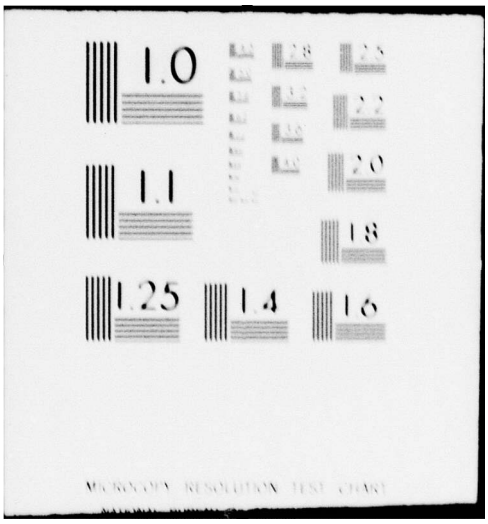
AD-A061 627 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
DECOMPOSABLE SEARCHING PROBLEMS.(U)
OCT 78 J L BENTLEY N00014-76-C-0370
UNCLASSIFIED CMU-CS-78-145 NL

1 OF 1
AD
A061 627

END
DATE
FILMED
2-79
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART

CMU-CS-78-145

B.S. 12

LEVEL II

DECOMPOSABLE SEARCHING PROBLEMS

Jon Louis Bentley

Oct 78

Interim rept.

15 p.

N00014-76-C-0370

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-78-145

DECOMPOSABLE SEARCHING PROBLEMS

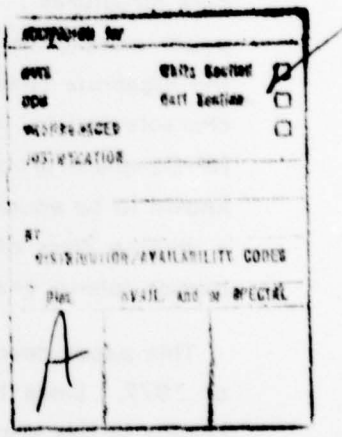
Jon Louis Bentley
Departments of Computer Science and Mathematics
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

Although searching is one of the most important problems in computer science and many particular results are known for searching problems, there really is no satisfactory "theory of searching". In this paper we propose a first step toward such a theory by defining the class of *decomposable* searching problems and then proving three theorems about problems in this class. These theorems are all of the form "given a data structure D for a decomposable searching problem we can transform D into a new data structure D' for a related problem". The correctness and complexity analysis of D are then used to establish the correctness and complexity of D'. We present transforms for converting a static structure into a dynamic structure, for adding "range variables" to queries, and for making preprocessing/query time tradeoffs. These transforms have already been used to develop a number of best known "theoretical" algorithms, and promise to be an important tool in software engineering.

This research was supported in part by the Office of Naval Research under Contract N00014-76-C-0370.

Table of Contents



1. Introduction

The young field of "Concrete Computational Complexity" has given birth to a large number of interesting and useful results in the past decade. These results have allowed us to apply the tools of mathematics to the problems of describing real-world costs of computations. Unfortunately, however, most of these results have been of a fairly limited scope; the literature is rich with theorems of the form "this problem is of this complexity" but has few statements such as "all problems in this class have this complexity". In this paper we define such a class of problems and prove three theorems about the complexity of problems in that class.

We investigate the problem of algorithms and data structures for searching and focus our attention on a class of searching problems we call *decomposable*. We give an algebraic definition of decomposable searching problems that allows one to test whether a given problem is in fact decomposable. We then give three constructions that allow one to transform a data structure for a "common" decomposable problem into a structure that solves a more "exotic" problem. We make no assumptions about the underlying "common" structure; we use only the fact that the problem is decomposable.

We see this work as one of the first steps towards Tarjan's [1978] goal of a "calculus of data structures". Although new results are presented, we see the main contributions of this paper more in its systematization of a broad class of results. We characterize the class by the *algebraic* structure of the problem statement, which is a fundamentally different kind of characterization than those used to identify other well-known complexity classes. The NP-Complete problems are defined by *reducibilities*, as are many diverse problems that are known to be equivalent in complexity to matrix multiplication. Lipton and Tarjan [1977] solve a diverse class of problems by giving a *method* applicable for many problems in a specific domain (planar graphs).

This paper describes the research on decomposable problems that was done up to the end of 1977. Since that time the author and James B. Saxe have found a number of further results in this area. This paper can therefore be viewed as an introduction; a more complete description of decomposable searching problems will appear in Bentley and Saxe [1978]. In Section 2 of this paper we present our model of searching and define the class of decomposable problems within that framework. The three main constructions are sketched in Sections 3, 4, and 5. In Section 6 we present conclusions.

2. Decomposable Problems

A *static* searching problem is usually given as follows: preprocess a set (or "file") F of N objects into a data structure D such that certain kinds of queries about F can be answered quickly. (In some contexts F is a multiset; we refer to it here as a set for brevity.) To analyze the structure D we give three functions of N: $S_D(N)$, the amount of storage required by D; $P_D(N)$, the preprocessing time required to build D; and $Q_D(N)$, the time required to answer a query. Unless explicitly stated otherwise, throughout this paper we will assume that we are measuring the "worst-case" complexity of these quantities.

The most well known example of a searching problem is usually called the *Member* problem, given as follows: preprocess N elements from a totally ordered set such that queries of the form "is x in set F?" can be answered quickly. A common solution to this problem is to store the elements of F in a vector sorted into increasing order and then answer queries by binary search. Analyzing this vector scheme shows that $P_V(N) = O(N \lg N)$, $S_V(N) = O(N)$, and $Q_V(N) = O(\lg N)$. A more difficult kind of searching problem is the *Nearest Neighbor* or *Post Office* problem which calls for preprocessing a set F of N points in the plane to facilitate queries of the form "what is the nearest neighbor in F to point x?" (where x is not necessarily in F). A data structure recently given by Lipton and Tarjan [1977] has $P_L(N) = O(N \lg N)$, $S_L(N) = O(N)$, and $Q_L(N) = O(\lg N)$.

Both Member and Nearest Neighbor are decomposable problems. Before formally describing the class of decomposable problems, we will illustrate certain features of the class with examples from these two problems. The first point to be made is that it is the Member and Nearest Neighbor *problems* themselves that are decomposable, and not the particular data structures or algorithms used to solve the problems. An informal definition of decomposability is that a search problem is decomposable if one can answer a query about set F by combining the answers to the query asked of sets A and B, where A and B are an (arbitrary) partition of F into two subsets. Member is decomposable because x is a member of F if and only if x is a member of A or x is a member of B, for any partitioning of F into A and B. Likewise Nearest Neighbor is decomposable because the distance from x to its nearest neighbor in F is the minimum of the distances from x to its nearest neighbors in A and B, once again for any partition. (We deal here with distance to nearest neighbor for tractability; extra bookkeeping will tell which point realizes that distance.)

We can now give a more formal definition of decomposability. We say that a searching problem is decomposable if the response to a query asking the relation of a new object x to set F can be written as

$$Q(x,F) = \square_{f \in F} q(x,f).$$

We assume that $\square$ is the repeated application of the binary operator $\square$ over all elements in its domain. For $\square$ to be well defined mathematically $\square$ must be associative, commutative, and have an identity; for computational efficiency we require that $\square$ be computable in constant time. We can cast Member in this framework as

$$\text{Member}(x,F) = \text{OR}_{f \in F} \text{equal}(x,f)$$

and Nearest Neighbor as

$$\text{NN}(x,F) = \text{MIN}_{f \in F} \text{distance}(x,f).$$

It is clear that the queries that can be cast in this schema obey the informal definition of decomposability we gave above; for any partition of F into A and B, we know by the associativity and commutativity of $\square$ that

$$\begin{aligned} Q(x,F) &= \square_{f \in F} q(x,f) \\ &= \square \left(\square_{g \in A} q(x,g), \square_{h \in B} q(x,h) \right) \\ &= \square \left(Q(x,A), Q(x,B) \right). \end{aligned}$$

More than twenty decomposable searching problems have been identified; we now mention a few of these. From problems on linearly ordered sets (where the file to be preprocessed contains elements from such a set) we have already seen the example of Member. Further examples are Successor (what is the least element in F greater than x?), Rank (how many elements in F are less than x?), and Count (how many elements in F have value x?). In Data Base problems, all of the problems that Rivest [1974] calls intersection queries are decomposable; these include secondary key retrieval, partial match searching, and range searching. Computational Geometry abounds with examples of decomposable problems. We have already mentioned Nearest Neighbor; related decomposable problems are Farthest Neighbor (which point is most distant from x?) and Fixed Radius Near Neighbors (which points are within some fixed distance d of x?). Queries dealing with more complicated geometric structures can also be decomposable; examples of such problems are given by Dobkin and Lipton [1976].

Although many searching problems are decomposable, some others are not. An example of a problem that is not decomposable is convex hull inclusion (preprocess N points in the plane; a query asks if a new point is within their convex hull). For any given point x within the convex hull of F it is not hard to find a partition of F into A and B such that x is not within

the convex hull of either A or B. We can use this to prove that convex hull inclusion is not decomposable.

3. Dynamic Structures

In this section we turn our attention from static problems to dynamic problems. In static problems the data is organized once-and-for-all before any searches are done. In dynamic problems the set F (and the data structure D) are initially empty and elements are then added to F one-by-one. (The term dynamic sometimes implies that elements can also be deleted from F; we do not use it in that sense.) The most well known example of a dynamic data structure is the AVL tree described by Knuth [1973]. An AVL tree allows N elements to be inserted at a total cost of O(N lg N); at any point one can answer Member queries in O(lg N) time. In this section we will first develop a dynamic structure for the Nearest Neighbor problem, and then show how the techniques we employ can be used to convert any static structure for a decomposable problem into a dynamic structure.

There are two naive approaches to this problem. The first stores the points in the plane in a linked list, inserts a new point by appending it to the front of the list, and answers a query by examining every point in the list. Calling this structure B (for "brute force") we have

$$P_B(N) = S_B(N) = Q_B(N) = O(N).$$

A second naive solution calls for using the Lipton/Tarjan structure and rebuilding it after each insertion. This "rebuilding" scheme yields $P_R(N) = O(N^2 \lg N)$, $S_R(N) = O(N)$, and $Q_R(N) = O(\lg N)$. In choosing between these schemes one could achieve very good insertion time or very good query time, but not both. We will now develop a structure that has both.

Our faster structure will consist of a set of Lipton/Tarjan structures (which we abbreviate as *LTS*), each of a distinct size which is a power of two. Our structure is initially empty. When the first point is inserted, we build an LTS of size one. When the second point is inserted, we build the two points into an LTS of size two, discarding our previous LTS of size one. When the third point is inserted we have two LTSs (of sizes one and two), and after the fourth point is inserted we have one LTS of size four. Insertions proceed in a way analogous to binary counting: after the N-th element is inserted we have an LTS of size 2^j if and only if the j-th bit of the binary integer N is one. (This structure is similar to the binomial queues of Vuillemin [1978].) To answer a Nearest Neighbor query for a new point x we perform a nearest neighbor search for x in all the LTSs in our structure, then return the minimum of those as the answer to the query.

To analyze our structure (which we call L', because it is a transformation of the LTS L) we

note that if L' contains N elements then we are using at most lg N LTSs. Since each of those is of size less than N we know that we can perform a nearest neighbor search on any of them in O(lg N) time. Therefore the time required to search is $Q_{L'}(N) = O(\lg^2 N)$. Since each of the LTSs requires space linear in the number of elements it contains, the total space requirement of L' is $S_{L'}(N) = O(N)$. To count the total cost of having inserted N elements into L' is a bit more difficult. The total cost of building LTSs of size $m = 2^j$ is the cost of building one LTS of size m multiplied by the number of times the j-th bit in a binary word turns from zero to one in counting from zero to N. If we assume that N is a power of two then a simple sum shows that $P_{L'}(N) = O(N \lg^2 N)$, and we can use this fact to show that $P_{L'}(N)$ is of the same order for N not a power of two.

The structure that we have used for dynamic Nearest Neighbor searching can be used to convert any static structure for a decomposable problem into a dynamic structure for that problem. We will assume that we are given a static structure D with performances $P_D(N)$, $S_D(N)$, and $Q_D(N)$. Our dynamic structure D' will consist of a set of static structures, each of distinct size a power of two, built as before in a manner analogous to binary counting. Analyzing D' shows

$P_{D'}(N) = O[P_D(N) \lg N]$,
$Q_{D'}(N) = O[Q_D(N) \lg N]$, and
$S_{D'}(N) = O[S_D(N)]$.

The above analysis shows that we can convert a static structure to a dynamic structure with only a logarithmic increase in query and insertion times. We can achieve even better results for some structures. If the static query time $Q_D(N)$ grows as $\Omega(N^\epsilon)$ for some $\epsilon > 0$, then we can show that $Q_{D'}(N)$ is bounded above by some constant times $Q_D(N)$. Likewise if $P_D(N)$ grows faster than $N^{1+\epsilon}$, then we will not incur the additional logarithmic factor in $P_{D'}(N)$. For certain structures we do not have to build the static structures over again from scratch--we can *merge* existing structures. In that case too we can avoid the extra logarithmic term in $P_{D'}(N)$. (We can give at least five examples of such structures.) We can also show that for some problems the increase of the logarithmic factor is not incurred if we measure average instead of worst-case query times.

This static-to-dynamic transformation has already been used to yield a number of new algorithms. Bentley, Detig, Guibas and Saxe [1978] describe "binomial lists", which were obtained by applying the transformation to sorted arrays. They show that with proper implementation the storage used by binomial lists is absolutely minimal and that their structure is optimal over the class of all structures using only minimal storage. In addition to theoretical interest, their structure is efficiently implemented and experiments show that it is competitive with other dynamic member structures in many applications. Another use of this

transform has been given by Yao, Yao and Bentley [1978]. They reduced the complexity of calculating the "rank function" in a vector set from barely sub-quadratic to N times a polynomial in lg N by transforming a static maxima searching structure to dynamic.

Many other aspects of converting static search structures to dynamic will be described by Bentley and Saxe [1978]. In this section we have seen a transform that increases both query and processing costs by a factor of O(lg N). Bentley and Saxe display a whole set of transformations which add a factor of k to query time and a factor of $kN^{1/k}$ to processing time, for any fixed integer k. (For example, it is possible to convert a static structure to dynamic by adding a factor of 5 to query time and $5N^{1/5}$ to processing time; this might be useful if many queries were anticipated.) Further, they show that using only the properties of decomposability, these transformations are optimal to second-order terms. They then demonstrate "dual" transformations that add a factor of k to processing times at the cost of increasing query times by a factor of $(k^2/2)N^{1/k}$. They investigate the question of dynamic structures in which elements can be deleted as well as inserted and show that although deletion is provably infeasible in general, if $\square$ is invertible then deletion can be achieved at a cost of only a constant factor. They also show how the processing time between insertions can be bounded; this is useful in on-line systems. In related work, Bentley and Shaw [1978] have formally proved the correctness of the static-to-dynamic conversion discussed in this chapter by writing it as an Alphard form and proving its correctness in that context. Similar methods can be used to prove formally the correctness of all the transforms of Bentley and Saxe.

4. Adding Range Variables

In the last section we showed how to transform a static structure into a dynamic structure; in this section we will show how to transform a static structure for a particular query into a static structure for a related query. We will illustrate the construction by first applying it to the planar Nearest Neighbor problem, and then considering the general case. The related searching problem that we will solve is most easily stated if the points in the plane are viewed as cities, each of which has an associated population. We are to preprocess the N cities and their populations. A query will give a point x in the plane and a population range (described by upper and lower bounds), and the search must determine which of the cities in the desired population range is the closest to point x.[1] (So a query might ask "of *all* the cities with population between 60,000 and 120,000, which is the closest to point x?".)

---

[1] This search is of the same form as one described by Knuth [1973, p. 550] as "more complicated than those already quite difficult and therefore usually not considered".

The data structure we use to solve this problem is a tree that has LTSs in all its nodes. The root of our tree contains an LTS representing all the cities in F. The left son of the root will represent all cities with population less than the median population, and the right son will represent the cities with populations greater than the median. In each of those nodes there will be an LTS representing half of the cities. This partitioning continues so that on the i-th level of the tree there are 2^i LTSs, each representing cities contiguous in the population dimension. To answer a query for the nearest neighbor to point x in population range R we will search for the nearest neighbor to x in a subset of the LTSs (the union of which is all of the cities in the range), and then return the minimum of the reported distances. Specifically we search all the the LTSs that represent a population range contained in R and having fathers with range not contained in R. We can easily describe exactly which LTSs will be searched by a recursive algorithm which visits the nodes in the tree containing the relevant LTSs.

The first step in analyzing our structure is to note that the tree we build is lg n levels deep. The time required for building the LTSs on each level is bounded above by O(N lg N), so we have $P_{L'}(N) = O(N \lg^2 N)$ (L' now represents the LTS with range restriction capability added). Likewise the storage required on any level is O(N), so $S_{L'}(N) = O(N \lg N)$. Since at most two LTSs are searched on any level (at cost bounded above by O(lg N)), we have $Q_{L'}(N) = O(\lg^2 N)$. Thus we see that our new structure adds a factor of lg N to each of the cost functions we measure.

We consider now the general case in which each element f of the set F contains an additional population dimension written p(f). The modified query we want to answer (for object x and range R) is

$$Q'(x,R,F) = \square_{\substack{f \in F \\ p(f) \in R}} q(x,f).$$

The structure we use is the tree structure described above in which each node of the tree contains one of the original structures D. Analyzing the resulting structure D' as before shows that a factor of lg N is added to each of the cost functions giving

$P_{D'}(N) = O[P_D(N) \lg N]$,
$S_{D'}(N) = O[S_D(N) \lg N]$, and
$Q_{D'}(N) = O[Q_D(N) \lg N]$.

This transformation can be used to solve the "range searching" problem defined by Knuth [1973, p. 554]. In this problem we preprocess N points in k-space and answer subsequent queries asking for all points that have every key value in some specified range (that is, for all

points lying in some rectilinearly-oriented hyperrectangle). Range searching in one dimension can be accomplished by use of a sorted vector. We can use this structure as a "base" and add k-1 additional range variables by the transform of this section. We then apply the "speedup" of merging structures to yield the "range tree" structure R with performances

$P_R(N,k) = O(N (\lg N)^{k-1})$,
$S_R(N,k) = O(N (\lg N)^{k-1})$, and
$Q_R(N,k) = O((\lg N)^k + F)$

where F is the number of points found in response to the query. This structure is the best known structure for range searching, and was discovered through the use of this decomposable transform. This transformation can also be used to derive the data structures for "Maxima Searching" and "ECDF Searching" described by Bentley and Shamos [1977].

Further aspects of adding range variables will be described by Bentley and Saxe [1978]. Just as in the static-to-dynamic transformation there were a set of conversions, so there are in this transformation also. We have seen in this section a transformation which increases the preprocessing, storage, and query costs by a factor of O(lg N). Bentley and Saxe show a set of transformations that increase preprocessing and storage costs by a factor of k and query costs by a factor of $O(N^{1/k})$. The dual transform adds a factor of k to query costs and increases preprocessing and storage requrements by $O(N^{1/k})$. They also show that these increases are not always incurred. They can be avoided in exactly the same way as in the static-to-dynamic conversion: if the underlying structures can be merged quickly, if the underlying functions are fast growing, or if average query times are considered. The transformations that we have described in this section add a range variable to a static structure, yielding a new static structure. Lueker [1978] has described a new transformation that is applied to a dynamic sructure and yields a new dynamic structure in which range variables can be specified. He has used this transform to produce best-known structures for dynamic range searching, ECDF searching, and maxima searching.

5. Preprocessing/Query Time Tradeoffs

In this section we examine the final construction for decomposable problems. Unlike the two other constructions, this construction is not due to the author. It has been used previously in a number of algorithms, and it is included here because its scope of applicability is precisely the decomposable problems. The construction is appropriate for a structure that has very high preprocessing time or storage; it allows us to develop a class of structures with decreased preprocessing and storage requirements at the cost of increased search time.

The data structure makes use of *clusters*. The preprocessing partitions the N elements of F (at random) into N/c clusters of c elements each, then applies the preprocessing algorithm of

D to each of the clusters to build N/c structures. To answer a query we search each of the clusters and then combine those answers (by decomposability) to form an answer to the original query. Analyzing the resulting structure D' shows

$Q_{D'}(N) = (N/c)\, Q_D(c)$,
$S_{D'}(N) = (N/c)\, S_D(c)$, and
$P_{D'}(N) = (N/c)\, P_D(c)$.

Notice that a continuum of performances is achievable.

This transform has been used by A. Yao [1977] to achieve a number of new results. He used as "underlying" structures those presented by Dobkin and Lipton [1976] that had logarithmic search times but preprocessing and storage that were (a large) polynomial in N. By choosing c appropriately (as a function of N) Yao was able to achieve the best known algorithms for many multidimensional problems. Yao's methods can be applied to many other problems that do not *prima facie* appear to be searching problems. These are problems which ask a "decomposable function" to be computed over every element in a set. Naive algorithms for performing this task require quadratic time, but the problem can be reduced to performing N decomposable searches. Yao's methods can then be applied to yield sub-quadratic algorithms. There are other applications of this transformation of structures in more typical searching contexts. In many applications the available storage is bounded, so we can use these methods to design a structure that will use exactly as much storage as is available. If we know in a certain application exactly how many searches will be made, then we can choose the cluster size to minimize the total cost of preprocessing and searching.

6. Conclusions

To summarize this paper we have seen at least two types of results. On a higher level we have seen three transforms that can be applied to searching structures for decomposable problems: converting a static structure to dynamic, adding "range variables" to a structure, and making preprocessing/query time tradeoffs. These transforms have already been used to yield results on a more concrete level. These include the currently best known "theoretical" algorithms for such problems as the rank function, member searching, range searching, and minimal spanning trees. These transforms can also be used as software engineering tools: each transform is easily coded and would be a valuable entry in a "software engineering handbook".

Throughout this paper we have mentioned other work on decomposable problems that has already been done and will be reported by Bentley and Saxe [1978]. Much further work, however, remains to be done. Open problems include identifying additional decomposable problems, showing more efficient transforms than those we discussed, giving new types of

transforms, showing the optimality of transforms, performing exact (Knuthian) analysis of transforms, and developing software to implement these transforms. Perhaps the most obvious open problem, though, is to identify other classes of problems and prove general results concerning problems in the class.

In conclusion, this paper contains three kinds of results. The first kind of result that we have seen is a set of particular algorithms; the framework of decomposable problems has been useful in both the discovery and the presentation of these algorithms. The second kind of result we have seen is the three particular transforms, and the different tradeoffs available within each transform. The final kind of result in this paper is the definition of the class of decomposable problems. To the author's knowledge, this is the first example of a class of problems that is defined algebraically and for which a general set of conversions is known. This kind of result promises to be valuable both in proving theorems about the asymptotic complexity of computational problems and in providing software engineers with more powerful tools.

Acknowledgements

The author gratefully acknowledges helpful conversations with David Jefferson and Andrew and Frances Yao. The contributions of James Saxe to the author's understanding of decomposable problems (and thereby to the presentation of this introductory paper) are too numerous to mention; a simple "thanks" will have to suffice.

References

Bentley, J. L., D. Detig, L. Guibas, and J. B. Saxe [1978]. An optimal data structure for minimal-storage dynamic member searching, in preparation.

Bentley, J. L. and J. B. Saxe. Decomposable searching problems, in preparation.

Bentley, J. L. and M. Shaw [1978]. A class of correct and efficient transformations for converting data structures from static to dynamic, in preparation.

Bentley, J. L. and M. I. Shamos [1977]. "A problem in multivariate statistics: algorithm, data structure, and applications", *Proceedings of the Fifteenth Allerton Conference on Communication, Control and Computing*, pp. 193-201.

Dobkin, D. and R. J. Lipton [1976]. "Multidimensional searching problems", *SIAM Journal of Computing* 5, pp. 181-186.

Knuth, D. E. [1973]. *The Art of Computer Programming, volume 3: Sorting and Searching,*

Addison-Wesley, Reading, Mass.

Lipton, R. J. and R. E. Tarjan [1977]. "Applications of a planar separator theorem", *Eighteenth Symposium on the Foundations of Computer Science*, pp. 162-170.

Lueker, G. S. [1978]. A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems, UC Irvine Technical Report.

Rivest, R. L. [1974]. Analysis of associative retrieval algorithms, Stanford Computer Science Department Report STAN-CS-74-415, 102 pp.

Tarjan, R. E. [1978]. "Complexity of combinatorial algorithms", *SIAM Review 20*, 3, pp. 457-491 (July 1978).

Vuillemin, J. [1978]. "A data structure for manipulating priority queues," *Comm. of the ACM, 21*, 4, pp. 309-315 (April 1978).

Yao, A. C. [1977]. "Fast algorithms for finding minimum spanning trees in k dimensions", *Proceedings of the Fifteenth Allerton Conference on Communication, Control and Computing*, pp. 553-556.

Yao, A. C., F. F. Yao, and J. L. Bentley [1978]. On computing the rank function of a set of vectors, to appear.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-78-145	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DECOMPOSABLE SEARCHING PROBLEMS		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jon Louis Bentley		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept Schenley Park, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE October 1978 13. NUMBER OF PAGES 15
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) SAME AS ABOVE		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)