On Some Generalizations of Binary Search

David Dobkin and R. J. Lipton

Research Report #25

This paper will be presented at the Sixth Annual ACM Symposium on Theory Computing, to be held in Seattle, Washington, April 1974

February 1974

On Some Generalizations of Binary Search

David Dobkin and R.J. Lipton
Department of Computer Science
Yale University
New Haven, Connecticut 06520

Abstract

Classic binary search is extended to multidimensional search problems. These new search methods can efficiently solve several important problems of computer science. Applications of these results to an open problem in the theory of computation are discussed yielding new insight into the Lba problem.

I. Introduction

One of the most basic operations performed on a computer is searching. A search is used to decide whether or not a given word is in a given collection of words. Since many searches are usually performed on a given collection, it is generally worthwhile to organize the collection so that searching is efficient. The organization of the collection into a more desirable form, called preprocessing, can be assumed to be done at no cost relative to the cost of numerous searches. Thus, for example, dictionaries are arranged in alphabetical order; this organization allows for fast location of a word's definition.

One of the most popular methods of searching is the binary search method. In his encyclopediac work on sorting and searching, Knuth [4] gives to Inakibit-Anu of Uruk for discovering this method in 200 B.C. For the purposes of this paper we can view binary search as follows:

Data: A collection of m points on a line.
Query: Given a point, does it equal any of the m points?

Binary search can answer this query in $\lfloor \log m \rfloor$ * + 1 steps where a step is a comparison. Note that the preprocessing needed is a sort which requires $O(m \log_2 m)$ steps. For the algorithms under consideration here, we will define a step in an algorithm as a comparison of two scalars or the determination of whether a point in 2-dimensional Euclidean space lies on, above or below a given line. For notational simplicity we will define $g(m)$ as the number of steps necessary to perform a search through a set of m objects. Thus, $g(m) = \lfloor \log m \rfloor + 1$.

This paper generalizes binary search to higher dimensional problems. Throughout it is assumed that data can be organized in any manner desired at no cost. Thus, our cost criterion for evaluating the relative efficiencies of searching algorithms will be the number of steps required to make a single query into the reorganized data.

The search problems considered are specified by a collection of data and a class of queries. These problems include:

Example 1: Data: A set of m lines in the plane.
Query: Given a point, does it lie on any line?

Example 2: Data: A set of m regions in the plane.
Query: Given a point, in which region does it lie?

Example 3: Data: A set of m points in the plane.
Query: Given a new point, to which of the original points is it closest

Example 4: Data: A set of m lines in n-dimensional space.
Query: Given a point, does it lie on any line?

Example 5: Data: A set of m k-dimensional objects in n-dimensional space.
Query: Given a point, does it lie on any of the objects?

Example 6: Data: A set of m hyperplanes (n-1-dimensional objects) in n-dimensional space.

*Throughout this paper all logarithms are taken base 2.

Query: Given a point does it lie on any hyperplane?

These examples form the basis for some important problems in diverse areas of computer science. Examples 1, 2 and 3 are fundamental to certain operations in computer graphics [5] and secondary searching [4]. In particular, example 3 is a reformulation of an important problem discussed by Knuth [4] concerning information retrieval. Examples 4, 5 and 6 are generalizations of the widely studied knapsack problem [2]. More exactly, the knapsack problem is a special case of example 6: Given the integers $x_1,\ldots,x_n,b$ there are 0-1 valued numbers $a_1,\ldots,a_n$ such that

$$\sum_{p=1}^{n} a_p x_p = b$$

if and only if the point $\left(\frac{x_1}{b},\ldots,\frac{x_n}{b}\right)$ lies on one of the hyperplanes $H_i(v)=1$ where if the binary expansion of i is

$$i = \sum_{j=1}^{n} c_j 2^{j-1}$$

with $c_j = 1$ or 0, the $i^{\underline{th}}$ hyperplane is given by

$$H_i(v_1,\ldots,v_n) = \sum_{j=1}^{n} c_j v_j$$

Thus, for the knapsack problem a total of 2^n-1 such hyperplanes exist and we ask if a given point in n-dimensional space lies on any of these objects of dimension n-1. By taking intersections of hyperplanes, we can generate objects of dimension n-N in n-dimensional space such that the knapsack problem has N (or more) solutions if and only if the point $\left(\frac{x_1}{b},\ldots,\frac{x_n}{b}\right)$ lies on one of these objects. Examples 4 and 5 correspond to generalizations of the knapsack problem where we ask for many solutions rather than a single solution.

II. The Algorithm

All of our first algorithms are extensions of a fast algorithm that computes the predicate:

$$\exists\, 1 \le i \le m[(x,y) \text{ is on } L_i]$$

where $L_1,\ldots,L_m$ are lines and (x,y) a point in 2-dimensional Euclidean space (E^2). Therefore, we will sketch the proof that this predicate can be computed in $O(\log m)$ steps.

__Theorem 1:__ For any set of lines $L_1,\ldots,L_m$ in the plane, there is an algorithm that computes

$$\exists\, 1 \le i \le m[(x,y) \text{ is on } L_i]$$

in $3g(m)$ steps.

__Proof:__ Let p_i $(1 \le i \le n)$ be the projections of the intersection points formed by $L_1,\ldots,L_m$; moreover, assume that $p_1 <\ldots< p_n$. Define the relation $<_i$ $(1 \le i \le n)$ as follows: $L_j <_i L_k$ if and only if

$$\forall x \in E^1 [\text{if } p_i \leqq x \leqq p_{i+1}, \text{ then } L_j(x) \leqq L_k(x)].$$

(Note, $L(x)$ is equal to the y such that $(x,y) \in L$). By a simple continuity argument it follows that each $<_i$ is a linear ordering on the line $L_1,\ldots,L_m$ (see fig. 1). Let $\pi(i,1),\ldots,\pi(i,m)$ be such that

$$L_{\pi(i,1)} <_i L_{\pi(i,2)} <_i \cdots <_i L_{\pi(i,m)}$$

The complete algorithm (see figure 1) then operates on the point $(x,y) \in E^2$ as follows:

(A) Find an i such that $p_i \leqq x \leqq p_{i+1}$

(B) Do a binary search on the m objects $L_{\pi(i,1)},\ldots,L_{\pi(i,m)}$.

Since this algorithm consists of a binary search into a set of at most $\frac{m(m-1)}{2}$ objects (the points of intersection of the lines, i.e. $\{p_i\}$) and a binary search into a set of m objects (the lines $L_{\pi(i,1)},\ldots,L_{\pi(i,m)}$), it is clear the total number of steps is at most $g(m) + g\,\frac{m(m-1)}{2}$ and since g is a monotonically increasing function and $g(m^2) \le 2\,g(m)$, this quantity is at most $3\,g(m)$.

Before studying applications of this algorithm to the problems mentioned above, it is worthwhile to examine its structure in more detail. Basically, the algorithm works on two sets of objects which have been generated from the original set of lines during preprocessing. These sets are the set of projections of the intersection points (i.e. $\{p_i\}$), and the set of permutations giving the order

of the lines between p_i and p_{i+1} for each i, $0 \le i \le n$. The set of points can be ordered and therefore binary search can be used to determine where a new point lies with respect to these points and, since the lines are non-intersecting in the interval being considered, binary search can be applied to determine if the given point lies on any of the lines. Moreover, it is clear that the algorithm not only determines whether the point lies on one of the given lines, but also determines in which region of the plane the point lies. Thus, we have,

Corollary: Given a set of regions formed by m lines in the plane, we can determine in $3g(m)$ steps in which region a given point lies.

Although it may be possible to find other algorithms which compute the given predicate in less than linear time, it seems reasonable to conjecture that no algorithm which does not generate different objects from the original set can compute the predicate in less than a linear number of steps.

We study next some applications of the algorithm given above to examples 4 and 6 of section 1.

Theorem 2: Suppose that $L_1,\dots,L_m$ are m lines in n-dimensional Euclidean space $(n \ge 2)$. Then it is possible to determine if a point is on one of the lines in $(n+1)g(m)$ steps.

Proof: The proof is by induction on n. For $n = 2$, the theorem reduces to theorem 1. Now suppose that $n > 2$. Let x be the given point and project the lines and x onto a hyperplane of dimension $n-1$. Since $n > 2$, the result of this operation is m lines $L_1',\dots,L_m'$ (or points in degenerate cases) and a point x'. Furthermore, if x lies on L_i, then x' lies on L_i'. By the induction hypothesis, we can decide if x' lies on any line L_i' in $ng(m)$ steps. If x' doesn't lie on any L_i' then x does not lie on any L_i. And, if x' lies on lines $\{L_{i}',\dots,L_{i_k}'\}$, then in a binary search of $\{L_i,\dots,L_{i_k}\}$ ordered with respect to the projected parameter we can determine if x lies on any of the lines. Since this search requires at most $g(m)$ steps, the theorem is proved. □

Theorem 3: Suppose that $H_1,\dots,H_m$ are (n-1)-dimensional hyperplanes in $E^n (n \ge 2)$. Then, we can determine which region a point lies in or which hyperplane a point lies on in less than $(3\cdot 2^{n-2} + (n-2))g(m)$ steps.

Sketch of Proof: Let $f(n,m)$ be the time required to do the search. It is clear that $f(2,m) = 3g(m)$ and we will show that $f(n,m) \le f(n-1, m^2) + g(m)$. We proceed by forming the projections onto some fixed hyperplane of all the intersections $H_i \cap H_j$ $(1 \le i < j \le m)$. These hyperplanes form hyperplanes $J_1',\dots,J_k'$ $(k \le m^2)$ which are (n-2)-dimensional objects in an (n-1) dimensional space. If the point x projects onto x', we can by the inductive hypothesis determine in $f(n-1, m^2)$ steps in which region of n-1 dimensional space x' lies. Suppose that x' lies in region R. Then, since no $H_i \cap H_j$ projects onto R, the hyperplanes $H_1,\dots,H_m$ are linearly ordered at that part. So in $g(m)$ steps we can determine if x lies on any hyperplane. If x' lies on a hyperplane J_i', then by a search of hyperplanes requiring less than $g(m)$ steps, we can determine on which hyperplane x lies. This completes the proof that $f(n,m) \le f(n-1,m^2) + g(m)$. Applying the recursion yields $f(n,m) \le f(n-k,m^{2^k}) + kg(m)$ or $f(n,m) \le f(2,m^{2^{n-2}}) + (n-2)g(m) = (3\cdot 2^{n-2} + (n-2))g(m)$. □

III. Relation to Complete Problems

In light of the work of Cook [1], Karp [3], and Horowitz and Sahni [2], the results of the previous section may be surprising. Their research leads to solid evidence that no polynomial time algorithm can exist for solving the knapsack problem. While we have not disproved this conjecture, the results of the previous section yield contradictions to some extensions of this conjecture.

Before proceeding it is of value to reformulate the statement that there exists a polynomial time algorithm for solving the knapsack problem. This statement is clearly equivalent to

(1) $\exists p \exists \mathcal{Q} \forall n$ [algorithm $\mathcal{Q}$ solves the n-dimensional knapsack problem in $p(n)$ time].

Note, in order for (1) to be true, there must be an algorithm that operates uniformly for all n. Allowing preprocessing leads to

(2) $\exists p \forall n \exists \mathcal{Q}$ [algorithm $\mathcal{Q}$ solves the n-dimensional knapsack

problem in $p(n)$ time].

Clearly, (1) implies (2); however, (2) may not imply (1).

Our interest in the two versions (1) and (2) is twofold: First, the two versions point out a perhaps unnoticed difficulty in attempting to prove that $P \neq NP$. Second, the two versions demonstrate a possibly unnoticed difference among the complete problems.

One possible approach to proving that $P \neq NP$ is based on (2): select a polynomial p, then show that every algorithm for the n-dimensional knapsack must take at least $p(n)+1$ steps for some n. The results of the previous sections show that this may be false. More exactly, let N be an integer and consider the problem of the knapsack problem with at least $n-N$ solutions. The analogs of (1) and (2) are:

(1') $\exists p \exists \mathcal{Q} \forall n \geq N$ [algorithm $\mathcal{Q}$ solves the n-dimensional knapsack problem with at least N solutions in $p(n)$ time]

and

(2') $\exists p \forall n \exists \mathcal{Q}$ [algorithm $\mathcal{Q}$ solves the n-dimensional knapsack problem with at least N solutions in $p(n)$ time].

Then we can prove that (2') is true.

<u>Theorem 4:</u> (2') is true.

<u>Proof</u>: A knapsack problem of dimension n has a solution if the predicate

$$\exists i \; 1 \leq i \leq 2^n [(x_1,\ldots,x_n) \text{ is on } L_i]$$

where each L_i is an n-1 dimensional hyperplane. Similarly, we can generate n-k dimensional hyperplanes consisting of the intersection of k hyperplanes such that the knapsack problem has k solutions if and only if the point $(x_1,\ldots,x_n)$ lies on one of these hyperplanes. By a generalization of Theorem 3, we can compute this predicate in time exponential in $n-k$ and polynomial in n. □

Therefore, the outlined approach to proving $P \neq NP$ may be doomed to fail; in any case it does fail when applied to the N solution knapsack problem. This immediately leads to the question:

Is the N knapsack problem, for some N, a complete problem?

If this is true, then we would have a complete problem which has for each n a polynomial algorithm. Therefore, any attempt to prove that this problem has no polynomial algorithm must really exploit the non uniformity, i.e., it must attempt to prove $\sim$(1) and not $\sim$(2).

The above discussion also leads to a difference between the known complete problems. Consider instead of the knapsack problem the chromatic number problem. The analogies of (1) and (2) are now:

(1") $\exists p \exists \mathcal{Q} \forall n$ [algorithm $\mathcal{Q}$ solves the problem of chromatic number on n node graphs in $p(n)$ time]

(2") $\exists p \forall n \exists \mathcal{Q}$ [algorithm $\mathcal{Q}$ solves the problem of chromatic number on n node graphs in $p(n)$ time].

While (1) is open and (2') is true but nontrivial, (2") is easily seen to be true.

<u>Theorem 5:</u> (2") is true.

<u>Proof:</u> There exist 2^{n^2} graphs on n nodes, and for each graph a chromatic number can be found. We form a list of all graphs arranged in some order with their chromatic numbers. All of this work can be done before considering any data. Then, given a graph, in one probe of this list we can determine the graph's chromatic number and in one comparison we can determine if it is k-colorable. □

Therefore it seems possible that there are two kinds of complete problems: those that with preprocessing have trivial polynomial time algorithm and those that do not. Essentially the infinite precision allowed in the knapsack problem has no counterpart in the chromatic number problem. In Figure II, we have classified the complete problems presented by Karp according to this distinction. It appears that we may be able to extend Karp's notion of reducibility to obtain a finer division of this class of problems to reflect these differences. A set A is said to be polynomial reducible to a set B (in the sense of Karp) if there exists a polynomial computible function f such that $f(x) \in B \Longleftrightarrow x \in A$. By adding conditions to this definition so that A and B are of the same type in the sense of Figure II, it appears that a more realistic notion of reducibility, in the sense of the differences between statements (1) and (2) results.

Acknowledgements

We would like to thank Professor Larry Snyder for some helpful conversations and Mr. Michael Shamos

for comments on this manuscript.

References

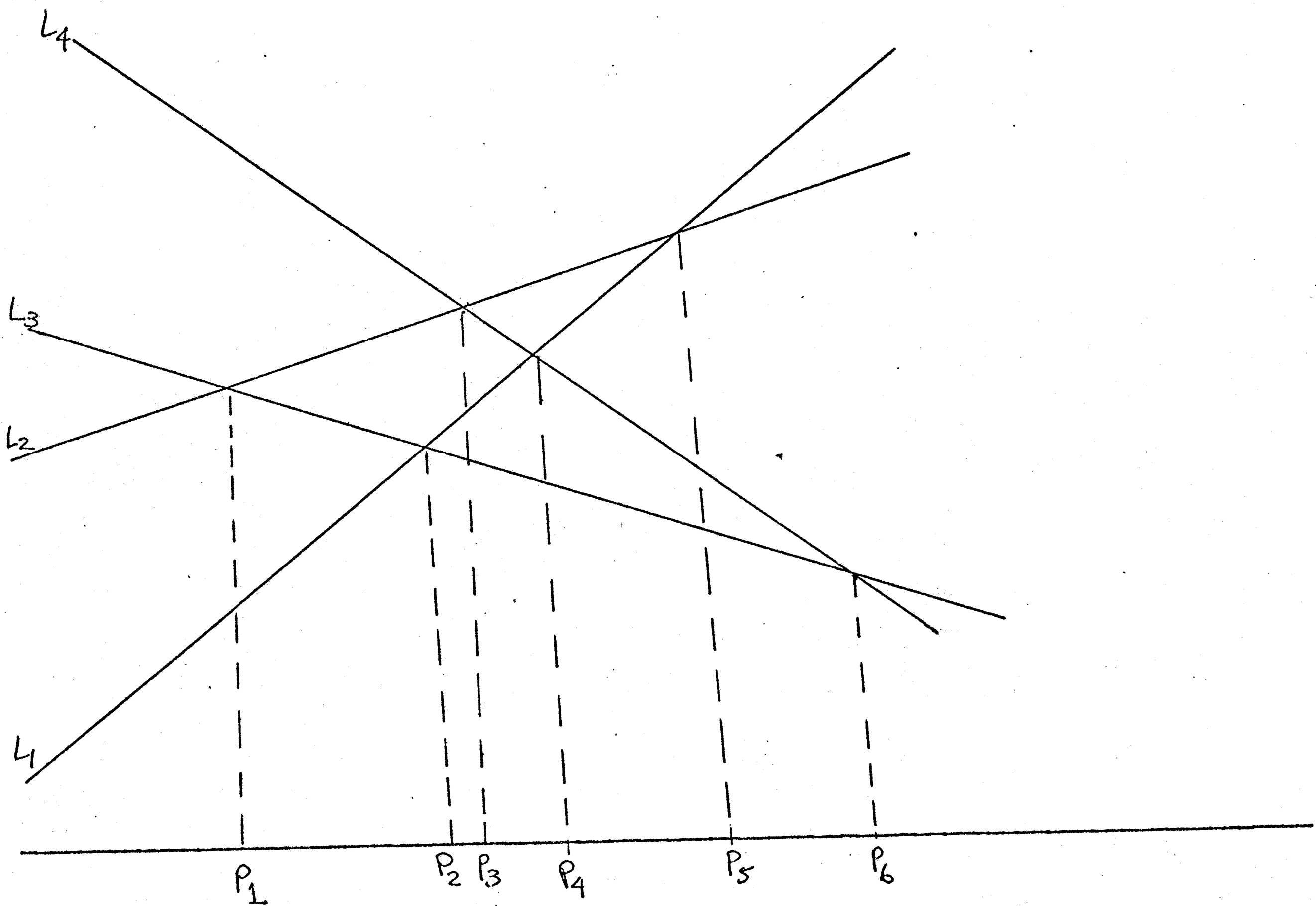
[1] S.A. Cook. The complexity of theorem proving procedures. Conference Record of the Third ACM Symposium on the Theory of Computing, Shaker Heights, Ohio May 1971.

[2] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. Cornell University Computer Science Technical Report 72-134, July 1972. (To appear in JACM.)

[3] R. Karp. Reducibility among combinatorial problems. Complexity of Computer Computations, edited by R. Miller and J. Thatcher, Plenum Press, 1972.

[4] D. Knuth. The Art of Computer Programming Volume 3: Sorting and Searching, Addison-Wesley, 1973.

[5] W. Newman and R. Sproull. Principles of Interactive Computer Graphics, McGraw-Hill, 1973.



<u>Figure I</u>: The permutations we need to consider here are,

i) in the interval (∞, p_1)	(1,2,3,4)
ii) in the interval (p_1, p_2)	(1,3,2,4)
iii) in the interval (p_2, p_3)	(3,1,2,4)
iv) in the interval (p_3, p_4)	(3,1,4,2)
v) in the interval (p_4, p_5)	(3,4,1,2)
vi) in the interval (p_5, p_6)	(3,4,2,1)
vii) in the interval (p_6, ∞)	(4,3,2,1)

Knapsack
Steiner Tree
Job Sequencing
Partition
Max Cut
0-1 Integer Programming

(a)

Satisfiability
Satisfiability with at most 3 literals per clause
Chromatic Number
Exact Cover
Clique Cover
3-dimensional matching
Hitting Set
Clique
Set Packing
Node Cover
Feedback Node Set
Feedback Arc Set
Directed Hamiltonian Circuit
Undirected Hamiltonian Circuit
Set Covering

(b)

Figure II: (a) Infinite precision problems for which statement (2) is nontrivial
(b) Graphical problems for which statement (2) is easily verified

(Note: All notation is as in Karp [3]).