SIAM J. COMPUT.
Vol. 10, No. 2, May 1981

0097-5397/81/1002-0008 $01.00/0

AN $O(n \log^2 n)$ ALGORITHM FOR THE kth LONGEST PATH IN A TREE WITH APPLICATIONS TO LOCATION PROBLEMS*

N. MEGIDDO, A. TAMIR, E. ZEMEL AND R. CHANDRASEKARAN†

Abstract. Many known algorithms are based on selection in a set whose cardinality is superlinear in terms of the input length. It is desirable in these cases to have selection algorithms that run in sublinear time in terms of the cardinality of the set. This paper presents a successful development in this direction. The methods developed here are applied to improve the previously known upper bounds for the time complexity of various location problems.

Key words. polynomial algorithm, selection, location theory, tree, p-center

1. Introduction. It is now well known that the kth largest element of an ordered set S can be found in linear time in the cardinality of S [1]. Since the discovery of that fact, it has been observed by several authors that in some structured sets, the kth largest element may be found even faster. For example, if $S = X + Y$ (where both X and Y consist of n numbers) then the kth largest element of S can be found in $O(n \log n)$ time, even though $|S| = n^2$. This was first achieved by Jefferson, Shamos and Tarjan [15] and by Johnson and Mizoguchi [11], and later generalized and improved by Frederickson and Johnson [6]. A more general case is the following. Suppose that the set S is partitioned into m sorted subsets such that the kth largest element in each subset can be found in constant time. Fox [5] finds the kth largest element of S in $O(m + k \log m)$ time. Galil and Megiddo [7] solve the problem in $O(m \log^2 (|S|/m))$ time. The basic idea of [15] can be used to solve this problem in $O(m \log (|S|/m))$ steps. This was improved by Frederickson and Johnson [6], who solve the same problem in $O(\max \{m, c \log (k/c)\}$ time, where $c = \min (k, m)$. This is also proved to be an asymptotically optimal bound [6], [10].

The structure of S in this latter example is quite abstract. It remains an open question how other structured sets should be handled. For example, suppose that S is the set of all pairs of nodes of a graph, ordered according to the distance (along a shortest path) between the members of the pair. How can we exploit this structure on S for finding the kth largest element? Another interesting example is when S is the set of maximum flows between pairs of source-sinks in a capacitated network.

In this paper we develop an algorithm for the kth largest element in the set of all simple paths in a tree with edge-lengths. The cardinality of this set is $O(n^2)$ (n is the number of nodes in the tree and each simple path is characterized by its two endpoints). However, our algorithm runs in $O(n \log^2 n)$ time. This fast method of selecting an internodal distance is shown to be very useful in the solution of different combinatorial location problems.

The organization of the paper is as follows. In § 2 we review the two basic approaches to selection in an ordered set with sorted subsets. In § 3 we discuss a decomposition scheme for trees, on which the partition of the set of paths is based. The partitioning itself is developed in § 4 and the solution of the selection problem in the set of paths is summarized in § 5. A brief survey of four different location problems is given in § 6. In § 7 we apply the methods developed in this paper to obtain improved algorithms for the location problems defined in § 6. In § 8 we briefly discuss the more general case of weighted location problems.

* Received by the editors May 5, 1979, and in revised form April 21, 1980.
† Northwestern University, Graduate School of Management, Evanston, Illinois 60201.

2. An overview of selection algorithms. Suppose that an ordered set S is partitioned into m subsets $S_1, S_2, \cdots, S_m$ such that the kth largest element in each subset can be found in constant time. We distinguish between two methods of selection in S. The first one, which we like to call "trimming," is attributed to Jefferson, Shamos and Tarjan [15] and is also used in Frederickson and Johnson [6]. The second, which we call "splitting," is a generalization of linear-time median finding [1] and is used in Johnson and Mizoguchi [11] and in Galil and Megiddo [7]. For simplicity of exposition, we assume in this section that all members of S are distinct. Handling the general case of S being a multiset is similar (see the above references). At a given iteration, let $S'_i \subseteq S_i$ be the set of elements still under consideration with $S' = \bigcup_{i=1}^m S'_i$.

A. *Trimming.* Suppose that we are looking for the kth largest element x of S', and assume without loss of generality that $k \leqq \frac{1}{2}|S'|$. We first find the lower quartile, y_i, in each S'_i. Next, we consider the set $Y = \{y_1, \cdots, y_m\}$ where each y_i is weighted by $|S'_i|$, and we find the (weighted) lower quartile y of Y. Obviously, at least one half of the elements of S' are greater than y, and hence $y \leqq x$. We can now reduce the set S' by discarding the lower quarter of each subset S'_i for which $y_i \leqq y$. This amounts to discarding $\frac{1}{16}$ of the set S', and the problem now reduces to finding the kth largest element of the remaining set.

B. *Splitting.* In this method we first find the median z_i in each S'_i. Next, we find the weighted median z of $Z = \{z_1, \cdots, z_m\}$ (relative to the weights $|S'_i|$). Obviously, z is between the lower and upper quartiles of S'. By computing the rank of z in S', we can tell whether $z \geqq x$ or $z < x$. In the latter case, the lower half of every S'_i such that $z_i \leqq z$ can be discarded, and we look for the kth largest element in the remaining set. Otherwise, the upper half of every S'_i such that $z_i \geqq z$ is discarded, and we look for the $(k - \frac{1}{4}|S'|)$th largest element of the remaining set.

It is interesting to compare the logic and overall efficiency of splitting and trimming. One difference between the two methods lies in the position of elements they eliminate. At any given iteration, splitting may eliminate elements from the upper or lower quartiles of S', depending on the outcome of a logical test. In contrast, the elimination process of trimming is not based on any test, and the elements eliminated always come from lower parts of S' if $k \leqq \frac{1}{2}|S'|$ and from its upper part if the reverse condition holds. As will be pointed out in § 7, a procedure similar to splitting (i.e., based on a logical test) turns out to be preferable for solving various location problems on a tree. As for the efficiency of identifying the kth element of S, we note that, in the worst case, splitting eliminates at each iteration more variables than trimming (four times as many in the formulation given above, although the difference can be reduced by a slight modification of the trimming procedure). However, the corresponding reduction in the number of iterations enjoyed by the splitting method is more than offset by the effort involved in identifying the rank of z in S' which is necessary to support the logical test. Thus, while the overall complexity achieved by the splitting procedure is $O(m \log^2 (|S|/m))$ the corresponding complexity for trimming is only $O(m \log (|S|/m))$.

Our algorithm for the kth longest simple path in a tree exploits the structure of the set S of paths in the following way. We partition S into $m = O(n \log n)$ subsets $S_1, \cdots, S_m$ with the following properties.

(i) The kth largest element in any S_i, as well as its length, can be computed in constant time.

(ii) All the elements of each S_i are paths leading from the same node v_i to other nodes of the tree.

(iii) The k largest and the k smallest elements of any S_i can be discarded in constant time.

(iv) The partitioning process is carried out in $O(n \log^2 n)$ time and $O(n \log n)$ space.

Once this partition is obtained, one can employ the trimming algorithm for finding the kth longest path. This amounts to a time bound of $O(n \log^2 n)$. Details are worked out in the following sections. The partitioning is carried out by a divide-and-conquer algorithm on the tree T. The first step in this direction is an efficient decomposition of the tree which we describe in the next section.

3. Decomposition of trees. In this section we show how to decompose a tree T into three (or fewer) subtrees such that precisely one node of T belongs to more than one subtree and such that each subtree has no more than $n/2+1$ nodes (where the set of nodes of T is $N=\{1,\cdots,n\}$).

Suppose that the tree T is given in the form of lists $N(i)$ of all the neighbors of a node i ($i=1,\cdots,n$). If i and j are neighbors, then by removing the edge (i,j) two subtrees of T are induced. We denote by $K(i,j)$ the number of nodes in that subtree which contain node i. (Note that K is defined on *ordered* pairs of neighboring nodes.) It is easy to verify the following:

(1) If i is a leaf where $N(i)=\{j\}$ then $K(i,j)=1$.

(2) $K(i,j)+K(j,i)=n$ for all pairs of neighbors.

(3) For all j, $\sum_{i\in N(j)} K(i,j)=n-1$.

(4) If $j,k\in N(i)$ $(j\neq k)$ then $K(j,i)<K(i,k)$.

In order to decompose T in the manner described above, we need to find a node x such that for all $i\in N(x)$, $K(i,x)\leqq n/2$. The existence of such a node, referred to as the *centroid* of T, was observed by Jordan in 1869 [12]. Linear time algorithms for finding the centroid appear in Goldman [8] and Kariv and Hakimi [13]. For the sake of completeness we provide such an algorithm below.

We first note that the computation of all the $K(i,j)$s can be carried out in $O(n)$ time. This is done as follows. Fix one of the nodes r as the "root," so that every other node i has a "father" $f(i)$ relative to r (i.e., $f(i)$ is the node following i on the path from i to r).

The quantities $K(i,f(i))$ $(i\neq r)$ can be computed recursively by $K(i,f(i))=1+\sum_{j:f(j)=i} K(j,f(j))$, and the computation of all $K(i,j)$s can be completed by (2). The whole process takes $O(n)$ time.

Once all the $K(i,j)$s are known, the following process can be used to find a node x such that $K(i,x)\leqq n/2$ for all $i\in N(x)$.

$x\leftarrow 1$

(1) **if** $K(i,x)\leqq n/2$ for all $i\in N(x)$ **then stop**
else (there is precisely one $i\in N(x)$ such that $K(i,x)>n/2$) $x\leftarrow i$
go to 1

This procedure generates a path $1=x_1,\cdots,x_k=x$ such that $K(x_{j+1},x_j)>n/2$ $(j=1,\cdots,k-1)$. By (4) and (2), the function $m(x_j)\equiv \max_{i\in N(x_j)} K(i,x_j)$ is monotone decreasing along that path, and hence an $x_k=x$ is reached for which $m(x)\leqq n/2$. Obviously, this procedure takes $O(n)$ time.

We now claim that the set $N(x)$ can be partitioned into three or fewer subsets N_1, N_2, N_3 such that $\sum_{i\in N_i} K(i,x)\leqq n/2$. This is easily proved as follows. Assume $N(x)=$

$\{v_1, \cdots, v_p\}$. By (3) there is s $(1 \leqq s \leqq p)$ such that

$$\sum_{i=1}^{s-1} K(v_i, x) \leqq \frac{n-1}{2} \quad \text{and} \quad \sum_{i=s+1}^{p} K(v_i, x) \leqq \frac{n-1}{2}.$$

The desired subsets are $N_1 = \{v_1, \cdots, v_{s-1}\}$, $N_2 = \{v_s\}$, $N_3 = \{v_{s+1}, \cdots, v_p\}$.

Finally, the partition of $N(x)$ induces a decomposition of T into three or fewer subtrees T_1, T_2, T_3; namely, T_j is the subtree consisting of x and all the nodes accessible from x via a member of N_j $(j = 1, 2, 3)$. Obviously, x is the only node of T that belongs to more than one such subtree, and also in each T_j there are no more than $(n/2)+1$ nodes. The decomposition is carried out in $O(n)$ time.

4. Partition of the set of paths in a tree. In the preceding section we described a decomposition of a tree into three subtrees with a single node x common to the three of them. We refer in this section to that node x as the "decomposer." In this section, S is the set of all simple paths in a tree T. Since there is a one-to-one correspondence between pairs of nodes and simple paths in a tree, we also consider S as the set of pairs of nodes, ordered according to the internodal distances. We partition S into subsets such that the kth largest element in any subset can be found in constant time.

The essence of the partitioning algorithm is as follows. First, we find a decomposer x (see § 3) and we look at the three subtrees T_1, T_2, T_3 in the corresponding decomposition. For each T_i $(i = 1, 2, 3)$, we compute all the distances from the node x to all other nodes of T_i, and we sort the set S_i of all simple paths leading from x into T_i according to these distances. Thus, the node x contributes three sorted subsets to our partition of S. Next, for each node $j \neq x$ in T_1 we can easily compute the sorted set of distances from j to all nodes of T_2, since this is obtained by adding a constant (namely, the distance between j and x) to all elements of S_2. Analogously, for each $j \neq x$ in either T_1 or T_2, we compute the sorted set of distances from j to all nodes of T_3, by adding the distance between j and x to all elements of S_3. Thus, each node $j \neq x$ of T_1 contributes at this stage two sorted subsets and each $j \neq x$ in T_2 contributes one sorted subset to our partition of S. We proceed by decomposing the subtrees T_1, T_2, T_3, each along the same lines described above, until all the paths (or equivalently, pairs) are enumerated. Throughout this process, we skip paths leading to or from nodes that have previously served as decomposers, to make sure that each pair of nodes is taken into account precisely once.

The number of subsets created during the partitioning process is estimated as follows. Let $M(n)$ denote the maximum number of subsets in such a partition of S for a tree with n vertices. The tree is decomposed into three subtrees. If n_1, n_2, n_3 are the numbers of nodes in these subtrees, then $n_1+n_2+n_3 = n+2$ and $n_i \leqq n/2+1$. Each node contributes no more than three subsets to the partition of S, and we proceed, recursively, with the subtrees. Hence

$$M(n) \leqq 3n + M(n_1) + M(n_2) + M(n_3),$$

and it follows that $M(n) = O(n \log n)$.

We now estimate the running time $T(n)$ of the partitioning process. It is very essential to note here what is meant by "creating" subsets. The creation of the subsets contributed by the first decomposer requires $O(n \log n)$ time, since we need to compute all distances from the decomposer to all other nodes and then sort them. However, the creation of other subsets (i.e., subsets contributed by nondecomposers) requires only a few pointers, as discussed later in this section. Thus, the general step in the partitioning process consists of: (i) tree decomposition, $O(n)$; (ii) computing all distances from a single node, $O(n)$; (iii) sorting the set of all these distances and discarding those associated with previous decomposers, $O(n \log n)$; (iv) creating the subsets, pointers and constants, $O(n)$. Thus, the recursive relation is

$$T(n) \leqq Cn \log n + T(n_1) + T(n_2) + T(n_3),$$

and therefore $T(n) = O(n \log^2 n)$.

Next, we discuss the storage aspects of the partitioning algorithm. Whenever a node x serves as a decomposer for a subtree T_1, three sorted sets R_1, R_2, R_3 of distances from x into T_1 are generated. We distinguish between the sets R_i and the subsets S_i that actually constitute our partition. Each set is stored as an array, and the total space for storing these arrays is $O(n \log n)$. (This can be proved by induction.) The partition of S into subsets $S_1, \cdots, S_m$, as well as the reduced forms of S that are processed by the trimming or splitting procedures (see § 2), are handled as follows. Each S_i is characterized by four items. First is a pointer to the corresponding R_j from which S_i is created. Second is a constant number that should be added to an element of R_j in order to get an element of S_i. Third and fourth are two pointers needed to specify the boundary of that portion of R_j from which S_i is generated. (These two pointers are at the start the same for all the S_is that rely on the same R_j, but during the trimming or splitting process they may become different.) Thus, the total amount of storage that we need is $O(n \log n)$. In addition, at most $O(n \log n)$ storage is required in order to properly maintain the set of trees T_i which are generated throughout the algorithm.

We conclude this section with a pidgin Algol description of the partitioning process. It receives as input a tree T with a set of nodes $N = \{1, \cdots, n\}$ and produces as output a partition $S_1, \cdots, S_m$ of the set of internodal distances of T, where $m = O(n \log n)$. The sets $S_1, \cdots, S_m$ satisfy the properties (i)–(iii) of § 2. The overall complexity bounds for the algorithm are $O(n \log^2 n)$ time and $O(n \log n)$ space. The procedure uses the following terminology:

Q	current set of subtrees not yet subdivided.
B	current set of nodes which have not as yet served as decomposers.
R_j	jth sorted set of distances between a decomposer and the nodes of a subtree.
k	index for set S_k used in the partition.
$\gamma(k)$	a label identifying the index of subset R_j used to create S_k.
$\beta(k)$	the constant increment which must be added to each element of R_j to get the corresponding element of S_k.

In addition, the procedure uses the following subroutines in the course of its execution:

CENTROID (T)	Given a tree T returns its centroid.
SUBTREE (T, x, i)	Given a tree T, its centroid x and an index $i = 1, \cdots, 3$, returns the subtree T_i (see last paragraph of § 3).
DISTANCE (T, A, B)	Given a tree T and two sets of nodes A and B returns a vector of all the distances $d(i, j)$, $i \in A$, $j \in B$, $i \neq j$.
SORT (D)	Given a vector D, returns the entries in a sorted way.

Procedure DECOMPOSE (T)
begin
  $Q \leftarrow T$
  $B \leftarrow N$
  $j \leftarrow 0$
  $k \leftarrow 0$

```
while Q ≠ ∅ do
  begin
    choose T' from Q
    x ← CENTROID (T')
    for i = 1, · · · , 3 do
      begin
        T_i ← SUBTREE (T, x, i)
        N_i ← Nodes of T_i
        N'_i ← N_i ∩ B\{x}
        D_i ← DISTANCE (T_i, x, N'_i)
        j ← j + 1
        v_i ← j
        R_j ← SORT (D_i)
      end
    for each j ∈ N'_1 do
      begin
        k ← k + 1
        γ(k) ← v_2
        β(k) ← d(j, x)
        k ← k + 1
        γ(k) = v_3
        β(k) ← d(j, x)
      end
    for each j ∈ N'_2 do
      begin
        k ← k + 1
        γ(k) ← v_3
        β(k) ← d(j, x)
      end
    if x ∈ B do
      begin
        for i = 1, · · · , 3 do
          begin
            k ← k + 1
            γ(k) ← v_i
            β(k) ← 0
          end
        B ← B\{x}
      end
    for i = 1, · · · , 3 do
      begin
        if |N_i| ≧ 3 then Q ← Q ∪ T_i
        else
          D_i ← DISTANCE (T, N'_i, N'_i)
          j ← j + 1
          R_j ← SORT (D_i)
          k ← k + 1
          γ(k) ← j
          β(k) ← 0
      end
end
```

5. The kth longest path in a tree. Once the partition of the set S of all paths into $m = O(n \log n)$ subsets is established, one can use the techniques introduced in § 2 to find the kth longest path. This amounts to an effort of $O(m \log n/m) = O(n \log^2 n)$ if one uses trimming and an inferior bound, of $O(n \log^3 n)$, if splitting is used. As the effort involved in generating the partition of S is also $O(n \log^2 n)$, we can conclude that the overall effort for finding the kth longest path in T is $O(n \log^2 n)$.

Can this bound be further beaten down? Possibly, but the margin for improvement is slim. An $O(n \log n)$ lower bound on the complexity of the problem can be obtained in a number of ways. The following simple reduction was offered to us by one of the referees. Consider the tree of Fig. 1 where the heavy line in the center is chosen long enough to ensure that the longest paths in T include one element from X and one from Y. Thus, the well-known $O(n \log n)$ bound on selection in $X + Y$ is valid for our problem as well.

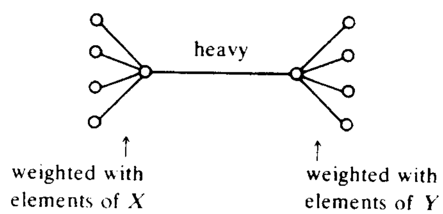

FIG. 1

6. Location problems. We consider here the following different problems of location. First, we assume that a tree T is embedded in the Euclidean plane, so that the edges are line segments whose endpoints are the nodes and whose edges intersect one another only at nodes. Moreover, each edge has a positive length. (Any tree with positive edge-weights can be so embedded in R^2). This embedding enables us to talk about points, not necessarily nodes, on the edges. We then denote by $d(x, y)$ the distance, measured along the edges of the tree, between any two points x, y of the tree.

In a typical location problem, one has to select p points of the tree under different assumptions depending on the particular model considered. In each model, we distinguish between the "supply" set Σ (this is the set from which we select the p points) and the "demand" set Δ, with reference to which the objective function is defined. The p-center problem seeks to choose p points $x_1, \cdots, x_p$ from Σ so as to minimize $\sup_{y \in \Delta} \min_{1 \leq i \leq p} d(x_i, y)$. The four special cases, where the sets Σ and Δ are either the set of all nodes or the set of all points of the tree, have been discussed and given different algorithms in [2], [3], [4], [9], [13].

Following Handler [9], we use the categorization scheme $\{{}^N_A\}/\{{}^N_A\}/p$, interpreted as follows. The first cell describes the supply set Σ, which could be either the set N of all nodes or the set A of all points. The second cell describes the demand set Δ, which could also be either N or A. The third cell indicates the number of points that we have to select from Σ. For example, $N/A/2$ refers to selecting two nodes so as to minimize the maximum (over all points of the tree) of a distance between a point of the tree and the selected node that is nearest to that point. Kariv and Hakimi [13] provide $O(n^2 \log n)$ algorithms for $A/N/p$ and $N/N/p$. Chandrasekaran and Tamir [3], using a unified approach, solve $A/N/p$, $N/N/p$ and $N/A/p$ in $O(n^2 \log n)$ time. The $A/A/p$ problem is solved in [4] in $O(n^2 \log^2 p)$ time.

All the algorithms mentioned above are based on the same principle. First, a finite set R of real numbers, which is known to contain the optimal objective function value, is identified. Next, we search R for the minimum value which is feasible in the following sense. A value $r > 0$ is feasible if there exists a set of p points $x_1, \cdots, x_p$ of Σ such that the distance between any demand point y and its nearest x_i, is not greater than r. Efficient algorithms are known for deciding whether a given r is feasible, and hence the location problem can be solved by a binary search of R using such a feasibility test. For all four problems this test runs in $O(n)$ time. (See [13] for $N/N/p$ and $A/N/p$ and [4] for $N/A/p$ and $A/A/p$.) The set R of relevant values in the four different problems is given in Table 1 (see [3], [4], [13]).

TABLE 1

Model	The set R
$N/N/p$	$\{d(i,j)\}_{i,j\in N}$
$A/N/p$	$\{\frac{1}{2}d(i,j)\}_{i,j\in N}$
$N/A/p$	$\{d(i,j), \frac{1}{2}d(i,j)\}_{i,j\in N}$
$A/A/p$	$\{(1/2k)d(i,j)\}_{i,j\in N, k=1,\cdots,p}$

Along the lines discussed above, each one of these problems can be solved by computing the set R and then searching R by repeatedly using linear-time median finding [1]. This amounts to $O(|R|+n \log |R|)$ time where $|R|$ is the dominant term. Thus, in order to improve this upper bound, one has to bypass the computation of the set R and still be able to search in that set. This is essentially where we apply the techniques developed in the previous sections.

7. Improved algorithms for location problems. The sets R of relevant values for the various versions of the p-center problem bear a close resemblance to the set S of internodal distances on T. Thus, we can use any algorithm for finding the kth longest element in S to support a binary search over R. Such search involves at each iteration identifying the median element of R, performing the feasibility test and finally discarding one half of the elements. However, we note that identifying the median element at each iteration may be more than one needs. In fact, one can do better by applying a search strategy similar to that of splitting.

Assume that the set R is partitioned into m subsets $R_1, \cdots, R_m$ such that the kth largest element in each subset can be found in constant time. We can employ the following procedure. First, we find the median element z_i in each subset R_i. Next, we find the median element z in the set $z = (z_1, \cdots, z_m)$ relative to the weights $|R_i|$. Thus, z is between the lower and upper quartiles of R. This value z can now be tested for feasibility. The test takes $O(n)$ time and determines whether the optimal value v is greater than z (this is when z is not feasible) or not. If $v > z$, we discard the lower half (including z_i) from each R_i such that $z_i \leqq z$. If $v \leqq z$, we discard the upper half (including z_i) from each R_i such that $z_i \geqq z$, with the exception that z itself is not discarded. The search then proceeds with the reduced set R until the optimal value is singled out.

Since each reduction eliminates one quarter of the remaining set, the number of such stages is $O(\log |R|)$. Following is a more detailed analysis for the particular cases.

A. $N/N/p$. Here R is the set of the internodal distances. It follows from § 4 that R can be appropriately partitioned into $O(n \log n)$ subsets where the partitioning process takes $O(n \log^2 n)$ time. During the searching process, in each iteration we need $O(n \log n)$ time for identifying the element z and $O(n)$ time for the feasibility test. Thus, the searching stage takes $O((n + n \log n) \log |R|)$ time, and hence the location problem is solved in $O(n \log^2 n)$ time.

B. $A/N/p$. Since $R = \{\frac{1}{2}d(i, j): i, j \in N\}$ in this case, the location problem is solved by the same partition which is used in $N/N/P$. Hence, the time bound for this case is $O(n \log^2 n)$.

C. $N/A/p$. The relevant set here is $R = \{d(i, j), \frac{1}{2}d(i, j): i, j \in N\}$. Thus, we use essentially the same partition as the one for $N/N/p$ and $A/N/p$, but in terms of pairs of nodes, each pair is counted twice: once for the distance $d(i, j)$ and once for the number $\frac{1}{2}d(i, j)$. This implies the same time bound of $O(n \log^2 n)$ for this case too.

D. $A/A/p$. This last case is slightly more complicated than the previous ones. Since $R = \{(1/2k)d(i, j): i, j \in N, k = 1, \cdots, p\}$ in this case, one way of partitioning R is by using the partition of § 4 for the set of pairs of nodes and replicating each subset p times, so that each $d(i, j)$ is multiplied by all the numbers $\frac{1}{2}, \frac{1}{4}, \frac{1}{6}, \cdots, 1/2p$. Thus, R is partitioned into $m = O(pn \log n)$ subsets while $|R| = O(pn^2)$. By applying the searching method, R is successively reduced by a factor of one quarter. Let the set of remaining variables at a given iteration be R', and denote by $T(|R'|)$ the time required by the algorithm to handle this set. We now have

$$T(R') \leqq C_i m + c_2 n + T(3/4|R'|).$$

When the cardinality of R' reaches the level $|R'| = O(m)$ we can search over R' directly using the method of linear-time median finding. This involves, at each iteration, finding the median element and performing the test. The total effort involved in the identification of *all* the median elements is clearly $O(m)$. Also, since the number of iterations is $O(\log m)$, and since each test requires an effort of $O(n)$, the total effort associated with handling a set of cardinality $O(m)$ is $O(n \log m) = O(m)$. Solving the recursion relation with the initial condition $T(m) = O(m)$ we then get that $T(|R'|) = O(m \log (|R'|/m))$ and hence the location problem is solved in this approach in time

$$O\left(pn \log n \log \left(\frac{n}{\log n}\right)\right) = O(pn \log^2 n).$$

There is an alternative partition that in some cases leads to a better upper bound. We first compute the $\frac{1}{2}n(n-1)$ internodal distances (in $O(n^2)$ time). Then we partition R into $m = \frac{1}{2}n(n-1)$ subsets of the form $R_{ij} = \{(1/2k)d(i, j): k = 1, \cdots, p\}$, where computing the kth largest element in each set is trivial. Applying the searching procedure we obtain the following bound:

$$O(m \log (|R|/m)) = O(n^2 \log p).$$

To summarize, the different cases are solved with the upper bounds in Table 2.

TABLE 2

Model	Upper bound
$N/N/p$	$O(n \log^2 n)$
$A/N/p$	$O(n \log^2 n)$
$N/A/p$	$O(n \log^2 n)$
$A/A/p$	$O(n \min\{p \log^2 n, n \log p\})$

8. Location problem with weighted demands. A more general type of location problem is where the demand is weighted. Specifically, when $\Delta = N$ we may have weights $w_i > 0$ $(i \in N)$ and seek to select $x_1, \cdots, x_p \in \Sigma$ so as to minimize

$$\max_{i \in N} \{w_i \cdot \min_{1 \leqq j \leqq p} d(x_j, i)\}.$$

It is shown in [3], [13] that the relevant set R generalizes to $\{w_i d(i, j)\}_{i,j \in N}$ in the $N/N/p$ case and to $\{(w_i w_j/(w_i + w_j))d(i, j)\}_{i,j \in N}$ in the $A/N/p$ case. Both cases are solvable in $O(n^2)$ time [4], [13], but based on our method an $O(n \log^2 n)$ algorithm for the weighted $N/N/p$ case can be constructed as follows.

Essentially, we consider the set S' of all *ordered* pairs of nodes together with the linear order induced by the weighted distances $w_i d(i, j)$. The set S' is partitioned, along lines similar to those of § 4, into $O(n \log n)$ subsets. All the pairs (i, j) in any subset are with the same i, hence the restriction of the order to each subset is independent of the weight w_i. All we have to do during the algorithm, is to multiply the kth largest distance in a set corresponding to i by the weight w_i. Thus the partition satisfies all the properties that are required to obtain a bound of $O(n \log^2 n)$.

REFERENCES

[1] M. BLUM, R. W. FLOYD, V. R. PRATT, R. L. RIVEST AND R. E. TARJAN, *Time bounds for selection*, J. Comp. Sys. Sci., 7 (1972), pp. 448–461.

[2] R. CHANDRASEKARAN AND A. DAUGHETY, *Location on tree networks: p-center and n-dispersion problems*, Math. Oper. Res., to appear.

[3] R. CHANDRASEKARAN AND A. TAMIR, *Polynomially bounded algorithms for locating p-centers on a tree*, Discussion Paper No. 358, Center for Mathematical Studies in Economics and Management Science, Northwestern University, Evanston, IL, 1978.

[4] ———, *An $O((n \log P)^2)$ algorithm for the continuous p-center problem on a tree*, Discussion Paper No. 367, Center for Mathematical Studies in Economics and Management Science, Northwestern University, Evanston, IL, 1978.

[5] B. L. FOX, *Discrete optimization via marginal analysis*, Management Sci., 13 (1966), pp. 210–216.

[6] G. N. FREDERICKSON AND D. B. JOHNSON, *Optimal algorithms for generating quantile information in $X + Y$ and matrices with sorted columns*, Proceedings of the 1979 Conference on Information Sciences and Systems, The Johns Hopkins University, to appear.

[7] Z. GALIL AND N. MEGIDDO, *A fast selection algorithm and the problem of optimum distribution of effort*, J. Assoc. Comput. Mach., 26 (1979), pp. 58–64.

[8] A. J. GOLDMAN, *Optimal center location in simple networks*, Transportation Sci., 5 (1971), pp. 212–221.

[9] G. Y. HANDLER, *Finding two-centers of a tree: The continuous case*, Transportation Sci., 12 (1978), pp. 93–106.

[10] D. B. JOHNSON AND D. S. KASHDAN, *Lower bounds for selection in $X + Y$ and other multisets*, J. Assoc. Comput. Mach., 25 (1978), pp. 556–570.

[11] D. B. JOHNSON AND T. MIZOGUCHI, *Selecting the Kth element in $X + Y$ and $X_1 + X_2 + \cdots + X_m$*, this Journal, 7 (1978), pp. 147–153.

[12] C. JORDAN, *Sur les Assemblayes des lignes*, J. Reine Angew. Math, (1869), pp. 185–190.

[13] O. KARIV AND S. L. HAKIMI, *An algorithmic approach to network location problems. I: The p-centers*, SIAM J. Appl. Math, 37 (1979), 513–538.

[14] ———, *An algorithmic approach to network location problems. II: The p-medians*, SIAM J. Appl. Math, 37 (1979), pp. 539–560.

[15] 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.