort

ZVI GALIL

Tel Aviv University, Tel Aviv, Israel

AND

NIMROD MEGIDDO

Tel Aviv University, Tel Aviv Israel, and Northwestern University, Evanston, Illinois

ABSTRACT. An algorithm is developed which finds the nth largest element of a linearly ordered set S, given in the form of m pairwise disjoint subsets. Each of the m subsets satisfies the property that its kth largest element can be computed in a constant amount of time. The algorithm terminates in time $O(m \cdot \log^2(|S|/m))$. The selection algorithm applies to the problem of optimum distribution of effort, namely, the maximization of the total utility of allocating n persons to m activities, where the utility of k persons assigned to activity j is a concave function $u_j(k)$. Consequently, this problem can be solved in time $O(m \cdot \log^2 n)$.

KEY WORDS AND PHRASES: selection, time complexity, discrete optimization, distribution of effort

CR CATEGORIES: 3.5, 5.25, 5.40

1. *Introduction*

The problem of optimum distribution of effort has been dealt with in the literature of operations research for a long time. Selected references are [3, 4, 9–12, 14]. We shall be dealing with the following version.

Problem 1. Given m concave functions $u_j:[0, n] \to R$ $(j = 1, \dots, m)$, find a nonnegative integral m-vector x which maximizes $\sum_j u_j(x_j)$ subject to $\sum_j x_j \leqq n$.

Solution techniques for this problem which appear in the literature use Kuhn-Tucker conditions or dynamic programming, and usually only particular forms of the u_j are considered. Karush [10] solves the problem for general (not necessarily concave) u_j in time $O(mn^2)$ using a dynamic programming approach.

Our study is motivated by the observation that Problem 1 is reducible to finding the nth largest entry of the matrix $D = (d_{ij})$, where $d_{ij} = u_j(i) - u_j(i-1)$ $(i = 1, \dots, n,$ $j = 1, \dots, m)$. Concavity implies that $d_{ij} \geqq d_{i+1,j}$ $(1 \leqq i \leqq n-1,\ 1 \leqq j \leqq m)$. This is in fact a special case of the so-called selection problem (i.e. finding the nth largest element of a linearly ordered set S). An algorithm which solves the problem in time $O(|S|)$ appears in [2] (see also [1]). This implies an $O(mn)$ algorithm for Problem 1. However, this immediate solution does not take advantage of the monotonicity in the columns of D and the fact that

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Part of the work reported in this paper was done while Z. Galil was visiting at the Department of Mathematics, Cornell University, and was supported by the National Science Foundation under Grant MCS 75-22481. At Tel Aviv University his work was supported in part by the Bat-Sheva Fund.

Authors' present addresses: Z. Galil, Department of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel; N. Megiddo, Graduate School of Management, Northwestern University, Evanston, IL 60201.

© 1979 ACM 0004-5411/79/0100-0058 $00.75

D need not be precomputed. (We assume that the time to compute d_{ij} is independent of i and j.)

Another simple solution for this special case of the selection problem is the following: Let $R_0 = \emptyset$ and $T_0 = \{(1, j): 1 \leqq j \leqq m\}$. After defining R_k and T_k for $0 \leqq k \leqq n$ choose $d_{i,j} = \max_{(r,s) \in T_k} d_{r,s}$ and define $R_{k+1} = R_k \cup \{(i, j)\}$ and $T_{k+1} = (T_k - \{(i, j)\}) \cup \{(i + 1, j)\}$. R_n is the set of indices of the n largest entries of D. (It is easy to see that if $\{(i, j)\} = T_{k+1} - T_k$, then $d_{i,j}$ is the $(k + 1)$-th largest entry in D.) A straightforward implementation of this algorithm yields an $O(mn)$ running time, but using a suitable data structure (any efficient implementation of a priority queue [1]) it can be implemented in time $O(n \log m + m)$. This solution follows from the work of Fox [5], where a more general problem is solved by similar methods. In [6] Fox used a so-called heap to maintain the analogue of T_k and to obtain an $O(n \log m + m)$ running time. Note that the time bound is sublinear in the size of $D(mn)$ because it is not necessary to precompute D. The algorithm inspects only $n + m$ of the mn entries of D, and it computes each such entry when needed. Our algorithms will satisfy a similar property.

In Section 2 we develop a selection algorithm for a set S with a special structure, namely, S is partitioned into m disjoint subsets S_j, $1 \leqq j \leqq m$, and for $1 \leqq k \leqq |S_j|$ the kth largest element of S_j can be computed in a constant amount of time. Our algorithm runs in time $O(m \cdot \log^2(|S|/m))$.[1]

In Section 3 we reduce Problem 1 to the above special selection problem. This reduction is implicit in the work of Fox. As a result we derive an algorithm of time complexity $O(m \cdot \log^2 n)$. Note that unlike the previously mentioned algorithms for Problem 1, this time bound is sublinear in n. Hence, in case n is very large and it is much larger than m this algorithm may be the only tractable technique. Moreover, for practical purposes this algorithm successfully approximates a solution for the continuous version of the problem (relax the integrality constraint and replace n by some real number B) by choosing a sufficiently large n. Another method for this case that involves a systematic search for a suitable Lagrange multiplier appears in [7].

2. *The Selection Algorithm*

We deal with the following particular selection problem.

Problem 2. Given a multiset[2] $S = \{v_j(i): j = 1, \ldots, m, p_j < i \leqq q_j\}$ where $p_j < q_j$, and $v_j: N \rightarrow R$ is monotone nonincreasing for $1 \leqq j \leqq m$, and a number n, $1 \leqq n \leqq |S|$, find an a in S such that there exist $n - 1$ elements of S not smaller than a and all the other elements of S are not larger than a.

We develop below two algorithms that solve Problem 2 in time which is sublinear in the size of S. Denote $s = |S|$ and let $S_j = \{v_j(i): p_j < i \leqq q_j\}$ and $n_j = |S_j| = q_j - P_j$ $(j = 1, \ldots, m)$. (Without loss of generality $n_j \leqq n$ and $s \leqq nm$.) We assume that the time to compute $v_j(i)$ is independent of i and j. Hence it takes a constant amount of time to compute the kth largest element of S_j, namely, $v_j(p_j + k)$.

For the sake of convenience define $v_j(p_j) = +\infty$ and $v_j(q_j + 1) = -\infty$ $(j = 1, \ldots, m)$. Denote by $(l, k) \leftarrow \text{FIND}(a, j)$ the operation which, given a real number a and j $(1 \leqq j \leqq m)$, finds a pair of indices (l, k), $p_j \leqq l < k \leqq q_j + 1$, such that $v_j(l) > a > v_j(k)$ and $v_j(i) = a$ for $l < i < k$. (Note that $a \notin S_j$ if and only if $k - l = 1$.) The cost of $\text{FIND}(a, j)$ is $O(\log n_j)$ since it can be carried out by using binary search (see [1]).

For $a \in S$ let $f_1(a) = \#\{b: b \in S, b > a\}$, $f_2(a) = \#\{b: b \in S, b \geqq a\}$. Note that a is the nth largest element of S if and only if $f_1(a) < n \leqq f_2(a)$. The pair $(f_1(a), f_2(a))$ can be computed as follows:

for $j \leftarrow 1$ **until** m **do** $(l_j, k_j) \leftarrow \text{FIND}(a, j)$;
$f_1(a) \leftarrow \sum_{j=1}^{m} (l_j - p_j)$; $f_2(a) \leftarrow \sum_{j=1}^{m} (k_j - 1 - p_j)$;

Hence, the cost of computing the pair $(f_1(a), f_2(a))$ is $O(\sum_{j=1}^{m} \log n_j)$. Let $P = \{(l, k): 0 < l$

[1] Throughout the paper $\log(x)$ will stand for $\max(\log_2 x, 1)$.
[2] A multiset is a set in which some of the elements may appear more than once.

$< k \leqq s\}$, and consider the following partial order on P: $(l_1, k_1) \leqq (l_2, k_2)$ iff $k_1 \leqq l_2$. Note that if $a \leqq b$ $(a, b \in S)$ then $f_2(a) \leqq f_1(b)$ and therefore the restriction of the partial order to the elements of $\{(f_1(a), f_2(a)): a \in S\} \subseteq P$ is a linear order. Hence, a solution to the selection problem is simply obtained as follows:

SELECT(S, n)
for $j \leftarrow 1$ **until** m **do**
begin
use binary search on $\{(f_1(a), f_2(a)): a \in S_j\}$ to look for $a \in S_j$ such that $f_1(a) < n \leqq f_2(a)$: **if** such an a is found **then return** (a)
end

Obviously, the search in the loop must succeed for at least one j $(1 \leqq j \leqq m)$ since $\bigcup_{a \in S} \{f_1(a) + 1, \ldots, f_2(a)\} = \{1, 2, \ldots, s\}$.

The binary search in the jth execution of the loop requires $\log n_j$ computations of $(f_1(a), f_2(a))$ for the a in S_j. Each computation costs $C \cdot \sum_{j=1}^{m} \log n_j$ and hence the time bound for SELECT1 is $C \cdot (\sum_{j=1}^{m} \log n_j)^2$. Using the fact that the geometric mean is not larger than the arithmetic mean, we get the bound of $O(m^2 \log^2(s/m))$.

It is possible to improve SELECT1 by using the information which is obtained in each execution of the loop to narrow the search in the subsequent executions of the loop. However, this improvement does not yield a better asymptotic upper bound, so we omit the details here.

We describe below another solution to the problem which runs in time $O(m \cdot \log^2 \cdot (s/m))$. The algorithm is inspired by the linear-time algorithm for the selection problem [2]. If $T = \{a_j : i = 1, \ldots, t\}$ where $a_i \leqq a_{i+1}$ $(i = 1, \ldots, t-1)$ then the number $(a_{\lceil t/2 \rceil} + a_{\lfloor t/2 \rfloor + 1})/2$ is called the *median* of T. Note that the number of elements of T which are greater [less] than or equal to its median is at least $t/2$.

Before stating the complete algorithm, we first describe it with the aid of the following general scheme:

1. If the size, s, of the set $(s = \sum_{j=1}^{m} n_j)$ is not larger than m, then find the nth largest element directly by first sorting the set; otherwise, execute steps 2 through 5.

2. Find the median a_j of each S_j and generate a set $T = \{(a_j, j): j = 1, \ldots, m\}$.

3. Sort the elements of T according to their first component. (Thus the set of medians is sorted and the algorithm keeps track of the sorting permutation.) Assume that the sorted T is $((a_{j_1}, j_1), \ldots, (a_{j_m}, j_m))$.

4. Find the unique k $(1 \leqq k \leqq m)$ such that $\sum_{i=1}^{k-1} n_{j_i} < s/2$ and $\sum_{i=1}^{k} n_{j_i} \geqq s/2$, and set $a = a_{j_k}$.

5. Employ $(l_j, k_j) \leftarrow \text{FIND}(a, j)$ $(j = 1, \ldots, m)$ to partition S into the subsets $S^1 = \{b : b \in S, b > a\}$, $S^2 = \{b : b \in S, b = a\}$, and $S^3 = \{b : b \in S, b < a\}$. (Note that $|S^1| = \sum_{j=1}^{m} (l_j - p_j)$ and $|S^1| + |S^2| = \sum_{j=1}^{m} (k_j - 1 - p_j)$.)

6. Depending on the sizes of S^1, S^2, and S^3, continue as follows:
   6(a). If $n \leqq |S^1|$ then look (recursively) for the nth largest element in S^1.
   6(b). If $|S^1| < n \leqq |S^1| + |S^2|$ then a is the nth largest element in S; terminate.
   6(c). If $|S^1| + |S^2| < n$ then look (recursively) for the $(n - |S^1| - |S^2|)$-th largest element in S^3.

We now compute the time complexity $\tau(s)$ of the algorithm. It is obvious that steps 2 and 4 take time $O(m)$, steps 1 and 3 take time $O(m \cdot \log m)$, and step 5 takes time $O(\sum_{j=1}^{m} \log n_j) \leqq O(m \cdot \log(s/m))$. We prove below that $|S^1| + |S^2| \geqq s/4$ and $|S^2| + |S^3| \geqq s/4$. Hence $|S^1| \leqq 3s/4$ and $|S^3| \leqq 3s/4$ and therefore the recursive call (in 6(a) or 6(c)) takes $\tau(3s/4)$ at most. Thus the following inequality holds:

$$\tau(s) \leqq \begin{cases} C \cdot m \cdot \log m & \text{if } s \leqq m; \\ C \cdot m \cdot \log s + \tau(3s/4) & \text{otherwise.} \end{cases}$$

Hence $\tau(s) \leqq C \cdot m \cdot \log s \cdot \log(s/m)$. To prove that $|S^1| + |S^2| \geqq s/4$, consider Figure 1.

The number of elements of S in the upper left [lower right] box in Figure 1 is at least $\sum_{i=1}^{k} n_{j_i}/2 [\sum_{i=k}^{m} n_{j_i}/2]$, which is at least $s/4$, by the choice of k. However, all these elements

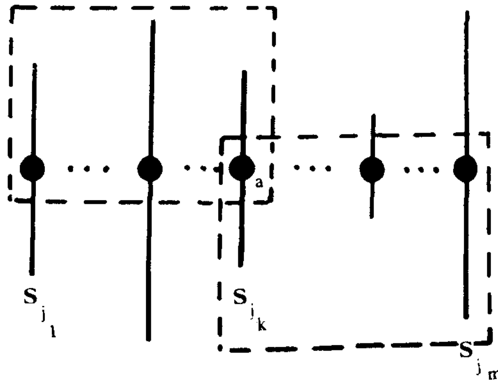

FIG. 1.

belong to $S^2 \cup S^3$ $[S^1 \cup S^2]$ since each such element is not larger [smaller] than the median of the corresponding subset, which is in turn not larger [smaller] than a. Hence, the claim is proved.

A detailed algorithm SELECT2(n, S_{PQ}) (where $P = \{p_1, \dots, p_m\}$ and $Q = \{q_1, \dots, q_m\}$ define the set S) is given below. The numerals at the left refer to the corresponding steps of the scheme described above.

```
        SELECT2(n, S_PQ)
          begin
            s ← Σ_{j=1}^{m} (q_j − p_j)
1           if s ≦ m then do
              begin
                SORT(S)
                return (nth element of S)
              end
            else do
              begin
2               T ← ∅
                for j ← 1 until m do
                  begin
                    x ← (p_j + q_j + 1)/2
                    b ← (v_j([x]) + v_j([x] + 1))/2
                    T ← T ∪ {(b, j)}
                  end
3               SORT(T)
4               sum ← 0; i ← 0
                while sum < s/2 do
                  begin i ← i + 1; sum ← sum + (q_{j_i} − p_{j_i}) end
                a ← a_{j_i}
                lsum ← 0; ksum ← 0
5               for j ← 1 until m do
                  begin
                    (l_j, k_j) ← FIND(a, j)
                    lsum ← lsum + (l_j − p_j)
                    ksum ← ksum + (k_j − 1 − p_j)
                  end
6               if n ≦ lsum then do
6(a)              begin
                    for j ← 1 until m do q'_j ← l_j
                    return (SELECT2(n, S_PQ'))
                  end
                else do
6(b)              if n ≦ ksum then return (a)
6(c)                else do
                      begin
                        for j ← 1 until m do p'_j ← k_j − 1
```

```
                return (SELECT2(n − ksum, S_{P'Q}))
            end
    end
end SELECT2
```

Note that we use recursion only for the ease of presentation. We can easily replace the recursion by an iterative process which modifies p and q as in 6(a) and 6(c) and goes back to the beginning as long as $s > m$.

As was shown above, SELECT2 runs in time bounded by $C \cdot m \cdot \log s \cdot \log(s/m)$ and if $s > m^2$ ($n > m$ in case $s = mn$), then the run time is bounded by $2C \cdot m \cdot \log^2(s/m)$. It is possible to change SELECT2 so that this bound holds also for small values of s: The results of steps 1 and 4 can be achieved without sorting and in time $O(m)$. In step 1 the nth largest element is found directly by using the linear-time selection algorithm of [2] which we denote by SELECT0. In steps 3 and 4 $a = a_{j_k}$ is found by the following binary search on the set T, which we denote by $BS(T, s/2)$: Using SELECT0 we find the median b of the set of first components of elements of T. We partition T into two sets T_1 and T_2 in such a way that for every $(a_i, i) \in T_1$, $a_i \leqq b$, and for every $(a_i, i) \in T_2$, $a_i \geqq b$ and $|T_1| = [|T|/2]$. (This is possible by appropriately partitioning the pairs (a_i, i) with $a_i = b$.) Let $p = \sum_{i \in T_1} n_{j_i}$. If $p \geqq s/2$ we repeat the same with T_1 by calling $BS(T_1, s/2)$; otherwise we repeat the same with T_2 by calling $BS(T_2, s/2 - p)$. So each time we deal with a set that is half the size, and the time it takes to find a is bounded by $Cm + Cm/2 + Cm/4 + \cdots \leqq 2Cm$. As a result, if we denote by $\tau'(s)$ the time complexity of the modified algorithm, the following inequality holds:

$$\tau'(s) \leqq \begin{cases} Cm & \text{if } s \leqq m; \\ Cm \log(s/m) + \tau'(3s/4) & \text{otherwise.} \end{cases}$$

Hence, $\tau'(s) \leqq C \cdot m \cdot \log^2(s/m)$.

We did not include this modification in the detailed algorithm because of several reasons. First, the modified algorithm is much more involved and it improves the algorithm only for the case $s \ll m^2$. Our main motivation for constructing SELECT2 was Problem 1 and in particular the case $n \gg m$ (i.e. $s = mn \gg m^2$). Moreover, if $s = mn \leqq m^2$ the solution due to Fox takes time bounded by $C(m + (s/m)\log m) \leqq C \cdot m \cdot \log^2(s/m)$.

3. *Optimum Distribution of Effort*

It can be easily verified that there is actually no loss of generality in assuming that u_j ($j = 1, \ldots, m$) (see Problem 1) are monotone nondecreasing. Indeed, if x solves Problem 1, then $u_j(x_j) \geqq u_j(x_j - 1) \geqq \cdots \geqq u_j(0)$ ($j = 1, \ldots, m$).

A nonnegative integral m-vector x such that $\sum_j x_j \leqq n$ is called *feasible*; if x also solves Problem 1 then it is called *optimal*. Optimality is characterized as follows.

PROPOSITION 1. *A feasible x is optimal if and only if* $\max\{u_j(x_j + 1) - u_j(x_j): x_j < n\} \leqq \min\{u_j(x_j) - u_j(x_j - 1): x_j > 0\}$.

Proposition 1 is due to Gross [8]. Its proof can also be found in [13]. In terms of the d_{ij} (Section 1) optimality is characterized as follows:

PROPOSITION 2. *Let λ be the nth largest entry of D. A feasible x is optimal if and only if* $\max\{d_{ij}: x_j < i \leqq n\} \leqq \lambda \leqq \min\{d_{ij}: 1 \leqq i \leqq x_j\}$.

Proposition 2 follows immediately from Proposition 1. It is also implicit in [5].

The following algorithm is based on Proposition 2.

ALGORITHM

1. Find the nth largest entry λ of the matrix D.
2. In each column j of D compute $(l_j, k_j) \leftarrow \text{FIND}(\lambda, j)$.
3. Let $l = \sum_{j=1}^{m} l_j$ and $\Delta_r = \sum_{j=1}^{r}(k_j - l_j - 1)$, $r = 1, \ldots, m$. (Note that $l < n \leqq l + \Delta_m$.)
4. Find the smallest r ($1 \leqq r \leqq m$) such that $l + \Delta_r \geqq n$ and let $x_j = k_j - 1$ for $j < r$; $x_j = (k_r - 1) - (l + \Delta_r - n)$ for $j = r$; $x_j = l_j$ for $j > r$.

The time bounds for steps 2, 3, and 4 are $O(m \cdot \log n)$, $O(m)$, $O(m)$, respectively, and

hence are dominated by the time bound for step 1, which is $O(m \cdot \log^2 n)$. In case the u_j are not monotone, then if $\lambda < 0$ all we have to do is to replace λ in the Algorithm by 0.

4. *Conclusion*

We constructed two algorithms for solving Problem 2. We used them to obtain new algorithms for Problem 1; but we feel that Problem 2 is important by itself. The simple algorithm due to Fox described in the Introduction takes $O(n \log m + m)$ time. Note that the time bounds of SELECT1 and SELECT2 do not depend on n. So in case n is small, Fox's algorithm is better. But in general n can be as large as s. (For example, finding the median of S by Fox's algorithm will take $O(s \log m)$ time.)

Let SELECT0′ be the straightforward algorithm for the general selection problem that first sorts the set and takes time $C \cdot n \log n$. Recall that SELECT0 is the $O(n)$ time algorithm of [2] for the selection problem. There is an analogy between the new algorithms SELECT1 and SELECT2, and the pair of algorithms SELECT0′ and SELECT0. SELECT1 [SELECT0′] is simpler and SELECT2 [SELECT0] is asymptotically faster. But while the asymptotic bound for SELECT0 is better than that of SELECT0′ by a factor of $\log n$, the bound for SELECT2 is better than that of SELECT1 by a factor of m, and m can be proportional to the size of the problem. So by using similar ideas to those used in SELECT0 we were able to derive for the special case of Problem 2 a better improvement than the one obtained for the general selection problem.

Finally, we compare the new algorithm for Problem 1 with Fox's algorithm. The former takes $O(m \cdot \log^2 n)$ time and the latter takes $O(n \cdot \log m + m)$ time. Hence our algorithm is better if n is very large and much larger than m. More precisely, the point where the asymptotic times are the same is when $n/(\log n)^2 \approx m/\log m$. Near this point Fox's algorithm is better because the constant factor in the bound for our algorithm is larger than the one for Fox's algorithm.

In addition Fox's algorithm has two advantages:

(1) Our algorithm assumes that $d_{i,j}$ can be computed in time independent of i and j, while for Fox's algorithm it suffices that given $d_{i-1,j}$ it is possible to compute $d_{i,j}$ in time independent of i and j. Application I in [5] is an example when the latter condition, but not the former, is satisfied.

(2) Fox's algorithm finds the n largest elements of S one by one in decreasing order. So if after finding the nth largest element of S we want to find the kth element of S for $k < n$, then no extra work is needed if Fox's algorithm is used. This is not the case if SELECT2 is used. If, however, $k > n$, then both algorithms can exploit effectively the information obtained while finding the nth largest element of S.

Postscript

Don Johnson has pointed out to us that it is possible to modify the improved version of SELECT2 outlined at the end of Section 2 to get an $O(m \cdot \log(|S|/m))$ algorithm for Problem 2 [an $O(m \log n)$ algorithm for Problem 1]. The modification is based on ideas attributed to Jefferson, Shamos, and Tarjan (see Theorem 3.4 in M.I. Shamos, Geometry and statistics: Problems at the interface, in *Algorithms and Complexity: New Directions and Recent Results*, J.F. Traub, Ed., Academic Press, New York, 1976, pp. 251–280).

This new improved version of SELECT2 uses heavily SELECT0, the recursive linear time selection algorithm, and consequently it has a much larger constant than SELECT2. Therefore, SELECT2 may be better in many practical cases.

Theoretically, the $O(m \log n)$ solution for Problem 1 can be combined with Fox's $O(n \log m + m)$ to yield an $O(\min(n, m)\log(\max(n, m)) + m)$ algorithm.

ACKNOWLEDGMENTS. We are indebted to A. Tamir for his critical reading of several versions of the paper, and for letting us know that we have actually rediscovered

Proposition 1. We would like also to thank one of the referees for pointing out to us the works by Fox.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. BLUM, M., FLOYD, R.W., PRATT, V.R., RIVEST, R.L., AND TARJAN, R.E. Time bounds for selection. *J. Comptr. Syst. Sci. 7* (1972), 448-461.
3. CHARNES, A., AND COOPER, W.W. The theory of search: Optimal distribution of effort. *Manage. Sci. 5* (1958), 44-49.
4. DE GUENNI, J. Optimum distribution of effort: An extension of the Koopman basic theory. *J. ORSA* 9 (1961), 1-7.
5. FOX, B.L. Discrete optimization via marginal analysis. *Manage. Sci. 13* (1966), 210-216.
6. FOX, B.L. Accelerating list processing in discrete programming. *J. ACM 17* (1970), 383-384.
7. FOX, B.L., AND LANDI, D.M. Searching for the multiplier in one constraint optimization problems. *Oper. Res. 18* (1970), 253-262.
8. GROSS, O. Class of discrete type minimization problems. RM-1644, Rand Corp., Santa Monica, Calif., Feb. 1956.
9. KARUSH, W. A queuing model for an inventory problem. *J. ORSA 5* (1957), 693-703.
10. KARUSH, W. A general algorithm for the optimal distribution of effort. *Manage. Sci.* 9 (1962), 50-72.
11. KOOPMAN, B.O. The optimum distribution of effort. *J. ORSA 1* (1953) 52-63.
12. LUSS, H., AND GUPTA, S.K. Allocation of effort resources among competitive activities. *Oper. Res. 23* (1975), 360-366.
13. SAATY, T.L. *Optimization in Integers and Related Extremal Problems.* McGraw-Hill, New York, p. 184.
14. WILKINSON, C., AND GUPTA, S.K. Allocating promotional effort to competing activities: A dynamic programming approach. International Federation of Operations Research Socs. Conf., Venice, 1969, pp. 419-432.

RECEIVED MARCH 1977; REVISED MARCH 1978