ON THE COMPLEXITY OF VECTOR SEARCHING

D. S. Hirschberg

June, 1978

TECHNICAL REPORT #7807

Research supported by NSF grant MCS-76-3933.

ON THE COMPLEXITY OF VECTOR SEARCHING

D. S. Hirschberg

Rice University

June, 1978

Abstract and Introduction:

The vector searching problem is, given k-vector A (a k-vector is a vector that has k components, over the integers) and given a set B of n distinct k-vectors, to determine whether or not A is a member of set B. Comparisons between components yielding "greater than-equal-less than" results are permitted. It is shown that if the vectors in B are unordered then nk comparisons are necessary and sufficient. In the case when the vectors in B are ordered, it is shown that $\lfloor \log n \rfloor + k$ comparisons are necessary and, for $n \geq 4k$, $k\lceil \log(n/k) \rceil + 2k - 1$ comparisons are sufficient.

Key Words and Phrases: searching, vector, lower bounds, complexity

---

Research supported by NSF grant MCS-76-3933. Author's current address: Department of Electrical Engineering, Rice University, Houston Texas 77001.

Searching an unordered set

We first consider the case in which the vectors in $\bar{B}$ are <u>not</u> ordered. In this case, an upper bound of nk comparisons can be easily demonstrated. We show that nk is also a lower bound and thus nk is the complexity of this vector searching problem.

Let $A = a_1 \ldots a_k$; let $B = b_1 \ldots b_k$ be a typical vector in $\bar{B}$; and let b_t be a typical component of B.

We shall use a dynamic adversary to construct an oracle for distinguishing a path, P_*, in each decision tree that solves this problem. We shall show that each path P_* has length at least nk and thus, in the worst case, any algorithm solving this problem requires at least nk comparisons. For a review of the use of oracles to derive lower bounds, the reader is referred to [1,3,5].

Relative to the sequence of comparisons (and results) that have been performed (along path P_*), we make the following definitions for each component b_t in each vector B in $\bar{B}$.

$\langle b_t \rangle$ is the set of vectors in $\{A\} \cup \bar{B}$ whose t-components must equal b_t.

$\langle B \rangle$ is the set of vectors in $\{A\} \cup \bar{B} - \{B\}$ that might still be equal to B. We shall call $\langle B \rangle$ the set of vectors that

are viable for B.

$\langle \neg B \rangle$ is the set of vectors in $\{A\} \cup \mathbb{B} - \{B\}$ that cannot equal B.

Note that $\langle B \rangle$, $\langle \neg B \rangle$, and $\{B\}$ partition $\{A\} \cup \mathbb{B}$.

Each component b_t in B can be in one of four states:

1. E b_t must equal a_t.
2. F b_t has been "fixed" at some integer value f that is known to be less than a_t.
3. K there are two possibilities

   (1) $\langle B \rangle \neq \emptyset$ (some other vector might equal B)

   $\langle b_t \rangle \bigwedge \langle B \rangle = \emptyset$

   $\forall i \neq t \ \langle b_i \rangle \bigwedge \langle B \rangle \neq \emptyset$

   or (2) $\langle B \rangle = \emptyset$ (no other vector can equal B)

   $\exists$ unique $B' \in \langle b_t \rangle$ s.t. $\langle B' \rangle \neq \emptyset$

   and for which (1) holds.

4. N b_t is not in any of the above three states.

Each component a_t in A will be said to be in state E.

Initially, all components of all vectors in $\mathbb{B}$ are in state N and all components of vector A are in state E. Variable f is initialized to zero.

The oracle selects an answer to return to a comparison between components by determining which states these components are in, and then looking up the result in the

table given below.

There may be changes of state as a consequence of this comparison. These are also indicated in the table.

Define $\langle b_t \rangle_t$ to be the set of t-components of the vectors in $\langle b_t \rangle$ (this set includes b_t). We call $\langle b_t \rangle_t$ the equivalence class of b_t. The following table is used by the oracle for comparing $b_t : b'_t$ where b'_t is the t-component of some vector in $\{A\} \cup B$ other than B.

b_t	b'_t	result	new state of $\langle b_t \rangle_t$
E	E	=	
F	E	<	
	F	as indicated by "fix"	
N	E	=	E *
	F	>	
	N	=	*
K	E	<	F **
	F	>	
	K	= if $b'_t \in \langle b_t \rangle_t$	K
		< otherwise	F **
	N	<	F **

Notes:

* may cause one component b_x in B and all other elements in the equivalence class of b_x, $\langle b_x \rangle_x$, to change from state

N to state K.

** f is initialized at 0. Whenever a change of state to F occurs, say by b_t, then f is incremented by one and the value of b_t (and also the rest of $\langle b_t \rangle_t$) is <u>fixed</u> at f. We can assume that the value of each component a_t of A is at least nk. Thus, "fixing" b_t at value f effectively forces $b_t < a_t$. Also, note that this effectively forces b_t to be less than the value of all components that are currently in states E, K, or N. This may also cause some other components to change state from K to N.

It is easy to see that this change-of-state table is consistent with the definitions of the states. The only difficulty in observation is the uniqueness of B' in part (2) of state K.

If a comparison $b_t : b'_t$ results in "less than" then b_t enters state F and the vector B (of which b_t is a component) cannot be equal to any other vector ($\langle B \rangle = \emptyset$) since b_t will be fixed at some value f smaller than a_t. No element b_t will ever be found to have value greater than a_t. No element b_t will ever be restricted from being equal to a_t unless b_t enters state F.

Consider an element b_t that enters state K by condition (1). Thus, $\langle B \rangle \neq \emptyset$ and $\langle b_t \rangle \cap \langle B \rangle = \emptyset$. That is, all B' in $\langle b_t \rangle$

have been restricted from being equal to B. This occurs only if each B' has a component that has been involved in a < comparison. But once such a comparison occurs, $\langle B' \rangle$ is empty. Therefore, $\forall B' \neq B \in \langle b_t \rangle$, $\langle B' \rangle = \emptyset$.

Consider an element b'_t that enters state K by condition (2); at least one of the vectors in $\langle b'_t \rangle$ will satisfy (1). Let B be one such vector, $\langle B \rangle \neq \emptyset$. But, from above, all vectors B" in $\langle b_t \rangle$ (which equals $\langle b'_t \rangle$) other than B satisfy $\langle B'' \rangle = \emptyset$. Thus B is the unique vector in $\langle b'_t \rangle$ for which $\langle B \rangle \neq \emptyset$. □

The definition of state K and the results of comparisons with components in state K are such that no two vectors in $\bar{B}$ will ever be found to be equal. A component b_t enters state K when it is the only component of B that has not made an "equal" comparison with a component of any vector that might be equal to B. A component, b_t, in state K will not be found to be equal to any component other than those in $\langle b_t \rangle_t$, i.e. those components that are already known to be equal to b_t.

Thus we have shown that the change-of-state table is consistent with the definitions of the states and that the oracle will not find two vectors to be equal. We now show that nk comparisons are required in order to ensure that no vector in $\bar{B}$ can be equal to vector A. Path P_* must be of

sufficient length to distinguish between a "match" and a "no match" solution in order for the decision tree to solve the vector search problem. Thus nk will be a lower bound on the complexity of the vector search problem.

No component b_t can ever be restricted from being equal to a_t <u>unless</u> b_t is in state F. No vector B can be restricted from being equal to A unless at least one of the components in B is in state F. This cannot occur unless at least one of the components has previously entered state K. If $\langle B\rangle \neq \emptyset$ (and thus no component of B has entered state F) and a component of B is in state K then it must be that the other k-1 components of B have had at least one comparison each (resulting in "equal") with viable vectors. When a component b_t enters state F, causing $\langle B\rangle = \emptyset$, the rest of $\langle b_t\rangle_t$ also enters state F but all the other members B' in $\langle b_t\rangle$ already satisfy $\langle B'\rangle = \emptyset$. Thus the k-1 comparisons that we are counting toward vector B attaining $\langle B\rangle = \emptyset$ are not counted toward any other vector's similar count (only viable vectors qualify). Once $\langle B\rangle = \emptyset$, the k-1 comparisons that were "counted" do not contribute to any other vector's "count" since B is no longer viable.

If T(n) is the number of comparisons required to get at least one component of each of n vectors into state F, then $T(n) = (k-1) + 1 + T(n-1)$. The (k-1) is the "count" of

vector B, the 1 is the comparison causing b_t to enter state F, and T(n-1) is the number of comparisons required to get the other n-1 vectors into a similar situation (have a component enter state F). The comparisons with components in B do not contribute to the T(n-1) comparisons as noted above and thus these three sets of comparisons are disjoint.

T(n) = k + T(n-1) with boundary condition T(0)=0 is solved by T(n)=nk. □

Searching an ordered set

We now consider the case in which preprocessing of the set $\bar{B}$ is permitted. That is, we can assume that $\bar{B}$ is in some prearranged order, such as lexicographic order.

In the discussions that follow, all logarithms are assumed to be base 2.

A lower bound of $\lfloor \log n \rfloor + k$ comparisons can be seen by observing that $\lfloor \log n \rfloor + 1$ comparisons are required to determine if there is any vector having the correct value of one component, and k-1 comparisons are required to verify the agreement of the remaining components.

The oracle for distinguishing a path (which will be of length at least $\lfloor \log n \rfloor + k$) in each decision tree that solves this problem is as follows.

Initially, define low=1 and high=n.

Let the next comparison presented to the oracle be $a_j : b_{ij}$.

```
mid <- (low+high)/2
If low≠high then:
     if i<low then return >
     if low≤i≤mid then [low <- i+1; return > ]
     if mid<i≤high then [high <- i-1; return < ]
     if high<i then return <
Else (low=high) return =
```

low will not equal high until after at least $\lfloor \log n \rfloor$ comparisons. During these comparisons, vector A could equal any of the vectors $B_{low} \ldots B_{high}$.

When low=high, until all components of B_{low} have been compared with A, B_{low} may equal A but it is also possible that one component of B_{low} will be less than the corresponding component of A and, in that case, it is possible that none of the vectors in B are equal to A.

Thus $\lfloor \log n \rfloor + k$ comparisons are necessary to solve this problem.

In the above analysis, we assumed that all comparisons are between a component of A and the corresponding component of a vector in B. It is straightforward to generalize and allow comparisons between components of vectors both of which are elements of B.

Having demonstrated a lower bound, we now consider upper bounds for this problem.

We present and analyze two algorithms that solve the ordered set problem and then combine them to obtain an algorithm that is faster than both.

The first algorithm is an example of binary search. Let $\bar{B} = \{B_1, B_2, \ldots, B_n\}$ and let $B_i = b_{i1} \ldots b_{ik}$. Proceed comparing the components of A with those of the central vector, i.e. compare a_h with b_{jh} for $h=1,2,\ldots$ where $j = \lfloor (n+1)/2 \rfloor$. If all comparisons result in "equal" then $A=B_j$. Otherwise, if at some point we get a "less than" result then $A \neq B_j$ and we can restrict our attention to $\bar{B}' = \{B_1, \ldots, B_{j-1}\}$. Similarly, if we get a "greater than" result, then we can restrict our attention to $\bar{B}' = \{B_{j+1}, \ldots, B_n\}$. In the worst case, we will require k comparisons in each of $1 + \lfloor \log n \rfloor$ iterations for a total of $k + k\lfloor \log n \rfloor$ comparisons. This is equivalent to the result in [2].

The second algorithm uses linear search and is as follows:

```
j <- 1
h <- 1
while j≤n AND h≤k
do compare a_h :b_jh
   if = then h <- h + 1
   else if > then j <- j + 1
   else [print 'NO SOLUTION'; stop]
od
if j>n then [print 'NO SOLUTION'; stop]
```

```
final:   if a_i=b_ji  for all i ∈ {1,2,...,k-1}
         then print j     ;comment A=B_j
         else print 'NO SOLUTION'
         stop
```

The algorithm finds the first (lowest indexed) vector, B_j, that matches A in the hth component. All lower indexed vectors are not considered further. All other vectors are assumed to match in this component. The algorithm then iterates on the (h+1)st component. This part of the algorithm will make at most n+k comparisons (each iteration increments either j with upper limit n, or h with upper limit k). If we succeed in matching all k components in this manner then the vector, B_j, that is found will be equal to A if all assumptions made earlier apply to B_j. However, if B_j disagrees with A in any component h' then A does not appear in $\bar{B}$ since all $j'<j$ have been eliminated, $B_j \neq A$ (assumed here) and for all $j''>j$, $B_{j''}$ will disagree with A among the first h' components since $\bar{B}$ is in lexicographic order. The final phase of the algorithm, in which the components of B_j (which were assumed to agree with A) are compared with A, requires at most k-1 comparisons for a total of at most n+2k-1 comparisons.

We note that the linear search algorithm's mirror image also works. That is, we can start with j=n and decrement j, being careful to interchange the <'s and >'s. We can, as an

initial improvement, compare A with the central vector in B rather than with B_1 or B_n and, at the first "less than" or "greater than" result, continue with the linear search algorithm applied to only half of the original set B. This leads to an algorithm that requires, in the worst case, only $\lfloor n/2 \rfloor + 2k-1$ comparisons. We call this improved algorithm the modified linear search algorithm.

We can make further improvements by deciding, at the time that a "less than" or "greater than" result is obtained, whether to continue in the style of the binary or the modified linear search algorithm depending upon which will lead to fewer comparisons in the worst case. If, after making h comparisons (resulting in "equal") along vector B_j within feasible set B of cardinality n, we make a comparison resulting in "less than" or "greater than" then continuing with the linear search algorithm requires, in the worst case, at most $\lfloor n/2 \rfloor + 2k-1-h$ additional comparisons. If, however, we decide to proceed with comparisons in a new vector within B and thus follow the binary search algorithm or follow the modified linear search algorithm on a feasible set of half the size, then we will have upper bounds of $k \lfloor \log n \rfloor$ and $\lfloor n/4 \rfloor + 2k-1$ additional comparisons respectively. We should continue with linear search only if

$$\lfloor n/2 \rfloor + 2k-1-h < \min \{ \lfloor n/4 \rfloor + 2k-1, k \lfloor \log n \rfloor \}$$

which holds only if $n<4h$. For particular values of k, we can solve this inequality to gain further restrictions. For example, if k=3 then we should continue with linear search only if n=4 and h=2 or n=5 and h=2.

Let $T(n,k)$ be the minimum number of comparisons required for the ordered vector search problem when $\bar{B}$ consists of n k-vectors. Then, for $n \leq 4k$, $T(n,k) \leq \lfloor n/2 \rfloor + 2k-1$.

For $n=k2^r$, $T(n,k) \leq k + T(n/2,k) \leq (r-2)k + T(4k,k) \leq (r+2)k - 1$.

For $k2^{r-1}<n<k2^r$, $T(n,k) \leq (r-2)k + T(4k-1,k) \leq (r+2)k - 2$. Note that in both cases, $r = \lceil \log(n/k) \rceil$.

Algorithm VECTOR_SEARCH incorporates the modifications mentioned above.

```
          VECTOR_SEARCH(A, B̄, n, k)
          low ←- 1
          high ←- n
binary:   binsearch ←- TRUE
          while binsearch AND low≤high
          do j ←- (low + high)/2
             compare a_1:b_j1
             if > then high ←- j-1
             else if < then low ←- j+1
             else binsearch ←- FALSE
          od
          if low>high then [print 'NO SOLUTION'; stop]
          n ←- high - low + 1
modlin:   h ←- 2
```

```
            while h≤k
            do compare a_h :b_jh
               if = then h <- h + 1
               else if > then  if n≥4*h
                               then [high <- j-1; goto binary]
                               else goto linear
               else if < then  if n≥4*h
                               then [low <- j+1; goto binary]
                               else goto linear2
            od
            print j    ;comment A=B_j
            stop
linear:     while j≤high AND h≤k
            do compare a_h :b_jh
               if = then h <- h + 1
               else if > then j <- j + 1
               else [print 'NO SOLUTION'; stop]
            od
            if j>high then [print 'NO SOLUTION'; stop]
final:      if a_i =b_ji  for all i ∈ {1,2,...,k-1}
            then print j    ;comment A=B_j
            else print 'NO SOLUTION'
            stop
linear2:    while j≥low AND h≤k
            do compare a_h :b_jh
               if = then h <- h + 1
               else if < then j <- j - 1
               else [print 'NO SOLUTION'; stop]
            od
            if j<low then [print 'NO SOLUTION'; stop]
            goto final
```

ACKNOWLEDGEMENT. The author wishes to thank C. K. Wong for suggesting this problem.

References

1. A.V. Aho, D.S. Hirschberg, and J.D. Ullman. Bounds on the Complexity of the Longest Common Subsequence Problem. Journal ACM **23**:1 (Jan. 1976), pp.1-12.

2. D. Dobkin and R.J. Lipton. Multidimensional Searching Problems. SIAM J.Computing **5**:2 (June 1976), pp.181-186.

3. D.E. Knuth the Art of Computer Programming, Vol. 3. Addison-Wesley, 1973, pp.200-204.

4. V.V Raghavan and C.T. Yu. A note on a multidimensional searching problem. IPL **6**:4 (Aug. 1977), pp.133-135.

5. E.M. Reingold. On the Optimality of some Set Algorithms. Journal ACM **19**:4 (Oct. 1972), pp.649-659.