Fast Algorithms for the All Nearest Neighbors Problem

Kenneth L. Clarkson

Computer Science Department
Stanford University
Stanford, CA 94305

§1 Introduction

We present three new algorithms to solve the *all nearest neighbors* problem:

> Given a set A of n points in d-dimensional Euclidean space, find the nearest neighbors in set A of each point in A.

This problem in computational geometry [Sha] arises in the field of statistical data analysis and pattern recognition, and is closely related to other "proximity" problems such as finding minimum spanning trees in Euclidean spaces, or finding the farthest pair of a set of points [Yao].

The three algorithms are:

- Probabilistic algorithm ANN_p, with an expected running time of $O(n \log n)$, for any input set of points. The fastest known deterministic algorithm for this problem [Ben] has worst-case running time $O(n \log^{d-1} n)$.
- Algorithm ANN_e, with a linear expected running time for random input points independently identically distributed, under very general conditions on the probability density function. Previous work [BWY] has yielded similar results, but under much more restrictive conditions.
- Algorithm ANN_δ,with running time $O(n \log \delta)$, where δ is the ratio of the diameter of the point set to the distance between the closest pair of input points. No comparable result for this problem has appeared before.

The complexity bounds given for ANN_p and ANN_e are under the assumption that the logarithm, floor, and bitwise exclusive-or functions have unit cost.† The constant factors for the time bounds also depend on the dimension d.

The first two algorithms indicated involve the construction and use of a compressed form of the "hyperoctree" data structure [YS]. The following section describes algorithm ANN_δ, and the key idea for all of the algorithms. Sections 3 and 4 briefly describe algorithms ANN_p and ANN_e, respectively.

†The operation required, for values $0 \le \alpha, \beta \le 1$ with binary representations $.\alpha_1\alpha_2\ldots$, $.\beta_1\beta_2\ldots$, is: find k such that $\alpha_i = \beta_i$, for $1 \le i \le k$, but $\alpha_{k+1} \ne \beta_{k+1}$. This operation is constant time, for most applications, whether the point coordinate values are represented in fixed or floating point.

§2 Algorithm ANN_δ

The basic approach to algorithm ANN_δ, as well as to all of the algorithms presented here, is to use a cell technique ([BWY], [BFP]). The region containing the input points is partitioned into equal-sized cubic cells. The distance from a point to its nearest neighbors is bounded using the distance of the cell containing the point to the nearest cell also occupied by a point. This bound is improved by refining the cellular partition into smaller and smaller cubic cells. Computational effort at each step is saved by storing, for each cell occupied by at least one point, a set of nearest neighbor occupied cells: those cells that contain points that might be nearest neighbors to the points in the cell.

More precisely, assume without loss of generality that the input points are contained in the unit d-cube $U = [0,1)^d$. Let $\mathcal{B}$ denote the hierarchy of cells defined by the following: U is in the hierarchy, and if a cell is in the hierarchy, so are the 2^d equal-sized cubes obtained by cutting the cell in half in each coordinate. Let $d_{\min}(C,D)$ denote the minimum distance between two cells C and D, and $d_{\max}(C,D)$ denote the maximum. We adopt the convention that $d_{\max}(C,C) = \mathit{diameter}(C)$. Let $\mathit{Children}(C)$ denote the set of children of cell C in $\mathcal{B}$. Algorithm ANN_δ can be described as follows:

```
procedure ANN_δ(Points : Set_of_Points);
begin
  Contents(U) ← Points;
  NN_Set(U) ← {U};  S ← {U};
  while S ≠ {} do
    S' ← ⋃_{C∈ S} {C' | C' ∈ Children(C), C' occupied};
    for each C' ∈ S' do Update_NNs(C') od;
    for each C' ∈ S' do
      if Contents(C) is a single point, for all C ∈ NN_Set(C')
        then output NN_distance(C'), Contents(C');
             S' ← S' − {C'};
      fi; od;
    S ← S';
  od;
end;
```


0272-5428/83/0000/0226$01.00 © 1983 IEEE

Procedure *Update_NNs*(C' : *Cell*) is:

```
begin
  C  ←  the parent of C' in B;
  Contents(C')  ←  {p ∈ Contents(C), p in C' };
  if |Contents(C')|  >  1
    then NN_distance(C') ← diameter(C')
    else NN_distance(C') ← min{d_max(D', C') |  D' ∈ Children(D),
                                                D ∈ NN_Set(C)};
  fi;
  NN_Sel(C') ← {D'  |  D ∈  NN_Set(C), D' ∈  Children(D),
                          d_min(D', C') < NN_distance(C')};
end;
```

The following lemma, proven in Appendix 1, indicates why this is a viable procedure.

Lemma 1. At each refinement step, the total of the sizes of the NN sets for all the occupied cells is linear in the number of occupied cells at that step.

Indeed, consider the number of cells for which a given cell is a nearest neighbor. This number is bounded by a constant dependent on the dimenson. If a cell is a nearest neighbor of too many cells, those cells must be closer to each other than to the given cell. This observation simply extends to equal-sized cells the observation made by Bentley about points [Ben].

The above basically describes ANN_δ. The work at each refinement step is $O(n)$, and after $\lceil \log \delta \rceil$ steps, no cell contains more than one point. Note that if the points were spread evenly, each occupied cell would be refined into at least two occupied cells at the next refinement step, so there would be $O(\log n)$ refinement steps and $O(n \log n)$ work overall.

§3 Algorithm ANN_p

Unfortunately, a smooth spread of points giving ANN_δ a fast running time is not necessarily present, as a cell may have a concentration of a large number of points in a small volume, so that repeatedly splitting the cell will not yield more than one occupied cell for some arbitrarily large number of steps. The hierarchy of occupied cells obtained by ANN_δ may be envisioned as a tree, with each node corresponding to an occupied cell. Trees of this kind are a generalization of the *quadtree* data structure [Sam] to higher dimensions. Such trees may have arbitrarily long paths without any branching. To use the basic approach of ANN_δ, but avoid this problem, we will show how to build, in probabilistic $O(n \log n)$ time, a *cell-tree* data structure that is simply the tree described, but with the non-branching paths replaced by single edges. This procedure is described in **3.1** below. In **3.2** an algorithm is described for using the cell-tree to solve the all nearest neighbor problem in worst-case linear time. Note that cell-trees are related to quadtrees in basically the same way as "Patricia" search trees are to digital search trees [Knu].

3.1 Building a Cell-Tree

To build a cell-tree, we need a procedure *Smallest_Cell*, that answers the following query:

> Given a set of points, what is the smallest cube in $\mathcal{B}$ containing those points?

Procedure *Smallest_Cell* is the only part of ANN_p that requires the use of the floor, logarithm, and bitwise exclusive-or functions. It is described more fully in Appendix 2.

Given procedure *Smallest_Cell*, a cell-tree can be built in the following way. The process is recursive, and at each call, the input is a cell $T \in \mathcal{B}$ and a set of points and cell-trees within T. The input cell-trees are identified with the root cell containing them. T is the smallest cell in $\mathcal{B}$ containing those objects, and the output is a cell-tree with T as the root. If the number of objects (points and cells-trees) within T is no more than some small constant K, then the cell-tree with T as root is found deterministically, and the algorithm returns. Otherwise, the problem of building the cell-tree for T is split into two pieces of expected size between ϵk and $(1-\epsilon)k$, where k is the number of objects within T and $\epsilon = 1/(2^d + 1)$.

In pseudo-code:

```
function Build_Cell_Tree(T : cell, Objects : set_of_objects)
  returns cell_tree_root;
begin
  if |Objects|  ≤  K
    then return Build_Cell_Tree_Deterministially(T, Objects)
    else
      choose random S ⊂ Objects;
        { should have ε|S| > 1 to avoid degeneracy }
      find C ∈ B with proportion of objects in S contained
          in C between ε and 1 − ε;
      C' ← Smallest_Cell(Contents(C) ∩ S);
        {C' is the smallest cell in B containing objects in C}
      O'  ← {x |  x ∈  Objects, x inside C' };
      T' ← Build_Cell_Tree(C', O');
      return Build_Cell_Tree(T,  Objects ∪ T' − O');
  fi;
end
```

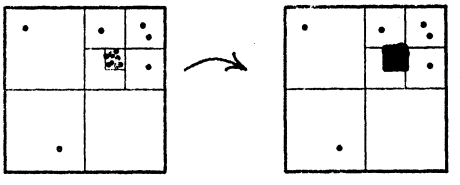
To find the desired cell C, we look at the quadtree children of T, looking for one with the desired number of objects. If there are none, there must be one with more than $1-\epsilon$ of the objects. That sub-cell is recursively examined to find the cell desired. Each time we begin to examine a sub-cell, we find the smallest cell in $\mathcal{B}$ that contains the sample objects in the sub-cell, using procedure *Smallest_Cell*.

Initially the algorithm is called with the n points and the smallest cube containing those points, scaled to be $[0,1)^d$. Since we expect to split the problem into two pieces with between ϵk and $(1-\epsilon)k$ objects, taking $O(k)$ time, the

process requires $O(n \log n)$ expected time overall. This time bound is proven as Lemma 2 in Appendix 1:

Lemma 2. Procedure *Build_Cell_Tree* requires $O(n \log n)$ expected time.

For example, in the square below, with a total of 15 points, a square is sought with more than 3 and less than 12 points. No child of the root square will do, so the heavy upper right square with 13 points is examined. Within this square, the lower left square will do, so we take as C' the smallest square in $\mathcal{B}$ containing all of the points within that lower left square. At the second recursive call, with 7 objects in the square (6 points and a square), again no child of the root will do, so the heavy upper right corner subsquare will be examined. This time, the upper right corner of the subsquare will do.



3.2 Using a Cell-Tree: Algorithm ANN_c

Having built the cell-tree, we want to use it to solve the original problem. In this section, modifications to algorithm ANN_δ are described that result in ANN_c, an algorithm that solves the all nearest neighbors problem in linear worst-case time, given the cell-tree for the input points. With the cell-tree available, each step of the refinement process of ANN_c can be performed in a time linear in the number of occupied cells. It is no longer necessary at each step to examine every point to determine the cell it occupies. However, cells are still split only into 2^d subcells each time, and a cell is not necessarily refined to its cell-tree children in one step. Without this limitation, Lemma 1 cannot be guaranteed. Therefore, the tree of cells obtained by the algorithm will not be as "branchy" as the cell-tree. On the other hand, with the modifications to be described, if there are k internal nodes (cells) at some level of the computation tree, then there will be $(1 + \beta)k$ nodes after a constant number of generations, for some $\beta > 0$. (By "internal nodes" is meant cells containing more than one point, that will eventually split before the algorithm terminates.) It will also be shown that the amount of work done for all the cells, both the internal cells and those containing just one point, will be linear in the number of internal cells. These two features of the algorithm – the guaranteed branchiness of the computation tree and the constant amount of work per internal cell – yield the linear time bound.

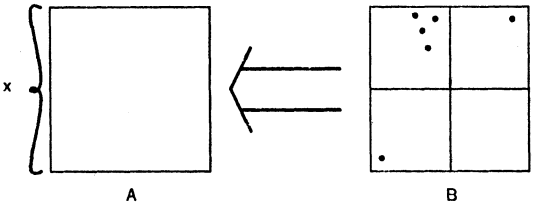
Subsections 3.2.1 and 3.2.2 below are concerned with the modifications to ANN_δ necessary to ensure the branchiness of the computation tree. In 3.2.1, it is shown that groups of cells that are "isolated" from the rest of the cells at a stage in the refinement process can be processed as essentially separate problems. In 3.2.2, it is shown that such isolated groups either contain few cells, or else contain many cells, these cells having few ancestors at the $2d$'th previous refinement step. In the former case, an isolated group with $m \leq 2^d$ cells can be processed such that it has at least $m+1$ descendants in the computation tree. Thus the $2d$'th ancestors of an isolated group will have branchiness bounded away from 1 after $2d + 1$ generations.

In Subsection 3.2.3, algorithm ANN_c is presented and the proof of its time bound completed.

3.2.1 Isolated Subproblems

Call two cells *neighbor-connected* if they share at least one corner, and a set of cells neighbor-connected when there is a neighbor-connected path of cells between any two cells in the set. Suppose that at a certain generation in the refinement process, we determine the neighbor-connected components of the occupied cells. Then any connected component G is isolated from the rest of the cells, in the sense that every cell in G is at least one "cube size" away from all the cells not in G. Now the nearest neighbors of the points within G may be found recursively, and the number of points "looking in" to G for nearest neighbors is linear in the number of cells in G, as is the number of points within the group "looking out" that might have nearest neighbors outside the group. Also, all of these points can be determined in time linear in the number of cells in G. That is, an isolated connected component can be split off from the rest of the cells and solved as a subproblem with, at most, an additional constant cost per cell.

To see that the troublesome points may indeed be found in linear time, consider squares A and B below, with B looking at A for possible nearest neighbor points. Suppose A and B have sides x long, and are x apart.



When the children of B are examined, it is found that one of them contains more than two points. The points in this internal node cell, therefore, have nearest neighbors no more than $x/\sqrt{2}$ away. Hence, they cannot possibly have nearest neighbors in A, since every point in A is at least x away from them.

In general, since the diameter of a cell with sides $x/\sqrt{d}$ long is x, we must subdivide a cell like B at most $\lceil \log_2 \sqrt{d} \rceil$ times, and only leaf cells in the resulting group could possibly have nearest neighbors in A.

Thus, for any pair of cells at least one "cell size" away from each other, we can find the points in each cell that might have nearest neighbors in the other cell in constant time. Since a given cell has a bounded number of cells looking at it for nearest neighbors, the points looking at each cell in an isolated group G can be determined in linear time.

3.2.2 Cell-Tree Nodes of Low Degree and Isolated Groups

The above observations are helpful in dealing with cell-tree nodes of low degree, because if a small group of cells and its descendants have low geometric average degree for more than $2d$ generations of refinement, then the descendants of the group cannot be connected. We have the slightly more formal theorem:

Theorem 1. Suppose a neighbor-connected group G has $|G|$ cells, and the set G' of $2d$'th ancestors of the cells in G has $|G'| > 2^d$ cells. Then $|G| \geq 2|G'|$.

In other words, the ancestors of G up to its $2d$'th ancestors have average degree at least $2^{1/(2d)} > 1$. For example, if G' has $2^d + 1$ cells, then two of those cells must be at least one side length away from each other; thus, after $2d$ generations, with cells of side length only 2^{-2d} of G', there must be at least $2 \cdot 2^d + 1$ cells for those cells to be connected.

To prove this theorem, we use the following lemma, proven in Appendix 1.

Lemma 3. Suppose we are given a set S of k cells in $\mathcal{B}$, all the same size, and a set of points P, with exactly one point for each cell of S. Each point is constrained to lie somewhere within the boundaries of its cell. Then the minimum path length $f(k)$ of a set of arcs connecting the points such that the result represents a connected graph satisfies $f(k) \geq \lceil k/2^d - 1 \rceil / \sqrt{d}$, measured by the side length of a cell in S, for any such set of cells and points.

Proof of Theorem 1. The ancestor group G' will correspond to the set S in the lemma, and neighbor-connected set G will correspond to the connected graph in Theorem 1. Choose any one cell of G per cell of G' to correspond to a point in S. The total path length connecting these cells of G is at least $2^{2d}\lceil k/2^d - 1 \rceil / \sqrt{d}$, measured by the side length of cells in G. Hence the number of neighbor-connected cells $|G|$ clearly exceeds $2k = 2|G'|$, for $|G'| > 2^d$. ∎

Note that the theorem does not say anything about what can happen when G' has no more than 2^d cells. In this case, a group of 2^d neighbor-connected cells could be arranged around a single point in d-space, and even if they have only one child each, those children can still be connected. Therefore, to ensure that the average degree of nodes remains above one, we will add the following step to our algorithm: whenever an isolated connected component is found with at most 2^d cells, we "shrink" it by finding the maximum size of the cell-tree descendants of those cells, and begin processing the component by refining its cells to that maximum size. This guarantees that at the next generation of refinement for the resulting cells, there will be $|G'| + 1$ cells present. Thus, for a connected component with no more than 2^d cells, an average degree of $1 + 1/2^d$ can be guaranteed. We also have the following theorem.

Theorem 2. For the refinement process with shrinking, a connected component group G with k cells has at least $k(1 + 1/2^d)$ descendants after $2d + 1$ generations.

Proof. If $k \leq 2^d$, G will be "shrunken" and have $k + 1$ descendants at the next generation, as noted. If $k > 2^d$, consider the connected components of its descendants after $2d$ generations. If such a component has at most 2^d cells, it will be shrunken and have at least one more cell in the next generation. Therefore the component's $k' \leq 2^d$ ancestors in G have at least $k'+1$ descendants after $2d+1$ generations. If a descendant connected component has more than 2^d cells, then by Theorem 1, it cannot have too many ancestors: if it had k' ancestors, with $k' \leq 2^d$, then those ancestors had average degree above one; if the connected component had $k' > 2^d$ ancestors, then those ancestors must have at least $2k'$ descendants at generation $2d$ for those descendants to form a connected component. ∎

3.2.3 Algorithm ANN_c

Algorithm ANN_c, a modification of ANN_δ using a cell-tree to solve the all nearest neighbors problem, is as follows. The basic process of ANN_δ is used; in addition the splitting into subproblems described in 3.2.1 and the "shrinking" of small neighbor-connected components described in 3.2.2 are used. These modifications are reflected in the use of procedures *Points_Looking_In* and *Shrink* in the pseudo-code below. We also assume the definition of a procedure *Update_Point_NNs* analogous to *Update_NNs*, as described in Section 2. The input to the algorithm is a set of cells S, together with a set of points P whose nearest neighbors within the cells are to be found. Each point, whether in P or within a cell in S, has an associated estimate of its nearest neighbor distance. The revision of such estimates is the output of the algorithm. The initializing procedure calling ANN_c is omitted.

```
procedure ANN_c(S : Set_of_Cells ; P : Set_of_Points);
begin
  S' ← ⋃_{C ∈ S} {C' | C' ∈ Children(C), C' occupied};
  for each C' ∈ S' do Update_NNs(C');
  for each p ∈ P do
    Update_Point_NNs(p);
    if Contents(C) is a single point, for all C ∈ NN_Set(p)
      then output revised NN estimate for p; P ← P − {p} fi;
  od;
  determine neighbor-connected components of S;
  for each component G do
    G_Points ← Points_Looking_In(G)
            ∪ {p ∈ P | ∃C ∈ G with C ∈ NN_Set(p)};
    if |G| ≤ 2^d then Shrink(G) fi;
    ANN_c(G, G_Points);
  od;
end;
```

To show that ANN_c requires linear time, we need to show, in addition to Theorem 2, that points from leaf cells, and points resulting from *Points_Looking_In*, do not result in more than a constant amount of work per internal node cell. Since leaf cells arise from cells at least as large as a given internal cell C being processed, there is a bounded number of such cells surrounding C closer than its diameter. Therefore, since C contains at least two points, there are a bounded number of such leaf cells that might contain nearest neighbors to points in C. Furthermore, points farther from C than its diameter, and looking at C as possibly containing nearest neighbor points, must be no closer to each other than they are to C; hence there must be a bounded number of such points looking at C. These considerations apply as well to points from nearby components that might have nearest neighbors in C. Therefore, isolated points that arise in the course of the algorithm cost a constant amount per occupied internal node cell. This concludes our proof of the linearity of the algorithm.

§4 Algorithm ANN_e

In this section, it is assumed that the input points are random, and are independently identically distributed. Further, it is assumed that the (unknown) probability density function has a finite number of poles, and is $O(1/r^{d+1})$, where r is distance from the origin. The described conditions are very nearly comparable to the best known for sorting ([Dev],[Lue]). By comparison, the algorithm described by Bentley et al. [BWY] is linear only when the density function is zero outside a bounded region, and obeys a smoothness condition.

A method of building a cell–tree in linear expected time under the given conditions will be described. Having built this data structure, the all nearest neighbors problem can be solved in linear time using ANN_c, described in the previous section.

A cell–tree "scheme" will be indicated, having the following properties:

- there are $O(n)$ leaves;
- each leaf contains $O(1/n)$ probability mass, unless the leaf contains a pole;
- the total mass contained in the tree is $1 - 1/n$;
- it is possible in $O(1)$ time to find the leaf, if any, in the tree that contains a given point, using address calculation techniques requiring the floor function.

Because of these properties, the expected work per leaf in building the tree is $O(1)$, except for leaves containing poles. For those leaves, the expected work is $O(n)$, if the fast cell–tree procedure described in Section 3.1 is used.

Since it is assumed that the probability density function is $O(1/r^{d+1})$, we further assume WLOG that the pdf is dependent on the l_∞ distance to the origin r, and takes the form

$$f(r) = \begin{cases} 1/2^{d+1}, & r < 1; \\ 1/d(2r)^{d+1}, & r \geq 1. \end{cases}$$

Let $F(r)$ be the function indicating the probability mass between 0 and r. Then with these assumptions, $F(r)$ is simply $1 - 1/2r$, for $r \geq 1$, so that the mass outside of the region with $r \leq n/2$ is just $1/n$. Within this region, we consider each layer $n/2^{k+1} \leq r \leq n/2^k$, for integer $1 \leq k < \lfloor \log_2 n \rfloor$. Each such region contains mass $2^k/n$, and can be partitioned into $2^d(d+1)$ equal-sized hyper-cubes. Hence if each such cube is considered a cell-tree node, its children after no more than $\log_2 2^k/2^d(d+1)$ generations will have $\Theta(1/n)$ probability mass each. Also appropriately subdividing the cube of points with $r \leq 2^{\log_2 n \pmod 1}$ gives a cell-tree scheme satisfying the first three properties above.

Verification of the last property of this scheme is straightforward, and is omitted. The proof of a linear bound in the presence of poles follows almost exactly as in Lueker's work [Lue], and is also omitted.

§5 Concluding Remarks

Some further questions regarding these results:

- How well do the algorithms perform in practice, especially as compared with other algorithms for the problem? This would be interesting to know in comparison with the algorithm given by Bentley [Ben], especially in view of Monier's results [Mon] showing that the constant factor dependence on d for multidimensional divide-and-conquer algorithms is quite small.

- The algorithms described give *all* the nearest neighbors of a point, so that an exponential dependence on the dimension is inevitable. Is there an algorithm that gives only one nearest neighbor, and does not have a time bound exponential in the dimension?
- The compressed form of hyperoctrees, called cell-trees here for brevity, might well have broader application for geometrical problems. They require linear space, unlike hyperoctrees, yet maintain geometrical information more "accurately" than k-d trees. The author has recently found an algorithm for constructing minimum spanning trees in the plane, under the l_1 and l_∞ distance measures, such that finding a supergraph of the minimum spanning tree with $O(n)$ edges requires linear worst-case time, given a cell-tree for the input points.

Acknowledgments. I would like to thank my advisor, Andy Yao, for suggesting the problem and for helpful discussions and encouragement. I would also like to thank Andrei Broder, Leo Guibas, Ernst Mayr, and Alan Siegel for helpful discussions, and Rhonda Cooperstein for editorial assistance.

References

[Ben] J. L. Bentley. Multidimensional divide-and-conquer. *CACM,* Volume 23 (1980), Number 4, pp. 214-229.

[BFP] J. L. Bentley, M. G. Faust, and F. P. Preparata. Approximation Algorithms for Convex Hulls. *CACM,* Volume 25 (1982), Number 1, pp. 64-68.

[BWY] J. L. Bentley, B. Weide, and A. C. Yao. Optimal expected-time algorithms for closest-point problems. *ACM Tran. Math. Software,* Volume 6 (1982), Number 4, pp. 563-579.

[Dev] L. Devroye. Average time behavior of distributive sorting algorithms. Tech. Rep. No. SOCS 79.4, School of Computer Science, McGill University, 1979.

[Knu] D. Knuth. *The art of computer programming, Volume 3: sorting and searching.* Reading, Mass.: Addison Wesley, 1973.

[Lue] G. Lueker. A survey of address calculation techniques with unknown input distributions. Unpublished manuscript, 1983.

[Mon] L. Monier. Combinatorial solutions of multidimensional divide-and-conquer recurrences. *Journal of Algorithms,* Volume 1 (1980), pp. 60-74.

[Sam] H. Samet. The quadtree and related hierarchical data structures. Tech. Rep., to appear, Computer Science Department, University of Maryland, College Park, Maryland.

[Sha] M. I. Shamos. Computational Geometry. Ph.D. Dissertation, Yale University, New Haven, Conn., 1978.

[Yao] A. C. Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal of Computing,* Volume 11 (1982), Number 4, pp. 721-736.

[YS] M. Yau, and S. N. Srihari. A hierarchical data structure for multidimensional images. *CACM,* Volume 26 (1983), Number 7, pp. 504–515.

Appendix 1

Proof of Lemma 1. We will show that the number of cells for which a given cell can be a nearest neighbor is bounded by a constant that is dependent on the dimension. Note that if a cell contains more than two points, those points cannot have nearest neighbors farther away than the cell diameter. Therefore, a cell C cannot be nearest neighbor to more than a constant number of such cells, and attention may be restricted to bounding the number of one-point cells for which C can be a nearest neighbor. Since the number of cells sharing a corner with C is bounded, it can be assumed that the points under consideration are farther away from C than the side length of a cell. The situation can be described as follows: a set of points A is contained in a sphere S_r centered at the origin with radius r, and another set of points B is outside a sphere S_R centered at the origin with radius R, where $R > r$. Each point of B has as nearest neighbor at least one of the points of A. To prove the lemma, the number of points in B must be bounded.

Let the nearest neighbor (NN) ball about a point be the closed ball centered at the point with radius equal to the distance to that point's nearest neighbor. It is clear that no point can be inside the surface of another point's nearest neighbor ball, and that the nearest neighbor balls of points in B must have non-empty intersection with S_r.

It will be shown that, for any configuration of points of B, there is another collection B' with the same number of points such that, in addition to the above conditions, all of the points of B' are on the surface of S_R, and all of the corresponding nearest neighbor balls are tangent to S_r. Now the number of points of B' must be bounded: the intersection of a NN ball of B' with S_R is a "spherical disk." The ratio of the surface area of S_R to the area of such a disk is bounded above zero, since the ratio $R/(R-r)$ of the radius of S_R to that of a NN ball is bounded above zero. Therefore, the number of disks that can be packed onto the surface of S_R is bounded, as is $|B'| = |B|$, the number of centers of such disks.

It remains to show how a given configuration of points B can be transformed into a set of points B' satisfying the additional assumptions above. Clearly the radius of a NN

ball can be reduced until the ball is tangent to S_r, without violating other conditions on B. Therefore, for a given configuration of points A and B, there is another with the same B points (possibly changing A), and with the NN balls of B tangent to S_r.

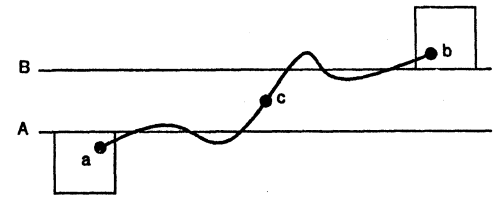
Finally, in order to determine the points of B', carry out the following transformation on B. Consider the point p of B with NN ball of largest radius. If p is moved in a straight line toward the origin, with its NN ball maintaining the same point of tangency with S_r, then its NN ball will stay within its original bounds. Therefore, the nearest neighbor of p will continue to be in S_r. However, in moving p toward the origin in this way, it may be that p becomes the nearest neighbor of some other point q. This cannot occur, however, unless the NN ball radius of p becomes smaller than that of q. Therefore, at the point at which the two radii are equal, no nearest neighbor condition has been violated. At this time, the two points may simultaneously be moved in straight lines toward the origin. This process would continue until the two radii are equal to that of the NN ball of some other point, at which time that point would move toward the origin, and so on, until all points are on S_R. The result is the point set B', having the same number of points as B, this number being bounded, as indicated above. ∎

Proof of Lemma 2. (Sketch) Consider the set of maximal unrelated (with respect to containment) cells of $\mathcal{B}$ each with less than ϵ/m of the input points, where m is some value greater than 1 to be determined later. If all of these "light" cells each contain less than an ϵ proportion of the sample points, then *Build_Cell* cannot choose a cell C' lighter than ϵ/m upon which to split. If, in addition, the minimal cell with more than a $1-\epsilon/m$ proportion of the input points contains more than a $1-\epsilon$ proportion of the sample points, then *Build_Cell* must split its task into pieces each of size greater than about $(\epsilon/m)n$ and less than $(1-\epsilon/m)n$. The probability of these events occurring is positive and independent of n, resulting in the desired bound. ∎

Proof of Lemma 3. To prove lemma 3 of Section 3.2, we will show that the total path length, measured with the l_1 distance, is at least $\lceil k/2^d - 1 \rceil$. Dividing this by $\sqrt{d}$ yields the bound for the Euclidean case.

The proof is by induction. The cases $k \leq 2^d$ are trivially true. To prove the result for $k > 2^d$, it will be shown that any connected graph for P can be disconnected into two connected graphs on two point sets, with a total path length savings of at least 1. This implies $f(k) \geq f(m) + f(n) + 1$, for $m + n = k$, which implies the desired bound.

Now given a connected graph on some $k > 2^d$ points, with exactly one point inside each cell of S, there must be two cells at least distance 1 apart, that is, separated by a cell side length. This implies a situation like that shown:



Now since there must be a path connecting a and b, that path must cross the region between hyperplanes A and B. Consider the portion of the path crossing the hyperplane. If there is no vertex on this portion of the path, that portion can be deleted, saving at least 1 side length in distance. If there is a vertex c along that path, then we do the following: delete the arc connecting c to a's component, and move c toward the B hyperplane. Also move all the points in the path from c to the B hyperplane toward that hyperplane. By so doing, a total of 1 side length in path length is saved, using the l_1 distance measure. Furthermore, the two connected components have minimal path length if the original graph did. Therefore, $f(k) \geq f(m) + f(n) + 1$, as desired. ∎

Appendix 2. Procedure *Smallest_Cell*

Given a set of points S, it is desired to find the smallest cell in the family $\mathcal{B}$ that contains those points. Observe that every cell at a level of subdivision k has the form $\bigotimes_{1 \leq j \leq d} [i_j 2^{-k}, (i_j + 1)2^{-k})$, where each i_j satisfies $0 \leq i_j < 2^k$. Let point p^{min} have j'th coordinate satisfying $p_j^{min} = \min\{p_j \mid p \in S\}$, and similarly define p^{max}. Then clearly a cell contains S if and only if it contains p^{min} and p^{max}. So for each coordinate, the largest k is sought such that there is a suitable i_j with $i_j 2^{-k} \leq p_j^{min} \leq p_j^{max} < (i_j + 1)2^{-k}$. Thus if function G computes such a value k, the subdivision level b of the cell to be found is equal to one less than $\min_{1 \leq j \leq d} G(p_j^{min}, p_j^{max})$. Given this value b, the cell desired has $i_j = \lfloor p_j^{min} 2^b \rfloor$. All of these bitwise computations can be computed easily using the floor, logarithm, and bitwise exclusive-or functions.