A Data Structure and an Algorithm for the Nearest Point Problem

IRAJ KALANTARI AND GERARD McDONALD

Abstract—**In this paper we present a tree structure for storing points from a normed space whose norm is effectively computable. We then give an algorithm for finding the nearest point from the tree to a given query point. Our algorithm searches the tree and uses the triangle inequality to eliminate searching of the entirety of some branches of the tree whenever a certain predicate is satisfied. Our data structure uses 0(n) for storage. Empirical data which we have gathered suggest that the expected complexity for preprocessing and the search time are, respectively, 0(nlogn) and 0(logn).**

Index Terms—**Algorithm, computational time, data structure, nearest point problem, preprocessing, storage binary search, trees, triangle inequality.**

I. Introduction

The *closest point problem* (also referred to as the post office search; see Knuth [4]) calls for organizing a set S of points $\{x_1, \cdots, x_n\}$ in $\mathbb{R}^k$ into a data structure and producing an algorithm for finding the nearest point in S to a new point z, *the query point*. This problem arises in many areas including data analysis, information retrieval, pattern recognition, statistics, image processing, communicating, and transport networks.

The efficiency of a solution can be measured in terms of three quantities: the *preprocessing time* P(n) (the number of operations required to construct the data structure); the amount of *storage* required by the data structure S(n); and the *search time* Q(n) (the number of operations required to locate the nearest point). In the case where $k = 1$, the standard solution using binary searches has $P(n) = 0(n\log n)$, $S(n) = 0(n)$ and $Q(n) = 0(\log n)$.

The problem becomes much more complicated when k is greater than 1. In the case $k = 2$, Shamos [9], [10] used Voronoi polygons (optimally constructed in $0(n\log n)$) to obtain an algorithm with $P(n) = 0(n^2)$, $S(n) = 0(n^2)$, and $Q(n) = 0(\log n)$. He found [9] another algorithm which decreased P and S ($P(n) = 0(n\log n)$ and $S(n) = 0(n)$) while increasing Q(n) to $0(\log^2 n)$. Lipton and Tarjan [6] found a solution with $Q(n) = 0(\log n)$ in the worst case, $P(n) = 0(n\log n)$, and a very complicated storage. [Bentley *et al.* [3] have shown that for certain distributions of query points the algorithm has expected preprocessing and search times both of 0(n), and 0(1).] For other work in the area the reader should see Shamos and Hoey [10], Preparata and Hong [7], Lee and Preparata [5] and Bentley *et al.* [2]. For algorithms with $k > 2$, see Bentley [1], Yao [11], and Yuval [12].

In this paper we present a solution to the problem using a tree as the data structure (inspired by Wilf [13]), and an algorithm based on the triangle inequality. The points of S are placed in a tree using a binary decision process (see Section III); hence $P(n) = 0(n\log n)$. It will also be seen that $S(n) = 0(n)$. Although the algorithm itself is fairly simple (see Section II), an analysis of it seems to be very complicated. However, we have performed experiments (Section V) indicating, we believe, that $P(n) = 0(\log n)$ in the average.

We would like to point out two advantages of the algorithm. First, it works for any k. In fact, S need only be a finite subset of any normed space whose norm is effectively computable. Second, the set S need not be fixed for the algorithm to work, i.e., the tree can be continually updated, setting $S = \{x_1, \cdots, x_n, z\}$ before a new query point is read.

II. Problem, Lemma, and Algorithm

We are given a set S of points in the plane and a point z also in the plane, and wish to find the point q in S closest to z (or one of the closest points if q is not unique). The points of S will be placed on a binary tree T. Each node N of T will contain either one point p_L, or two points p_L and p_R. Let N_L (N_R) be the left (right) subtree below N. The tree will be constructed in such a way that if N contains only one point, N_L and N_R will be empty. We will search the tree, using the triangle inequality to avoid searching the entirety of certain branches. Let $DL = |z - p_L|$ ($|a - b|$ denotes the distance between a and b), $DR = |z - p_R|$, $LRADIUS = \max\{|w - p_L|: w \text{ is in } N_L\}$, and $RRADIUS = \max\{|w - p_R|: w \text{ is in } N_R\}$.

Suppose we have already searched through a subset S′ of S and that the point in S′ closest to z is q′ (the current nearest point). Denote $|z - q'|$ by NDIST.

Lemma: If DL-LRADIUS $\geqslant$ NDIST, then no point in N_L is closer to z than q′. (Similarly for R in place of L.)

Proof: Let w be a point in N_L. By the triangle inequality we have

$$|w - z| \geqslant |z - p_L| - |w - p_L|.$$

Since $|w - p_L| \leqslant$ LRADIUS, we conclude that

$$|w - z| \geqslant DL - LRADIUS \geqslant NDIST,$$

$$\text{i.e., } |w - z| \geqslant |q' - z|.$$

The lemma tells us that when searching T for q, if DL − LRADIUS $\geqslant$ NDIST (DR − RRADIUS $\geqslant$ NDIST), then there is no need to search N_L (N_R), since no point in the subtree will be closer to z than the current nearest point q′. We will say

Manuscript received January 18, 1982; revised May 24, 1982.

I. Kalantari is with the Department of Mathematics, Western Illinois University, Macomb, IL 61455.

G. McDonald was with the Department of Mathematics, Western Illinois University, Macomb, IL 61455. He is now with Loyola University, Chicago, IL 60606.

that *condition (L) holds* if DL – LRADIUS < NDIST, and *condition (R) holds* if DR – RRADIUS < NDIST. We now sketch an algorithm for finding q. Details will be given in the next two sections.

Algorithm: Begin the search for q at the base of the tree T, and move downward, updating q′ and NDIST when necessary. At each node go down to the left if condition (L) holds, and to the right if condion (R) holds. If both conditions hold, go in one direction and leave a flag indicating which direction it was. Continue moving downward until neither conditon holds. Then begin moving back up the tree along the same path, until a node with a flag is reached. Remove the flag and begin moving down the other subtree, following the above rules. The process will eventually end when you arrive back at the base and there is no flag there.

Let us consider an example. Suppose we travel down the tree to a node N, and both condition (L) and (R) hold. Thus both N_L and N_R may contain points closer to z than the current nearest point q′. Suppose we decide to search N_L first. We leave a flag in N indicating we have gone to the left. When the search N_L is complete we will have returned to N for a second time. Since there is a flag in the node we again check to see if condition (R) holds. (The condition may no longer hold since NDIST may have decreased while we were searching N_L.) If the condition no longer holds, remove the flag and continue up the tree. Otherwise remove the flag and begin the search of N_R. When this search is complete, we will arrive back at N for the third and last time. Since there is no longer any flag there, we move back up the tree.

Note that if at each node only one of the conditions held, we would have Q(n) = logn. With respect to logn time, we lose time when both conditions hold, and gain time when neither holds. We also note that whether or not (L) and (R) hold depends not only on the query point, but also on the points in the tree below the node. In particular, consider LRADIUS (N) and RRADIUS (N) for a node N, and suppose C_L is the left and C_R the right child of N. Then we need not have

$$\begin{aligned} &\mathrm{LRADIUS}(C_L) \leqslant \mathrm{LRADIUS}(N),\\ &\quad \mathrm{RRADIUS}(C_L) \leqslant \mathrm{LRADIUS}(N) \end{aligned} \tag{*}$$

and

$$\begin{aligned} &\mathrm{LRADIUS}(C_R) \leqslant \mathrm{RRADIUS}(N),\\ &\quad \mathrm{RRADIUS}(C_R) \leqslant \mathrm{RRADIUS}(N). \end{aligned} \tag{**}$$

Obviously, the more often we can replace "$\leqslant$" with "<", the better the search time should be. We will refer to a tree for which strict inequality holds in (*) and (**) for every node in the tree as a *monotone tree*. We will consider such trees in Section V.

III. The Tree

We put the set of points S into a threaded binary tree T. When full each node will contain two points of S, p_L, and p_R. There will be four pointers in each node: LCHILD and RCHILD pointing to the children of the node, ANCESTOR pointing up to the preceding node, and TEMPSTORAGE (to be described in the next section). The node will also contain LRADIUS, RRADIUS (as defined above), and the Boolean variable FULL, to indicate whether or not the node contains two points.

Suppose p = (X,Y) is a point of S and we wish to position p in the tree. Beginning at the base node, we compute the distance from p to p_L and p_R. If p is closer to p_L we continue down to LCHILD, otherwise to RCHILD. The recursion stops either when there are no nodes in the direction we wish to go (LCHILD = nil or RCHILD = nil), or we reach a node containing only one point (FULL = false). In the first case we create a new node and set p_L = p. In the second case we set p_R = p and FULL = true. We outline the above:

```
PROCEDURE PLACEPOINT (X,Y: REAL; VAR V,W:
                                    NODEPOINTER);
BEGIN
  If V = NIL THEN create new node
          {with $p_L$ := p, FULL := false, and
           ANCESTOR := W};
  If V↑.FULL = FALSE THEN fill node
                  {$p_R$ := p, FULL := true }
    ELSE
      BEGIN
        IF |p - $p_L$| < |p - $p_R$| THEN {go left}
          BEGIN
            update LRADIUS if necessary
            {if |p - $p_L$| > LRADIUS};
            PLACEPOINT (X,Y,V↑.LCHILD,V)
          END
        ELSE
          BEGIN
            update RRADIUS if necessary
            {if |p - $p_R$| > RRADIUS};
            PLACEPOINT (X,Y,V↑.RCHILD,V)
          END
      END
END.
```

(When a new node is created we initially have LRADIUS = 0, RRADIUS = 0. Both quantities are continuously updated as points pass through the node.)

IV. The Program

Our algorithm requires three procedures to search the tree for the point q closest to z. Recall that as we proceed down the tree, at each node n we must make a decision on which direction to move in next. We call procedure DECISION which assigns to DIRECTION the value U(up), L(left) or R(right) according to the following chart:

DL – LRADIUS < NDIST	Yes	Yes	No	No
DR – RRADIUS < NDIST	No	Yes	Yes	No
DIRECTION (if DR $\leqslant$ DL)	L	R*	R	U
DIRECTION (if DR > DL)	L	L*	R	U

An asterisk occurs when both condition (L) and (R) hold, so DECISION then creates a flag in the form of a pointer TEMPSTORAGE, which is placed in the node. TEMPSTORAGE points to a record which contains DL, DR, and the Boolean variable WENTRIGHT, which equals false if DIRECTION = L*, and true if DIRECTION = R*.

We move down the tree by calling procedure DOWNTOWN:

```
PROCEDURE DOWNTOWN (P: NODEPOINTER);
BEGIN
  IF P = BASE
    THEN
      BEGIN
        compute DL and DR {DL = |z - $p_L$|,
                           DR = |z - $p_R$|};
        NDIST: = min(DL,DR);
        initialize q' {the current nearest point}
      END;
  IF p↑.FULL = FALSE {the node contains only one
                      point $p_L$}
    THEN
      BEGIN
        compute DL;
        IF DL < NDIST THEN update NDIST and q';
        UPTOWN (P↑.ANCESTOR) {go back up tree}
      END
    ELSE
      BEGIN
        compute DL and DR;
        update NDIST and q' if necessary;
        DECISION (DL,DR,P);
        IF DIRECTION = U THEN UPTOWN
                       (P↑.ANCESTOR);
        IF DIRECTION = L THEN DOWNTOWN
                       (P↑.LCHILD);
        IF DIRECTION = R THEN DOWNTOWN
                       (P↑.RCHILD);
      END
END.
```

We move up the tree by calling procedure UPTOWN:

```
PROCEDURE UPTOWN (P: NODEPOINTER);
BEGIN
  IF P = NIL GOTO end of procedure {search ends,
     having reached BASE.ANCESTOR};
  IF P↑.TEMPSTORAGE = NIL {DIRECTION ≠ L* or R*}
     THEN UPTOWN (P↑.ANCESTOR)
     ELSE {DIRECTION = L* or R*}
       BEGIN
         retrieve DL, DR, WENTRIGHT from
           TEMPSTORAGE;
         DISPOSE (TEMPSTORAGE);
         IF WENTRIGHT AND
           (DL - LRADIUS ≥ NDIST) OR
         NOT WENTRIGHT AND
           (DR - RRADIUS ≥ NDIST)
         THEN {NDIST has gotten smaller since
              DOWNTOWN went through the node}
           UPTOWN (P↑.ANCESTOR)
         ELSE
           IF WENTRIGHT {$N_R$ has already been
                         searched}
             THEN DOWNTOWN (P↑.LCHILD)
             ELSE DOWNTOWN (P↑.RCHILD)
       END
END.
```

Finally the entire search algorithm can be written:

```
PROGRAM SEARCH;
form tree;
DOWNTOWN (BASE)
END.
```

V. The Experiments

We conducted three series of tests of our algorithm. The first test consisted of randomly generating a set S, of 512 points in [0,100] × [0,100], and randomly reading S into Program Search, forming a tree T. A set Z of query points was then read into the program, where Z = {(x,y): x = -50 + 4k, y = -50 + 4l} where k and l range over all integers between 0 and 50. Thus we were running 2601 query points z through the tree. For each z, the search time Q was computed as the number of nodes in the tree visited by z. The average search time $\overline{Q}$ was computed. We then generated nine more sets S and repeated the process. Finally we increased the number of points in S to 1024, 2048, and 4096, each time repeating the experiment. Our results are summarized in the following chart, where n is the number of points in the tree.

n	logn	Ave. $\overline{Q}$	Best $\overline{Q}$	Worst $\overline{Q}$	Δ(Ave $\overline{Q}$)
256	8	15.3	12.9	17.8	–
512	9	18.0	16.9	20.7	2.7
1024	10	22.2	20.1	24.5	4.2
2048	11	26.4	25.1	27.9	4.2
4096	12	30.5	28.2	32.3	4.1

Here Δ(Ave $\overline{Q}$) is the change in the average $\overline{Q}$. We note the small number of nodes that the algorithm needs to search. For example, when there are 4096 points in the tree, there will be at least 2048 nodes in the tree, and on the average we will need to visit less than 2 percent of them. We also note that for $k > 8$, Δ is nearly constant, suggesting that Q(n) is behaving like log n.

The second series of tests was the same as the first except that Z now consisted of the smaller set {(x,y): x = -50 + 4k, y = -50 + 4l}, where k = 0 and 1, and l ranges over all integers between 0 and 50. We obtained the following results:

n	logn	Ave. $\overline{Q}$	Best $\overline{Q}$	Worst $\overline{Q}$	Δ(Ave $\overline{Q}$)
256	8	15.7	12.8	18.5	–
512	9	18.0	15.0	21.1	2.3
1024	10	22.5	19.0	27.3	3.5
2048	11	27.6	24.1	30.5	5.1
4096	12	30.8	26.6	32.6	2.8

The results are compatible with the first series of experiments, the wider range of $\overline{Q}$ being accounted for by the smaller number of points run through each tree.

Finally, to test our hypothesis that the algorithm would work best on monotone trees, we constructed three monotone trees in the following manner. The points (0,0) and (100,0) were put into the base. Then points (0, -25) and (0,25) were read into the left child of the base, and (100, -25) and (100,25) into the right child. We continued this process as follows. Suppose (a,b) and (c,d) are points in a node, set $r = (c - a)/4$ and $r' = (d - b)/4$. We then place

(a, b ± r) in the left node, (c, d ± r) in the right node (†)

(a ± r′, b) in the left node, (c ± r′, d) in the right node, (††)

alternating (†) and (††) from generation to generation (think of this as placing figure-eights inside figure-eights.) We then ran the set Z, used in the first series of tests, through the tree. We obtained the following:

n	≈logn	$\overline{Q}$	σ
126	7	5.19	0.85
254	8	6.35	1.12
510	9	7.55	1.52

Here n represents the number of points in the tree, $\overline{Q}$ the length of the average search, and σ the standard deviation.

VI. Remarks

Clearly, more studies are needed. For example, mathematical methods could be employed to analyze the algorithm both for monotone and random trees with various probabilistic distributions for the points. Another interesting question pertains to the dimension of the search space. In $\mathbb{R}$ a point is a "cut" in the space in the sense that it divides $\mathbb{R}$ into two disjoint half lines. Our algorithm for $\mathbb{R}^2$ requires to look at two points at every node of the tree. These two points help to divide the $\mathbb{R}^2$ into two disjoint half-planes.

In general, (for $\mathbb{R}^k$), even though we still need only two points, x_1 and x_2 to "divide" $\mathbb{R}^k$ into two "halves" (the half closer to x_1 than to x_2 and vice versa), we could decide to divide $\mathbb{R}^k$ into two "half-spaces." For that we need k (independent) points to form a hyperplane and divide $\mathbb{R}^k$ accordingly. Then k points may be put at each node of a tree which branches k times. We can then fill the tree with the same strategy described in this paper for $\mathbb{R}^2$, namely if x is a child x_i where $x_1, \cdots, x_k$ are all at one node, then x is closer to x_i than it is to x_j for $j \neq i$. The same lemma as in Section II holds here and the same search may be carried out. We conjecture that the expected search time will still be 0 (logn) but as the dimension grows higher the constant involved (which is a function of dimension) will grow exponentially.

Interestingly, even in $\mathbb{R}^2$ we could put more than two points, say l points, in every node of the tree and then branch it l times to l new nodes and still use the same algorithm presented in this note. An intriguing question is whether there is an optimal l for $\mathbb{R}^2$ for the monotone trees or for uniformly distributed (as well as others) points? For a fixed n, as l increases, the diameter of an element of a node decreases and one expects to be able to avoid searching less number of branches. Hence, there is clearly a tradeoff. Along these lines a simple question is which is better for $\mathbb{R}^2$: l = 2 or l = 3?

We intend to answer some of these questions in future work.

Acknowledgment

We wish to thank George Nagy for introducing us to this problem.

References

[1] J. L. Bentley, "Multidimensional divide-and-conquer," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 214-229, Apr. 1980.

[2] J. L. Bentley, D. F. Stanat, and E. H. Williams, Jr., "The complexity of finding fixed-radius near neighbors," *Inform. Processing Lett.*, vol. 6, pp. 209-212, Dec. 1977.

[3] J. O. Bentley, B. W. Weide, and A. C. Yuo, "Optimal expected time algorithms for closest point problems," *ACM Trans. Math. Software*, vol. 6, pp. 563-580, Dec. 1980.

[4] D. E. Knuth, *The Art of Computer Programming*, vol. 3. Reading, MA: Addison-Wesley, 1969.

[5] D. T. Lee and F. P. Preparata, "Location of a point on a planar subdivision and its applications," Coordinated Sci. Lab. Rep. R-699, Univ. Illinois, Urbana, Nov. 1975.

[6] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," in *Proc. 18th IEEE Symp. Foundations Comput. Sci.*, Oct. 1977, pp. 162-170.

[7] F. P. Preparata and S. J. Hong, "Convex hull of finite sets of points in two and three dimensions," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 87-93, Feb. 1977.

[8] M. I. Shamos, "Problems in computational geometry," Dep. Comput. Sci., Yale Univ., New Haven, CT, May 1975.

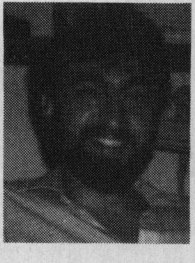
[9] M. I. Shamos, "Geometric complexity," in *Conf. Rec. 7th Annu. ACM Symp. Theory of Comput.*, Albuquerque, NM, May 1975, pp. 224-233.

[10] M. I. Shamos and D. Hoey, "Closest-point problems," in *Proc. 16th IEEE Symp. Foundations Comput. Sci.*, Oct. 1975, pp. 151-162.

[11] A. C. Yao, "On constructing minimum spanning trees in k-dimensional space and related problems," Dep. Comput. Sci., Stanford Univ., Res. Rep. STAN-CS-77-642, 1977.

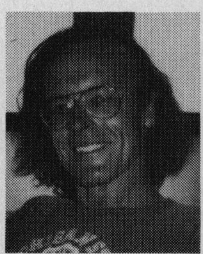
[12] G. Yuval, "Finding nearest neighbors," *Inform. Processing Lett.*, vol. 5, pp. 63-65, Aug. 1976.

[13] H. S. Wilf, "The 'Why-don't-you-just . . . ?' barrier in discrete algorithms," *Amer. Math. Month.*, vol. 86, no. 1, pp. 30-36, 1979.



Iraj Kalantari received the M.S. degree in applied mathematics and the Ph.D. degree in mathematics from Cornell University, Ithaca, NY.

He held a lectureship at the University of California at Santa Barbara and a visiting position at the Department of Computer Science, University of Nebraska, Lincoln. Since 1978, he has been at Western Illinois University, Macomb.



Gerard McDonald received the Ph.D. degree in mathematics from the State University of New York at Stony Brook in 1975.

He was a Research Associate and Instructor in the Department of Mathematics, University of Wisconsin, Madison, from 1975 to 1977. From 1977 to 1979 he was an instructor at Michigan State University. He was with the Department of Mathematics, Western Illinois University, Macomb. He is currently with Loyola University, Chicago, IL.