Optimal Expected-Time Algorithms for Closest Point Problems

JON LOUIS BENTLEY
Carnegie-Mellon University
BRUCE W. WEIDE
The Ohio State University
and
ANDREW C. YAO
Stanford University

Geometric *closest point problems* deal with the proximity relationships in k-dimensional point sets. Examples of closest point problems include building minimum spanning trees, nearest neighbor searching, and triangulation construction Shamos and Hoey [17] have shown how the *Voronoi diagram* can be used to solve a number of planar closest point problems in optimal worst case time. In this paper we extend their work by giving optimal *expected-time* algorithms for solving a number of closest point problems in k-space, including nearest neighbor searching, finding all nearest neighbors, and computing planar minimum spanning trees. In addition to establishing theoretical bounds, the algorithms in this paper can be implemented to solve practical problems very efficiently.

Key Words and Phrases: computational geometry, closest point problems, minimum spanning trees, nearest neighbor searching, optimal algorithms, probabilistic analysis of algorithms, Voronoi diagrams
CR Categories: 3.74, 5 25, 5.31, 5.32

1. INTRODUCTION

Shamos and Hoey [17] have collected and studied a group of problems in computational geometry that they refer to as *closest point* problems. Problems in this set are usually defined on point sets in Euclidean space and include such computational tasks as nearest neighbor searching, finding all nearest neighbor pairs, and constructing Voronoi diagrams. The merits of studying these problems *as a set* have been proved repeatedly since the class was first defined. Not only do the various problems often arise in the same application areas, but time and again we have seen that advances made in the computational efficiency of an algorithm for one of the problems can be applied to increase the computational

This work was supported in part by the Office of Naval Research under Contract N00014-76-C-0370 and in part by the National Science Foundation under Grants MCS-77-05313 and MCS-79-12688.

Author's addresses: J. L. Bentley, Departments of Computer Science and Mathematics, Carnegie-Mellon University, Pittsburgh, PA 15213, B. W. Weide, Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210; A. C. Yao, Computer Science Department, Stanford University, Stanford, CA 94305.

efficiency of others. In this paper we continue in this spirit by showing how the technique of "cells" can be used to produce optimal expected-time algorithms for many closest point problems.

All of the closest point problems that we will study in this paper have as input a set S of n points in Euclidean k-space. The *nearest neighbor searching* problem calls for organizing the set S into a data structure such that subsequent queries asking for the nearest point in S to a new point can be answered quickly. The *all nearest neighbors* problem is similar: it calls for finding for each point in S its nearest neighbor among the other points of S. Both of these problems arise in statistics, data analysis, and information retrieval. A problem similar to the nearest neighbor problems is that of finding the *closest pair* in a point set: that pair of points realizing the minimum interpoint distance in the set. The *minimum spanning tree* (or *MST*) problem calls for finding a tree connecting the points of the set with minimum total edge length. This problem arises in statistics, image processing, and communication and transport networks. The most complicated closest point problem that we will investigate in this paper is the problem of constructing the *Voronoi diagram* of a point set. This problem, along with its applications, is described in Section 3. All of the problems that we have just mentioned are described in great detail by Shamos [16]; he also discusses many of their applications.

Much previous work has been done on closest point problems. The seminal paper in this field is the classic work of Shamos and Hoey [17] in which the problems are defined and a number of optimal worst case algorithms for *planar* point sets are given. Algorithms for closest point problems in higher dimensional spaces have been given by Bentley [2], A. Yao [20], and Yuval [21]. Randomized algorithms for the closest pair problem have been given by Rabin [14] and Weide [18]; Fortune and Hopcroft [7] have recently shown that the speedup of the fast closest pair algorithms was not due to their randomized nature alone, but also to the model of computation employed (which allowed floor functions). Additional results on closest point problems are given by Preparata and Hong [13] and Lipton and Tarjan [11].

In this paper we study closest point problems from a viewpoint not taken in any of the above papers. We assume that the point sets are randomly drawn from some "bounded" underlying distribution, and then use the cell technique[1] to give fast expected-time algorithms for many closest point problems. In Sections 2 and 3 we illustrate this idea by applying it to two fundamental closest point problems (nearest neighbor searching and Voronoi diagram construction) under the very restrictive assumption that the points are drawn uniformly from the unit square. In Section 4 we extend these results to other planar closest point problems and to point sets drawn from a wide class of distributions. In Section 5 we extend our algorithms to problems in Euclidean k-space. Most of the algorithms we present solve a problem on n inputs in expected time proportional to n, and all searching structures we give have constant expected retrieval time; these results are therefore optimal by the trivial lower bounds. Having thus resolved the primary

[1] A discussion of the application of the cell technique to other geometric problems can be found in Bentley and Friedman [4, Section 1.3].

theoretical issues, we turn to implementation considerations in Section 6. Conclusions and directions for further research are then offered in Section 7.

Before proceeding to the body of the paper, it is important to state its goals clearly. The primary aim of the algorithms we will see is to establish theoretical results, displaying a number of best known (and many optimal) algorithms. The secondary aim, however, is to exhibit a general method for solving closest point problems that is quite practical. We therefore discuss the algorithms at a theoretical level in Sections 2 through 5, and then turn to implementation issues in Section 6.

2. NEAREST NEIGHBOR SEARCHING

The problem that is easiest to state and most clearly illustrates the cell method is *nearest neighbor searching,* sometimes called the *post office problem.* Given n points in Euclidean space, we are asked to preprocess the points in time $P(n)$ in such a way that for a new *query point* we can determine in time $Q(n)$ which point of the original set is closest to it. Lipton and Tarjan [11] describe a (complicated) structure for solving the planar problem with $P(n) = O(n \log n)$ and $Q(n) = O(\log n)$ in the worst case, but one might expect that on the average we can do better. In fact, we will see that for a large class of distributions of points, the expected values of $P(n)$ and $Q(n)$ can be made to be $O(n)$ and $O(1)$, respectively. Although we initially restrict our attention to this apparently simple problem (and the planar case at that), the techniques used also apply to other closest point problems, which we investigate in later sections.

We first consider the problem of nearest neighbor searching in the plane, where the points (both the original n points and the query point) are chosen independently from a uniform distribution over the unit square. The idea of the preprocessing step is to assign each point to a small square (*bin* or *cell*) of area C/n, so that the expected number of points in each bin is C; we refer to C as the *cell density.* This step is easily done by creating an array of size $(n/C)^{1/2}$ by $(n/C)^{1/2}$ that holds pointers to the lists of points in each bin.

To process a query point, we search the cell to which it would be assigned. If that cell is empty, then we start searching the cells surrounding it in an expanding pattern until a point is found. Once we have one point, we are guaranteed that there is no need to search any bin that does not intersect the circle of radius equal to the distance to the first point found and centered at the query point. Figure 1 shows how one spirallike search might proceed for the query point in the middle of the circle. Once the point in the northeast neighbor is found, only bins intersecting the circle must be searched. Each of these is marked in Figure 1 with an integer denoting the order in which it was visited. In order to make this test easy, we suppose that all bins that lie within that distance of the query point in either coordinate are examined, making the number of cell accesses slightly larger than necessary but simplifying the specification of how the appropriate bins are to be found.

It is clear that preprocessing, which consists of assigning each point to the appropriate bin, can be accomplished in linear time if computation of the floor function in constant time is allowed. *This assumption is necessary to solve most of the closest point problems in $o(n \log n)$ time because lower bounds propor-*

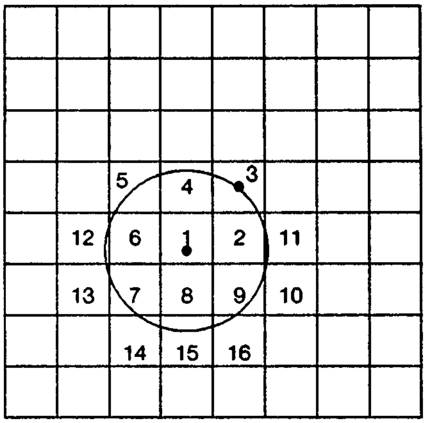

Fig. 1 Spiral nearest neighbor search using cells.

tional to n log n are known for many of them in the "decision tree with linear functions" model of computation. The following theorem shows that spiral search is indeed a constant-time solution to the nearest neighbor searching problem.

THEOREM 1. *If n points are chosen independently from a uniform distribution on the unit square, then spiral search finds the nearest neighbor of a query point in constant expected time.*

PROOF. Certain notation is required in this proof. We first define the concept of *layers* of cells surrounding the query point. We say that the cell containing the query point is in layer 1, the 8 cells surrounding that are in layer 2, the 16 cells surrounding those are in layer 3, and so on. In general, for any $k \geq 1$, the kth layer contains exactly $8(k - 1)$ cells and there are $(2k - 1)^2 \leq 4k^2$ cells on or within layer k. We will also use in our proof the constant

$$q = C/n.$$

Since the number of cells in the structure is n/C, q is the probability of a point being placed in a certain fixed cell. The probability that any particular point does not go into a certain fixed cell is $(1 - q)$, and the probability that a particular cell is empty is

$$(1 - q)^n = (1 - C/n)^n < e^{-C}$$

because it is empty if and only if all n points go elsewhere. (We use here the inequality $(1 + x/n)^n < e^x$.) By similar arguments, the probability that k given cells are all empty is bounded above by e^{-Ck}. We are now equipped to proceed to the probabilistic arguments required to prove Theorem 1.

Let $P(i)$ denote the expected number of points examined by the spiral search when the first $i - 1$ cells examined are empty and a point is found in the ith cell. Because the ith cell examined is in at most the $k = (i^{1/2} + 1)$st layer, one need search a total of at most $4k^2 \leq 12i + 4$ cells, or $11i + 4$ beyond the original i visited. We will now decompose $P(i)$ into $P(i) = P_F(i) + P_R(i)$, where $P_F(i)$ is the expected number of points in the first nonempty cell (the ith searched), and $P_R(i)$ is the expected number of points in the remaining (at most) $11i + 4$ cells;

recall that these definitions assume that cell i is the first nonempty cell. Letting j denote the number of points found in cell i, we have

$$P_R(i) \le \max \frac{(11i+4)(n-j)}{n/c - i} \bigg| \, 1 \le j \le n$$
$$\le \frac{(11i+4)(n-1)}{n/c - i},$$

because the probability of any one of the $n - j$ remaining points being placed into any particular one of the $11i + 4$ remaining cells is independently $1/(n/c - i)$. To evaluate $P_F(i)$, we let r_j denote the probability of observing j points in a particular cell if n points are placed randomly in $n/c - (i - 1)$ cells; the expected number of points in the ith cell given that it is nonempty is

$$P_F(i) = \frac{\sum_{1 \le j \le n} jr_j}{\sum_{1 \le j \le n} r_j}$$
$$= \frac{\sum_{0 \le j \le n} jr_j}{1 - r_0}$$
$$= \frac{n/[n/C - (i-1)]}{1 - r_0}$$
$$= \frac{n/[n/C - (i-1)]}{1 - [(n/C - i)/(n/C - i + 1)]^n}.$$

For $i \le (n/2C) - 2$ the denominator is bounded below by $\frac{1}{3}$, so we have

$$P(i) = P_F(i) + P_R(i)$$
$$\le \frac{3n}{n/C - i} + P_R(i)$$
$$\le \frac{(11i + 7 \cdot n)}{(n/C - i)}$$
$$\le 2C(11i + 7).$$

We can therefore bound $P(i)$ by the expression

$$P(i) \le \begin{cases} (11i+7)\cdot 2C, & \text{for } i \le (n/2C) - 2 \\ n, & \text{for } i > (n/2C) - 2. \end{cases}$$

We can now use the above probabilities to describe the expected cost of the spiral search. We count first the cost contributed to the search if the first point is found in cell i. Notice that the probability of this occurring is at most $e^{-C(i-1)}$, because the first $i - 1$ cells must be empty. If the first point is in the ith cell, then at most $12i + 4$ cells must be searched altogether; the expected number of points in those cells was defined above as $P(i)$. The expected cost contributed by the ith cell is the probability of its being the first in which a point is found times the sum of the number of cells searched and the expected number of points in those cells, which is bounded above by

$$e^{-C(i-1)} \cdot [12i + 4 + P(i)].$$

The total expected cost of the nearest neighbor search, denoted by $T(n)$, is just this product summed over all values of i from 1 to n/C (the number of cells). Using the previous upper bound on P (by cases), we have

$$T(n) \le \sum_{1\le i\le (n/2C)-2} e^{-C(i-1)}\cdot[12i+4+(11i+7)\cdot 2C] + \sum_{(n/2C)-2<i\le n/C} e^{-C(i-1)}\cdot n.$$

Simple techniques can now be applied to achieve an upper bound on $T(n)$. We first use "big ohs" (regarding C as a fixed constant) and conservatively rewrite the indices of summation to yield

$$T(n) \le O(1) \sum_{i\ge 1} (e^C)^{-i}\cdot O(i) + O(1) \sum_{(n/2C)-2\le i\le n/C} e^{-Ci}\cdot n.$$

The first term is bounded above by a constant (which can be proved by considering the convergence of the integral of the function xa^{-x}), so we have

$$T(n) \le O(1) + O(1) \sum_{(n/2C)-2\le i\le n/C} e^{-Ci}\cdot n.$$

Because i is always at least $(n/2C)-2$ in the above sum, we have

$$T(n) \le O(1) + O(1) \sum_{(n/2C)-2\le i\le n/C} e^{-C[(n/2C)-2]}\cdot n,$$

which is bounded above by

$$T(n) \le O(1) + O(1) \sum_{1\le i\le n/C} e^{-n/2}\cdot n.$$

This sum reduces to

$$T(n) \le O(1) + O(n^2\cdot e^{-n/2}) = O(1)$$

and establishes the theorem. □

Note that this proof is valid for any given query point in the unit square. The programming of the spiral search, however, must behave properly when cells on the boundary of the unit square are examined.

Although the proof of Theorem 1 is rather tedious, the theorem itself can be easily understood on an intuitive level. Phrased very briefly, nearest neighbors are a local phenomenon, and so are cells. The following is a lengthier but more graphic illustration. Suppose you were standing in the middle of a large hall that is covered with tiles that are 1 foot by 1 foot square; suppose furthermore that the hall has been sprinkled uniformly with pennies, so that there are a dozen pennies per tile, on the average. How many feet out will you have to look before you find the penny nearest you? Your answer will be independent of the size of the hall, because the *density* of pennies is the critical issue, and not their absolute number; whether the hall is 100 feet square or 1 mile square is immaterial. This is exactly the phenomenon we exploit in nearest neighbor searching by ensuring that there are a constant number of points per cell on the average.

We will now apply the cell method to a number of other closest point problems. Although the formal proofs of the algorithms will all have the rather complicated structure of the proof of Theorem 1, the reasons why the algorithms perform efficiently all come back to the same principle: closest point problems investigate local phenomena, and cells capture locality.

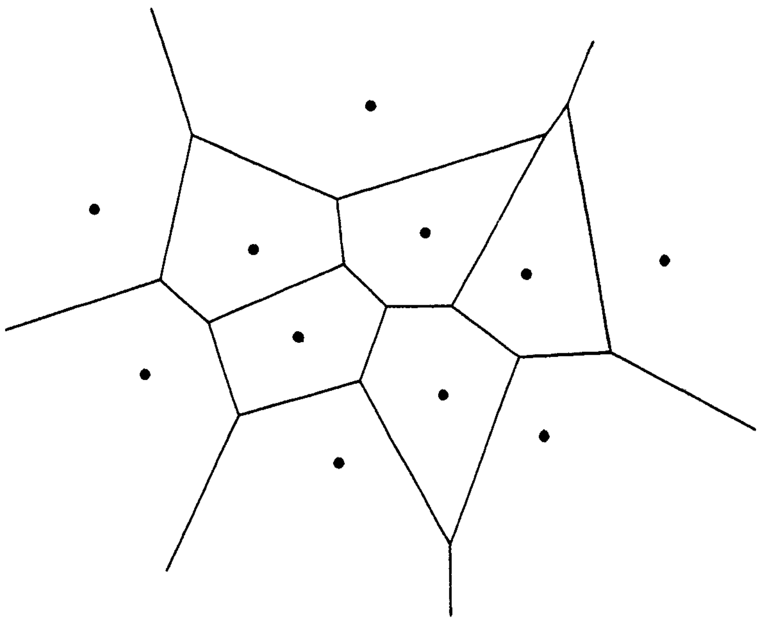

Fig. 2 A point set and its Voronoi diagram

3. THE VORONOI DIAGRAM

The *Voronoi diagram* of a point set is a device that captures many of the closeness properties necessary for solving closest point problems. For any point P in a set S the *Voronoi polygon* of P is defined to be the locus of all points that are nearer P than any other point in S. Notice that the Voronoi polygon of point P is a convex polygon with the property that any point lying in that polygon has P as its nearest neighbor. The union of the edges of all the Voronoi polygons in a set forms the Voronoi diagram of the set. A planar point set and its Voronoi diagram are illustrated in Figure 2. The Voronoi diagram has many fascinating properties that are quite useful computationally. We already mentioned the fact that the nearest neighbor to a new point is that point whose Voronoi polygon contains the new point. This fact can be used to give a fast worst case algorithm for nearest neighbor searching. Another interesting property of the Voronoi diagram is the fact that the *dual* of the diagram (that is, the graph obtained by connecting all pairs of points that share an edge in their respective Voronoi polygons) is a supergraph of the minimum spanning tree of the set and, furthermore, the dual contains at most $3n - 6$ edges. These and many other properties of the Voronoi diagram are discussed by Shamos [16] and Kirkpatrick [10]; these authors also describe computational representations of and manipulations on Voronoi diagrams. For instance, the Voronoi diagram of a set of n points can be found in worst case $O(n \log n)$ time, and the Voronoi diagrams of m-point and n-point sets can be merged to form the Voronoi diagram of the entire $(m + n)$-point set in worst case time of $O(m + n)$. Horspool [9] describes a computer program that implements the $O(n \log n)$ Voronoi diagram algorithm.

In this section we will see how the Voronoi diagram of n points distributed uniformly on the unit square can be constructed in linear expected time. We now

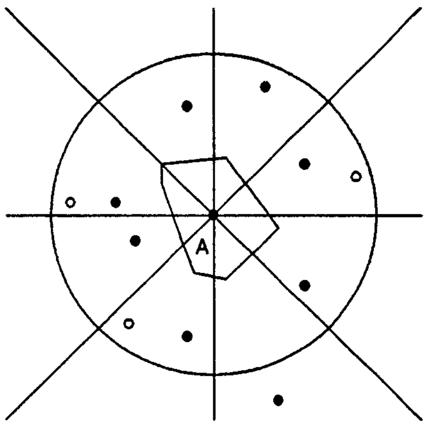

Fig 3 Construction of Voronoi polygon.

briefly sketch the algorithm before examining it in detail. Its first step is to place the points into the cells, which we divide into *inner* and *outer* sets, the outer containing all cells "near" the periphery of the square. The next two steps of the algorithm build separately the Voronoi diagrams of the points in the inner and outer cells. A cell-based method using constant expected time per point is used for the inner cells, and an $O(n \log n)$ worst case algorithm is applied to the (small) set of points in the outer cells. (This distinction is necessary because the periphery of a Voronoi diagram does not exhibit locality.) The expected running time of both of these steps is at most linear. The total expected running time of this algorithm is therefore linear in the number of input points.

Before studying the algorithm, we must specify precisely our representation of Voronoi diagrams. For each point P in the set we will give a linked list L_p that contains all of P's Voronoi neighbors (in counterclockwise order), along with a representation of the line segment that is the Voronoi edge separating the two points. This representation is fairly standard; furthermore, one can convert from this representation of a Voronoi diagram to any other proposed representation in linear worst case time. We are now equipped to examine the details of the algorithm.

Step 1. Divide the unit square into n cells, each with edge length $n^{-1/2}$. (Notice that we are just setting the cell density C of Section 2 to 1; the expected number of points per cell is therefore unity.) We then define all cells no further than C_1 $\log n$ cells from the boundary of the square to be *outer* cells; all the rest are *inner* cells. Note that the number of outer cells is less than $4C_1n^{1/2} \log n$, which is also an upper bound on the expected number of points in those cells.

Step 2. The goal of this step is to give the true Voronoi polygon of every inner point in linear expected time; we accomplish this by constructing the Voronoi polygon (in the representation described above) of each inner point in constant expected time. To find the Voronoi polygon for inner point P, we search all cells in a relatively small neighborhood of P in a spirallike fashion until at least one point is found in each of the eight[2] octants shown in Figure 3, or we give up

[2] Values less than 8 will also work, the choice of 8 is for didactic purposes only.

having searched out $C_1 \log n$ layers (approximately $(C_1 \log n)^2/2$ cells) in each octant. The *tentative* Voronoi polygon of the point P is that determined by considering just those eight points. (In Figure 3, the nearest points are dots and other points are circles.) Let d be the distance from point A to the farthest point of its tentative Voronoi polygon. Then no point farther than $2d$ from point A can have any effect on the actual Voronoi polygon of that point, which means that the Voronoi polygon of A can be constructed by considering only the small (expected constant) number of points that are in the circle of Figure 3.

In the case where there is at least one point found in each octant before the log n layers are searched, the point is called a *closed* point. A spiral search can be used to determine whether or not a given inner point is closed in constant expected time, and for a closed point its Voronoi polygon can then be computed in constant expected time. This can be proved by slightly modifying the proof of Theorem 1.[3] Performing the above operations on all inner points allows us to identify all closed points and compute their Voronoi polygons in linear expected time.

If all inner points were indeed closed, we would have the true Voronoi diagram of every point in the inner set. We now determine the probability that this is in fact the case. Note that any given octant of a point remains open after the spiral search iff $(C_1 \log n)^2/2$ cells were empty (for ease of writing, let $C_1^2/2$ be C_2). By the analysis of Section 2, the probability of this is bounded above by

$$e^{-C_2(\log n)^2} = n^{-C_3 \log n}$$

where $C_3 = C_2/\ln 2$. The probability of any given point being open is therefore bounded above by

$$8n^{-C_3 \log n}$$

because a point is open iff any of its eight octants are open. Since there are fewer than n inner points, the probability that one or more of them is open is at most

$$8n \cdot n^{-C_3 \log n} = 8n^{1-C_3 \log n},$$

which goes to zero very quickly as n increases.

The goal of step 2 is to construct in linear expected time the Voronoi polygon of every inner point. By the above analysis, the cell approach accomplishes this with high probability. If it does not (that is, if one of the inner points remains open), then we construct the Voronoi diagram of the entire point set in worst case $O(n \log n)$ time by one of the algorithms mentioned previously. The expected cost contributed by this is the product of the probability of its occurring times the cost incurred if it does, which is bounded above by

$$8n^{1-C_3 \log n} \cdot O(n \log n) = O(1) = O(n).$$

[3] The modifications to the proof of Theorem 1 can be sketched as follows. We let m be the maximum of the number of cells searched in any of the eight octants before a point was found in them. The probability that $m = i$ is an exponentially decreasing function of i, and the expected work required if $m = i$ is proportional to i. The details of this method are then quite similar to the proof of Theorem 1.

Thus we see that step 2 correctly computes the Voronoi polygons of the inner points in linear expected time.[4]

Step 3. The goal of this step is to compute the Voronoi polygon of every outer point. To accomplish this, we further divide the inner cells into two types: the *middle* cells are those cells within $2C_1 \log n$ layers of the outer cells, and the *innermost* cells are those inner cells that are not middle cells. The middle points are those points in middle cells, and likewise for the innermost points. The middle and innermost points thus form a partition of the inner points.

Let A be the union of the outer points and the middle points. Since A is contained in $3C_1 \log n$ layers of cells, the expected number of points in A is bounded above by $12C_1n^{1/2} \log n$. The algorithm now uses an $O(m \log m)$ worst case algorithm to compute the Voronoi diagram of all points in A, which requires $O(n^{1/2} (\log n)^2)$ expected time. We now show that with very high probability we have correctly computed the Voronoi polygon of every outer point, and that if we have not, then that situation can be quickly detected and remedied.

Our first task is to show that with high probability we have correctly computed the Voronoi polygon of every outer point. Note that if we do not yet have the true polygons, this must be because some outer point has an innermost point as a Voronoi neighbor. But this is true only if an octant of the given point is open (with respect to the middle points), which the analysis of step 2 shows occurs with exponentially decreasing probability. Because there are only a polynomial number of outer points, the probability of one or more of them having an innermost point as a Voronoi neighbor is exponentially decreasing.

We must, however, properly handle the rare event when an outer point has an innermost point as a neighbor. It is easy to check in linear time whether or not this is the case: we merely examine the set of all innermost polygons (which we correctly computed in step 2), and see if any have an outer point as a neighbor; this requires $O(n)$ time. If such a neighbor is found, then we merely compute the Voronoi diagram of the entire set in $O(n \log n)$ time. The expected contribution of this computation is the probability of its occurring times its cost, which is still exponentially decreasing. *End of Algorithm.*

Step 1 of the above algorithm has a linear worst case running time, while steps 2 and 3 have linear expected running time. It therefore follows that the Voronoi diagram of n points uniformly distributed on the unit square can be found in linear expected time.

4. EXTENSIONS OF PLANAR ALGORITHMS

The algorithms of Sections 2 and 3 can be used to solve a number of planar closest point problems. Given the fast nearest neighbor searching algorithm, we can easily solve the *all nearest neighbors* problem (which calls for finding the nearest neighbor of each point) in linear expected time, for point sets drawn from uniform distributions. This is accomplished by preprocessing the n points in linear time and then doing n searches, each of expected constant cost. Once we

[4] We note that we did not really require an $O(n \log n)$ time Voronoi diagram algorithm; any polynomial-time algorithm suffices to show that the expected contribution of the "fix-up" part of step 2 is exponentially decreasing.

have found all nearest neighbors, we can easily find the *closest pair* in the set by taking the minimum of the n distances. Shamos and Hoey [17] have shown that once we have constructed the Voronoi diagram of a point set, we can solve many other problems in linear worst case time. Together with the Voronoi diagram algorithm of Section 3, this allows us to solve both the minimum spanning tree and *Delaunay triangulation* problems in linear expected time. The details of these algorithms, together with some of the applications areas in which they arise, are discussed by Shamos [16].

All of the results that we have described so far are valid only for point sets drawn uniformly on the unit square; the algorithms can easily be adapted to work for many known distributions of points. The extension of these results to unknown distributions is a bit tricky. If we proceed for such a distribution as though it were uniform over some bounded region, a query can still be answered in constant expected time under certain conditions. The cells are chosen by first finding the minimum and maximum values in both x and y and then partitioning the rectangle defined by those four values into a number of squares proportional to n. The resulting cells can be represented by a two-dimensional array and our previous algorithms can operate as before. The only restriction on the underlying distribution required to achieve constant expected time is that it satisfy a condition similar to but more restrictive than a *Lipschitz condition* on its cumulative distribution function.

THEOREM 2. *Let n points be chosen independently from the distribution $F(x, y)$ over a bounded, convex open region in the plane, where F satisfies the condition that there exist constants $0 < C_1 \leq C_2$ such that for any region of area A, the probability assigned to the region by F lies between C_1A and C_2A. (Alternatively, F has a density with respect to Lebesgue measure that is bounded above and bounded below away from zero.) Then the same algorithm that was used for nearest neighbor searching in the uniform case answers a query in constant expected time.*

SKETCH OF PROOF. We first give the intuition behind the proof, and then sketch how the proof of Theorem 1 can be modified to prove this theorem. The requirement that the distribution be over some bounded convex region of the plane ensures that some constant proportion of the cells will be used to contain points of the distribution, and the expected number of points per cell will therefore be bounded above and below by constants. The lower bound on density, C_1, guarantees that the expected number of layers that need be examined before a point is found is small, and the upper bound C_2 guarantees that not many points will be in the neighboring cells when they are investigated.

We now formalize the above arguments. The critical observation is that there exist positive constants D_1 and D_2 such that the expected number of points in any of the cells is greater than D_1 and less than D_2.[5] The exact values of D_1 and D_2 depend on a number of parameters, including the size, shape, and orientation of

[5] Actually, there are some cells that intersect the region but are not contained entirely within it. For these cells, the upper bound D_2 remains in effect, but the lower bound D_1 might not hold. The fact that the region is open and convex is sufficient to show that there are at most $O(n^{1/2})$ such cells, and they are all on the boundary of the set. For this reason, their effect may be ignored in the rest of this analysis.

the convex region and the exact values of C_1 and C_2. The important point, however, is that such constants do exist.

Armed with these constants, it is easy to modify the proof of Theorem 1 to apply to the present theorem. We consider the case that the spiral search found its first nonempty cell in the ith cell visited. By the arguments in Theorem 1, the probability of this occurring is less than $e^{-D_1(i-1)}$. If this does occur, then the number of cells examined is at most $12i + 4$. Likewise the expected number of points in the cells is less than

$$\frac{(11i+4)\cdot(n-1)}{(n/D_2-i-1)}.$$

At this point, the rest of the argument in Theorem 1 follows immediately, because the "big oh" analysis performed there can be used here without change, assuming that D_1 and D_2 are constants. □

Similar arguments show how the above methods can be applied to give linear expected-time algorithms for all of the closest point problems mentioned above, when the point sets satisfy the "boundedness" conditions of Theorem 2.

5. EXTENSIONS TO HIGHER DIMENSIONS

In the previous section we showed how the algorithms of Sections 2 and 3 can be used to solve a number of problems with inputs drawn from a wide variety of planar distributions; in this section we see how the basic results can be extended to solve closest point problems in k-dimensional space. If the point sets are drawn independently and uniformly from the unit hypercube (that is, $[0, 1]^k$), then we can use the cell technique by dividing the hypercube into n/C cells, each of side $(C/n)^{1/k}$.

The first closest point problem in k-space that we examine is that of nearest neighbor searching. Dobkin and Lipton [6] showed that a nearest neighbor to a new point can be found in worst case time proportional to $k \log n$; their method requires preprocessing and storage prohibitive for any practical application, however. Friedman, Bentley, and Finkel [8] have described an algorithm that exhibits search time proportional to $\log n$ and has very modest preprocessing and storage costs. We will now examine a cell-based method for nearest neighbor searching that yields constant expected retrieval time. The preprocessing phase of the algorithm consists of placing the points into cells in k-space as described above. To perform a nearest neighbor search, we generalize the spiral search (searching out layer-by-layer) of Section 2, starting at the cell holding the query point and searching outward until we find a nonempty cell. At that point we must search all cells that intersect the ball centered at the query point with radius equal to the distance to the nearest neighbor found so far. The analysis of this algorithm is given in the following theorem.

THEOREM 3. *If n points are chosen independently from a uniform distribution on the k-dimensional unit hypercube, then spiral search finds the nearest neighbor of a query point in constant expected time.*

SKETCH OF PROOF. As before, the expected number of points in a given cell is exactly C. We again consider the case of finding the first point in the ith cell examined; the probability of this is at most $e^{-C(i-1)}$. If the first point is found in the ith cell, then the total number of cells examined is $O(i)$, and the expected number of points in those cells is $O(Ci)$. At this point, the analysis of Theorem 1 holds. □

Once we have the above result, we can solve both the all nearest neighbors and closest pair problems in linear expected time. The k-dimensional minimum spanning tree problem calls for finding a spanning tree of the point set of minimum total edge length. Straightforward algorithms for this task require $O(n^2)$ time. A. Yao [20] has shown that there is a subquadratic worst case algorithm for solving this problem, but his algorithm is probably slower than the straightforward method for most practical applications. Practical algorithms for this task have been proposed by Bentley and Friedman [3] and Rohlf [15], but the analysis of those algorithms remains primarily empirical. We now investigate the use of the cell technique to solve this problem in fast expected time. We use the method of Yao [20], which calls for finding the nearest neighbor of each of the n points in each of some critical number of generalized orthants. Yao has shown that the resulting graph is a supergraph of the minimum spanning tree of the point set. Since that graph contains a number of edges linear in n (for any fixed dimension k), the minimum spanning tree can be found in $O(n \log \log n)$ time (see Yao [19] or Cheriton and Tarjan [5]). Slight modification of the proof of Theorem 3 shows that the "nearest neighbor in orthant" searching can be accomplished in constant expected time for each point, and the total expected running time of this algorithm is therefore $O(n \log \log n)$.

Using the cell method to construct k-dimensional Voronoi diagrams in fast expected time appears to be a very difficult task. Generalizing the method of Section 3 allows us to find the Voronoi polytopes of all closed points in linear expected time, but at that point there still remain $O((n \log n)^{1-1/k})$ open points. Since no fast algorithms are known for constructing Voronoi diagrams in k-space, it is not clear how to find the true Voronoi diagram. Notice, however, that we have found the Voronoi polytopes of an increasing fraction of the points (that is, the ratio of open points to n approaches zero as n grows). This technique can be used for "Voronoi polytope searching," which asks for the actual Voronoi polytope containing the query point; our method will succeed in constant time with probability approaching 1.

The algorithms we have described above have all been for points drawn uniformly in $[0, 1]^k$. Methods analogous to those used in Section 4 can be used to show that the algorithms can be modified to work for point sets drawn from any distribution over some open bounded convex region with density bounded above and away from zero.

6. IMPLEMENTATION CONSIDERATIONS

In this section we discuss the implementation of the algorithms described in the preceding sections. It is important to mention one *caveat* that will be inherent in any application of the cell technique: the constant of linearity of most algorithms based on this method will increase *exponentially* with the dimension of the space k. This is true simply because a cell in k-space has $3^k - 1$ neighbor cells. It seems, though, that this complexity might be inherent in any algorithm for solving

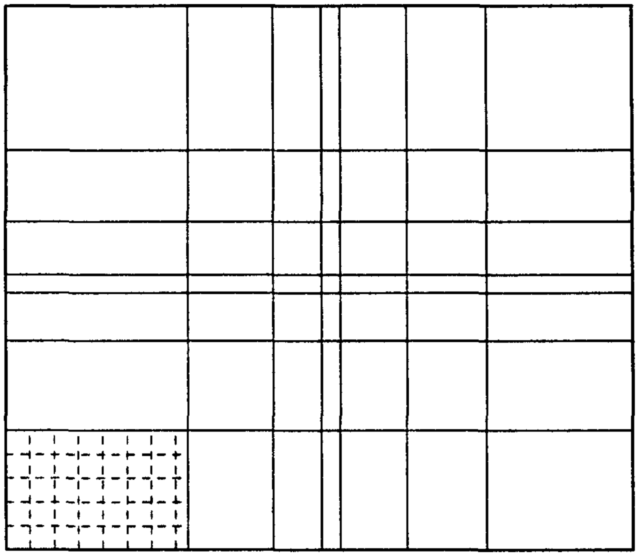

Fig 4. Adaptmg cell sizes by samplmg margmals.

closest point problems because a point in a high-dimensionl space can have many "close" neighbors. (More precisely, the maximum "adjacency" of point sets in k-space can be equated with the number of sphere touchings, which grows rapidly with k.) The practical outgrowth of this observation is that the methods we have described will prove impractical for large k; we estimate this will happen somewhere for k between 4 and 6 for data sets of less than 10,000 points.

Weide [18] has described how the empirical cumulative distribution function can be used to decrease the constants of the running times of programs based on cell techniques. We now briefly discuss the application of his techniques to the case of planar nearest neighbor searching. If the points to be stored for nearest neighbor searching indeed come from a uniform distribution on $[0, 1]^2$, then the cell technique performs very well. If the points come from a distribution that is not uniform (but still "smooth" enough to satisfy the requirements of Theorem 2), then the cells might perform poorly because they are at once both too large (in dense regions of the plane) and too small (in sparse regions). We would therefore like the cells to adapt their size in different regions of the space. One approximation to this "adaptive" behavior can be achieved with the cell method by incorporating a "conditioning pass" that examines the distribution before the points are placed in cells. This pass might work by finding the 10th, 20th, ..., 90th percentile points in both the x and y marginal distributions. Each set of 9 points partitions its dimension into 10 "slabs," and the cross product partitions space into 100 rectangles. Figure 4 illustrates such a partition of a heavily central distribution, such as a bivariate normal truncated at three standard deviations (where 6 points are sampled in each marginal, creating 49 rectangles). For many distributions satisfying the conditions of Theorem 2, the distribution of points within each rectangle will be much smoother than the distribution over the entire space. Because we sampled only a constant number of points in each dimension, we can locate which rectangle to examine in a nearest neighbor search in constant time. The exact number of rectangles to be used depends critically on the "roughness" of the underlying distribution—the more bounded the distribution, the fewer sample points required. These and other adaptation techniques are discussed in detail by Weide [18].

Although the searching structures that we have described in this paper are inherently *static*, they can be modified to become *dynamic*. We first consider the case in which the nearest neighbor structure is initially empty and must support a series of Insert and Search operations. To convert our cell structure from static to dynamic, we use a method that is similar to a method described by Aho, Hopcroft, and Ullman [1, p. 113] for converting a static hash table into a dynamic one. The nearest neighbor structure is initially defined to have a maximum allowable size of (say) eight; we will call this size max. When a new point is inserted into the structure, it is merely appended to the list of points currently in its cell. Whenever an insertion causes the number of points currently in the structure to exceed max, we perform the following operations: max is set to twice its current value, a new structure of max/C is created, and the points currently stored in the structure are "reinserted" into the new structure. Note that for any distribution satisfying the conditions of Theorem 2, the expected number of points per cell is always bounded above and below by constants. Furthermore, analysis shows that the total amount of computation required to insert n elements into this structure is proportional to n. (Whenever a structure of size m is rebuilt, it is because $m/2$ points were inserted, so the "amortized" cost per point is constant; for a more formal analysis, see Aho, Hopcroft, and Ullman [1].) Monier [12] has described a related technique that allows a hash table to support both insertions and deletions intermixed with queries. We can use his idea to give dynamic structures for all of the searching problems discussed in this paper with the following properties: a sequence of n insertions and deletions can be performed in time proportional to n, at any point in this sequence it is possible to perform a search in constant expected time, and the storage used by the structure is always proportional to the number of elements currently stored.

In the above discussion we have described a number of rather exotic extensions to the basic structures of this paper. For many applications, however, the basic structure is all that is needed. We will therefore briefly describe our experience in implementing the nearest neighbor searching algorithm[6] for point sets drawn uniformly on the unit square. The PASCAL implementation required 11 lines of executable code to place the points in cells and 34 lines to perform the search. The optimal number of points per cell (that is, the expected density, which we called C in Section 2) was 3; densities ranging from 1 to 9, however, decreased the running time by only 10 percent. The average running time for nearest neighbor searching (including preprocessing) was the constant 2765 microseconds per search, on a PDP-KL10. (This compares with $52n$ microseconds required by the

[6] Since one of the main points of this paper is that the algorithms described are of practical interest (in addition to theoretical), the authors feel obligated to explain that they implemented only this simple case because the code required to implement the more complicated algorithms is complex. (For example, the Fortran code available to the authors for constructing planar Voronoi diagrams was 1000 lines long.) This will be less of an obstacle for implementors of real programs who are already using complex "underlying" routines.

straightforward linear search; the break-even point is at $n = 53$.) In using this code to find all nearest neighbors in a 1000-point planar set, the cell method required less than 2.8 seconds while the quadratic algorithm required 52 seconds.

To learn more about the performance of the algorithm in real applications, we ran the program on two sets of real geographic data. Both data sets consisted of points representing the population centroids of political areas; the original latitudes and longitudes in seconds were scaled to be in [0, 1]. The first data set represented the 318 census tracts in San Diego County, California, and the second data set represented the 1122 precincts in the state of New Mexico. The points in the first data set were fairly uniform over approximately half of the unit square, while the points in the second were very clustered (in the few large cities in New Mexico). We used the programs described above to find the nearest neighbors in each set, with the results shown in Table I.

Table I

	San Diego	New Mexico
Number of points	318	1122
Quadratic algorithm		
Predicted time	5.25	65.5
Observed time	5.35	66.5
Cell algorithm		
Predicted time	0.88	3.11
Observed time	1.40	7.56
Optimum cell density	1.7	3.0

All times are noted in seconds; the predicted run times are $2765n$ microseconds for the cell algorithm and $52n^2$ microseconds for the quadratic algorithm.

Table I merits interpretation. Note first that the predictions of run time for the quadratic algorithm were extremely accurate. The predictors for the cell method did not fare as well: the ratio of observed run time to predicted is approximately 1.6 for San Diego and 2.4 for New Mexico. This is due to the nonuniformity of the data; the greater clustering in New Mexico led to a greater degradation of performance. Even so, though, note that the ratio of observed quadratic run time to observed cell run time is 3.9 for San Diego and 8.8 for New Mexico. Thus we may conclude that although our algorithms will experience a slight degradation when used on "real" data, they can still lead to a substantial savings in run time.

In this section we have discussed the implementation of the algorithms described in the preceding sections. Those algorithms share a common two-phase structure: in the first phase the points are stored in cells and in the second phase additional processing is done on the points. The implementation of the first phase is trivial; points can be placed in cells by first finding the cell number (accomplished by a multiplication for scaling and a floor function to find the integer cell index) and then performing an array index. The difficulty of the second phase of processing will depend on the particular problem being solved. In the case of nearest neighbor searching, all that is required is a "spiral search" and some distance calculations; the above discussion shows that both of these are easy to implement very efficiently. For the Voronoi diagram, however, the second phase of processing is very complicated. One advantage of the locality inherent in closest point problems is that very slow algorithms may be used to perform the operations that take place in a local area; this will increase the constant of linearity, but will not slow the asymptotic running time of the algorithms. This implies that a practitioner *may* be able to improve the performance of existing programs by dividing the points into cells in a first phase.

7. CONCLUSIONS

In this paper we have seen a number of algorithms for solving multidimensional closest point problems. The algorithms were all based on the simple idea of cells, and were analyzed under the assumption that the points were drawn from some underlying "bounded" distribution. All of the searching methods we described have linear preprocessing costs and constant expected searching costs; all of the algorithms (with the exception of k-dimensional minimum spanning trees) have linear expected running time. It is clear that these algorithms achieve the trivial lower bounds and are therefore optimal. Although we have described the algorithms primarily as theoretical devices (sacrificing efficiency for ease of analysis), the discussion in Section 6 described how many can be efficiently implemented on a random-access computer.

Much further work remains to be done in developing fast expected-time algorithms for closest point problems. Can the expected complexity of computing minimum spanning trees in k-space be reduced from $O(n \log \log n)$ to $O(n)$? A particularly important problem is to extend our results from the bounded distributions of Theorem 2 to unbounded distributions (the multivariate normal, for example). It appears that new algorithms will have to be developed for this problem, taking special care of "outliers." Another very interesting open problem is to describe precisely how much of the efficiency of our algorithms is gained from probabilistic assumptions and how much is gained by use of the floor function. (The recent paper of Fortune and Hopcroft [7] shows that floor can be used to speed up the computation of closest pair without making the randomization assumptions of Rabin [14] and Weide [18].)

ACKNOWLEDGMENTS

The helpful comments of James B. Saxe, Professor Michael Ian Shamos, and the two anonymous referees are gratefully acknowledged.

REFERENCES

1. AHO, A V , HOPCROFT, J.E., AND ULLMAN, J.D *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass , 1974.
2 BENTLEY, J.L. Multidimensional divide-and-conquer. *Commun. ACM 23*, 4 (April 1980), 214-229
3 BENTLEY, J L., AND FRIEDMAN, J.H. Fast algorithms for constructing minimal spanning trees in coordinate spaces *IEEE Trans. Comput C-27*, 2 (Feb. 1978), 97-105.
4. BENTLEY, J.L , AND FRIEDMAN, J.H. Data structures for range searching. *Comput. Surv. 11*, 4 (Dec. 1979), 398-409.
5. CHERITON, D , AND TARJAN, R E. Finding minimum spanning trees *SIAM J. Comput. 5*, 4 (Dec 1976), 724-742
6. DOBKIN, D , AND LIPTON, R J Multidimensional searching problems. *SIAM J. Comput 5*, 2 (June 1976), 181-186.

7. FORTUNE, S., AND HOPCROFT, J E A note on Rabin's nearest-neighbor algorithm. *Inf. Process. Lett. 8*, 1 (Jan. 1979), 20-23.
8. FRIEDMAN, J.H, BENTLEY, J.L., AND FINKEL, R.A. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 209-226.
9. HORSPOOL, R N. Constructing the Voronoi diagram in the plane Tech Rep SOCS-79.12, Comput Sci School, McGill Univ, Montreal, Canada, July 1979.
10 KIRKPATRICK, D G Efficient computation of continuous skeletons. *Proc. 20th IEEE Symp Foundations of Computer Science*, Oct. 1979, pp 18-27.
11. LIPTON, R.J., AND TARJAN, R.E. Application of a planar separator theorem. *Proc 18th IEEE Symp. Foundations of Computer Science*, Oct. 1977, pp. 162-170.
12. MONIER, L. Personal communication of Louis Monier of the Université de Paris-Sud to J.L. Bentley, June 1978.
13. PREPARATA, F.P., AND HONG, S.J. Convex hull of finite sets of points in two and three dimensions. *Commun. ACM 20*, 2 (Feb. 1977), 87-93.
14. RABIN, M O. Probabilistic algorithms, in *Algorithms and Complexity: New Directions and Recent Results*, J.F. Traub (Ed.), Academic Press, New York, 1976, pp. 21-39.
15. ROHLF, F J A probabilistic minimum spanning tree algorithm. *Inf. Process. Lett. 7*, 1 (Jan 1978), 44-48.
16. SHAMOS, M.I Computational geometry Ph.D. Dissertation, Yale Univ., New Haven, Conn., May 1978.
17. SHAMOS, M.I, AND HOEY, D. Closest-point problems *Proc 16th IEEE Symp. Foundations of Computer Science*, Oct 1975, pp. 151-162.
18. WEIDE, B.W. Statistical methods in algorithm design and analysis. Ph.D. Dissertation, Carnegie-Mellon Univ, Pittsburgh, Pa, Aug 1978 (Appeared as CMU Comput Sci. Rep. CMU-CS-78-142)
19. YAO, A.C. An $O(|E|\log\log|V|)$ algorithm for finding minimum spanning trees. *Inf. Process. Lett. 4*, 1 (Sept 1975), 21-23
20 YAO, A C On constructing minimum spanning trees in k-dimensional space and related problems. Res. Rep STAN-CS-77-642, Dep. Comput. Sci., Stanford Univ., Calif., 1977.
21. YUVAL, G. Finding nearest neighbors. *Inf. Process. Lett 5*, 3 (Aug. 1976), 63-65

Received April 1979, revised March 1980; accepted July 1980