If the sets of sites S and T are in quadtree cells A and B that satisfy conditions like these, the NFN problem for their sites can be solved in linear time. Using this observation in the straightforward recursion useful for the planar case results in a non-linear algorithm, since many more subproblems are generated than in that case. Suppose, however, that cells A and B correspond to the same cube, or are two different equal-sized descendents of the same quadtree cell, so that they are aligned in one or more dimension. Then at a given level of refinement, for a cell C in A, the cells "pending" for C will be aligned with it in at least one coordinate. That is, for the cells the size of C in B, all but those aligned with C can have their NFN site with it determined in constant time, using Lemma 2.4$'$. (The assumption is made here that the quadtrees rooted at A and B are preprocessed so that for each cell, the closest site to each corner of that cell is known.)

An algorithm using this observation is shown in pseudo-code in Figures 2.8a and 2.8b below. The heart of the algorithm is procedure *Exact_NFN*. Given quadtree cells A and B, this procedure processes all quadtree cells in them down to the *number_of_generations* level. After this processing, for every site s in such a cell C in A, closest sites to s are known, among all sites in cells of B not pending for C. The processing is performed for all cells in A and B of a given size, and then the information gained is used to determine the corresponding information for the children of those cells.

Procedure *Exact_NFN* organizes equal-sized quadtree cells into layers in each of the coordinate directions. Each cell appears in a layer of such cells aligned in the x_1 coordinate, in another layer of cells aligned in the x_2 coordinate, and so on, being represented at most d times. An aligned layer in A can be paired with an aligned layer in B, such that all cells in the pair of layers are aligned. Such aligned pairs of layers will be termed *pending pairs*. The information gained by *Exact_NFN* can be described in this terminology as follows: After the processing of C within the j-loop in *Exact_NFN,* closest sites to any site in C are known, among the sites in B not in

$$\bigcup_j \{s \mid s \in D, D \in Pending.B, C \in Pending.A, Pending \in Pending\_Pairs_j\}.$$

Such closest sites are known for each corner of C. The closest site for a corner is the closest among sites for which that corner is a "separating point" from C, as in Lemma 2.4$'$.

We next describe how to preserve this condition, moving from one level of refinement of quadtree cells to the next. This "splitting step" is performed by *Split_Pending_Pair,* shown in Figure 2.8b. For example, a given pending pair aligned in x_j at a particular step gives rise to two "child layers," consisting of those children with higher x_j coordinate, and those children with lower x_j coordinate. The high child layer in A forms a new pending pair with a high child layer in B, and similarly for the low layers. The two pairs of child layers that are separated

```
procedure NFN(A, B : quad_tree_cell,
      c : corner,
      number_of_generations : integer); co of cells to consider oc

  co A corner of a cell is described using an array, indexed from 1 to $d$, of values that are High,
  Low, or Aligned. If $c_j$ is High(Low), this indicates that $B$ is above(below) $A$ in the $j$ dimension.
  If $c_j$ is Aligned, $B$ and $A$ are aligned in the $j$ dimension. The corner $\overline{c}$ has $\overline{c}_j$ Low when $c_j$ is
  High, and vice versa, and is Aligned when $c_j$ is Aligned.

  Pending_Pairs is an array, indexed from 1 to $d$, of sets of pending pairs. A pending pair is a pair
  of sets of quadtree cells that are aligned along a dimension. Pending pair Pending has the set of
  cells Pending.A from $A$ and Pending.B from $B$.

  For a cell $C$ in $A$ and a corner $c$, $C.my\_closest_c$ denotes the closest site in $C$ to the corner indicated
  by $c$. The site $C.other\_closest_c$ is a closest currently known site in $B$ to the corner of $C$ indicated
  by $c$. oc;
begin
  Preprocess $A$ and $B$ so that all their descendents to number_of_generations have defined my_closest
  fields;
  Pending_Pairs ← Exact_NFN(A, B, c, number_of_generations);
  Projected_NFN(Pending_Pairs, c, number_of_generations);
end;
procedure Projected_NFN(Pending_Pairs : Pending_Pair_Array,
      c : corner,
      number_of_generations : integer);
begin
  for $j$ with $c_j$ == Aligned do
    for Pending ∈ $Pending\_Pairs_j$ do
      for $C$ ∈ Pending.A do make lower-dim. cell $C'$ from $C$ by projecting sites along $j$th coordinate
      od;
      for $C$ ∈ Pending.B do make cell $C'$ by projecting along $j$th coordinate od;
      Make quadtree $B'$ from from Pending.B similarly;
      Make $(d-1)$-corner $c'$ by deleting $j$th coordinate value from c;
      NFN(A', B', c', Number_of_generations);
      for $C$ ∈ Pending.A do
        Update $C.other\_closest_c$ with the site corresponding to $C'.other\_closest_{c'}$;
      od;
      for $C$ ∈ Pending.B do
        Update $C.other\_closest_{\overline{c}}$ with the site corresponding to $C'.other\_closest_{\overline{c'}}$
      od;
  od;od;
end;
```

Figure 2.8a. Procedures NFN and Projected_NFN.

in the x_j coordinate represent $(d-1)$-dimensional problems. The closest sites for these diagonal layers can be recursively determined, using *Exact_NFN*. This recursion, performed for each of the diagonal layers, yields closest sites for a cell C in a child layer of A, among each of the diagonal layers for C. Comparing these closest sites with that of the parent cell of C yields the closest sites in B to C, among those sites not in the pending pairs containing C. It follows that the inductive condition is preserved.

After refining *number_of_generations* times, it remains to determine closest sites among those sites in A and B each in a pending pair. This can be done approximately by "collapsing" each pending pair to a $(d-1)$-dimensional subproblem, and

```
function Exact_NFN(A, B : quad_tree_cell,
      c : corner,
      number_of_generations : integer)
return Pending_Pairs : Pending_Pair_Array;
begin
 if $c_j \neq$ Aligned for all $j$ then
  Update A.other_closest$_c$ with B.my_closest$_{\bar{c}}$;
  Update B.other_closest$_{\bar{c}}$ with A.my_closest$_c$;
 else
  Pending.A ← {A}; Pending.B ← {B};
  for $j$ with $c_j$ = Aligned do  Pending_Pairs$_j$ ← {Pending} od;
  for $i$ ← 1 to number_of_generations do
   for $j$ with $c_j$ = Aligned do
    New_Pending_Pairs$_j$ ← {};
    for Pending ∈ Pending_Pairs$_j$ do
     [Low_Pending, High_Pending] ← Split_Pending_Pair(Pending, c, i, j);
     New_Pending_Pairs$_j$ ← New_Pending_Pairs$_j$ ∪ Low_Pending ∪ High_Pending;
   od;od;
   Pending_Pairs ← New_Pending_Pairs;
   co A pruning step might be performed at this point oc;
  od;
 fi;
end;
function Split_Pending_Pair(Pending:Pending_Pair,
      c : corner,
      level : integer, co of refinement of cells in Pending oc
      j : dimension)
return [Low_Pending, High_Pending : Pending_Pair];
begin
 Low_Pending.A ← {C | C ∈ Children(D), D ∈ Pending.A,
  C has lower jth coordinate among such children};
 Determine Low_Pending.B, High_Pending analogously;
 Give cells in Low_Pending and High_Pending the other_closest sites of their parents;
 co Solve the problems resulting from the layers separated in $x_j$: co
  Make quadtrees AL, BL from cells in Low_Pending;
  Make quadtrees AH, BH from cells in High_Pending;
  cH ← c; cH$_j$ ← High;
  Exact_NFN(AL, BH, cH, level + 1);
  cL ← c; cL$_j$ ← Low;
  Exact_NFN(AH, BL, cL, level + 1);
end;
```

Figure 2.8b. Procedures Exact_NFN and Split_Pending_Pair of NFN.

solving that problem recursively, as in *Projected_NFN,* shown in Figure 2.8a. The result will be that the NFN distances are determined within $d\alpha'$, where α' is the side length of the smallest quadtree cells considered. The main algorithm *NFN* simply initializes the quadtrees, calls *Exact_NFN,* and then calls *Projected_NFN.* From the above discussion, we have the following theorem.

Theorem 2.5. Let A and B be quadtree cells that correspond to the same cube, or are equal-sized distinct descendents of the same quadtree cell. When called for these quadtree cells, and with *number_of_generations* set to some value j, algorithm *NFN* determines the nearest foreign neighbors for sites in A and B to accuracy $d\alpha'$, where $\alpha' = 2^{-j}$. The sidelength of A and B is the unit of measure of accuracy.

Using induction on the dimension, we will show that the time needed by *NFN* and *Exact_NFN*, for processing cells to the *number_of_generations* $= \lg(d/\alpha)$ level, is $O(k \log^d(1/\alpha))$, for k descendents of A and B at the lowest level. This results in a final accuracy of α. Using the inductive hypothesis, *Split_Pending_Pair* requires $O(\log^{d-1}(1/\alpha))$ work for each cell in the pending pairs input to it, since the work per cell is dominated by the work for a $(d-1)$-dimensional call to *Exact_NFN*. Therefore, *Exact_NFN* requires $O(\log^{d-1}(1/\alpha))$ for each quadtree cell in A and in B. Since there are $O(k \log(1/\alpha))$ such cells, the total work in *Exact_NFN* is $O(k \log^d(1/\alpha))$. It follows that the work in *NFN* is $O(k \log^d(1/\alpha))$, plus the time required for *Projected_NFN*. *Projected_NFN* requires linear time, plus the time for $(d-1)$-dimensional calls to *NFN*, with each of the k cells in exactly d of those calls. The work performed by *NFN* is therefore $O(k \log^d(1/\alpha))$. Since $k = O(n+m)$, where $n = |Contents(A)|$ and $m = |Contents(B)|$, the total work is $O((n+m) \log^d(1/\alpha))$. We have proven

Theorem 2.6. Algorithm *NFN* requires $O((n+m) \log^d(1/\alpha))$ time to solve the nearest foreign neighbor problem for quadtree cells containing n and m sites, under the conditions of Theorem 2.5.

When only the closest NFN pair is desired, and not the NFN distance for every site, a modification can be made in the algorithm that would probably improve its performance in practice. This modification is also important for achieving a better time bound in the MST algorithms of the following section. The change is to move procedure *Projected_NFN* into *Exact_NFN*, so that an upper bound on the NFN pair distance can be known at each step. If this is done, then any cell examined in the course of algorithm that can be ruled out using this upper bound may be "pruned", and its descendents not examined subsequently. The result is a smaller number of cells to examine, and the guarantee that each cell examined contains an approximate nearest neighbor site.

§2.3 Minimum Spanning Trees

In this section an algorithm will be described for computing an approximation to a minimum spanning tree for a set of sites. In §2.3.1, the problem of computing an approximate MST is reduced to that of solving the nearest foreign neighbor pair problem approximately. In §2.3.2, this reduction is further refined, and time bounds are given for the case when the L_1 NFN algorithm of the last section is used. In §2.3.3, these ideas are illustrated with an algorithm for finding MSTs in the 3D Euclidean case.

2.3.1 MST $\Rightarrow$ NFN

Several of the algorithms for geometric minimum spanning trees mentioned in §1.1.3 find an MST supergraph G' with $O(n)$ edges, using Voronoi diagrams, relative

```
procedure Find_MST_Edges(P);
co P is a connected component of cells oc;
begin
 Cell_Diameter ← diameter of a cell in P;
 S ← quadtree children of P;
 for each connected component $P_i$ of S do
  Find_MST_Edges($P_i$);
  for each cell C in $P_i$ do
   for each cell D in S not in $P_i$
        with $d_{\min}(C, D) \leq 2$Cell_Diameter do
    for each edge-cell A of C do
     for each edge-cell B of D do
      Find the NFN pair for A and B and add corresponding edge to $G'$;
 od; od; od; od; od;
end.
```

Figure 2.9. The reduction from MST to NFN.

neighbor graphs, or nearest geographic neighbor graphs. In this section, it will be shown that it is possible to find an MST supergraph by solving a number of nearest foreign neighbor pair problems. A constant number of NFN pair problems are solved for each cell, down to the singleton cell size, of the quadtree for the input sites. Each NFN pair subproblem results in one edge of G', so that the resulting G' has $O(n \log \sigma)$ edges, since the number of quadtree cells examined is $O(n \log \sigma)$. This reduction can be used for sites in a space of any dimension d, and with any L_p metric.

Furthermore, using this reduction, an algorithm that solves the nearest foreign neighbor pair problem approximately can be used to find an MST approximately, so that the algorithms of the last section can be used. In this respect the reduction given is more useful than the more direct one involving Borůvka's algorithm ([Tar2], p. 72).

Once an MST supergraph is found, the MST may be found using the algorithm of Fredman and Tarjan [FT]. An alternative procedure is to sort the edges by length and then use Kruskal's algorithm. When finding an approximate MST within $1+\rho$ in weight of an MST, only $\log \sigma + \log(1/\rho)$ bits of precision are needed to represent the weights of the edges. Hence a radix sorting procedure ([Knu], §5.2.5) may be performed, in time $O(n(\log \sigma + \log(1/\rho)))$, before employing Kruskal's algorithm.

A pseudo-code version of the algorithm *Find_MST_Edges* is shown in Figure 2.9. In order to understand its operation, we need the notion of a neighbor-connected component of quadtree cells:

> Consider all the cells in a quadtree of a given size. Call two such cells *neighbor-connected* if they share at least one corner. The *neighbor-connected components* of the cells are the connected components of the resulting implied graph.

Let T be the root cell of the quadtree containing the input sites S. Procedure *Find_MST_Edges* is called with the neighbor-connected component $\{T\}$, and recursively calls itself with neighbor-connected components as input. Specifically, if

P is the input component, then *Find_MST_Edges* calls itself with the neighbor-connected components $P_1, P_2, \ldots, P_k$ of the children of the cells in P. (For example, see Figure 2.10.) After *Find_MST_Edges* processes P, no further edges are found within P by the algorithm. That is, no edge corresponding to a pair of sites that are both in P is added to G'. This implies that after calling itself for $P_1, P_2, \ldots, P_k$, the only edges *Find_MST_Edges* will add to G' from P will be between these child components. These edges will correspond to pairs of sites s and t, with s in P_i, t in P_j, and $i \neq j$. Procedure *Find_MST_Edges* does not find all possible such edges, however. We will show that the edges not added to G' need not be in an MST supergraph for G.

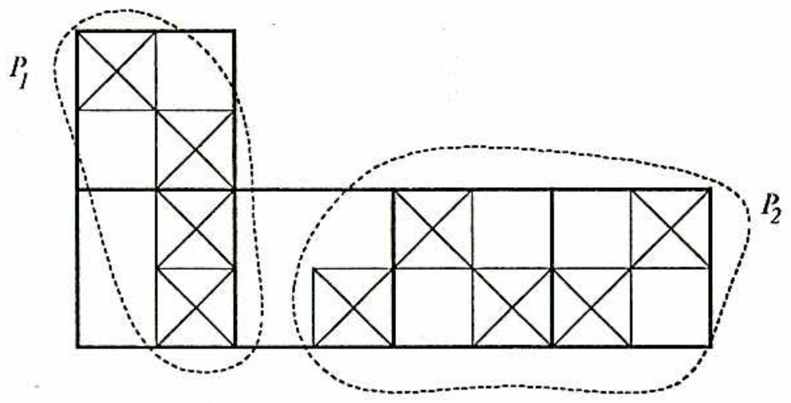

Figure 2.10. Finding edges between connected components.

One restriction that *Find_MST_Edges* places on the edges between child components is that such edges be between cells that are close together. Let the diameter of a cell in P be *Cell_Diameter*. Then *Find_MST_Edges* finds no edges between cells C and D with $d_{\min}(C, D) > 2\,\mathit{Cell\_Diameter}$. We will show that such edges cannot be in an MST for G, and therefore need not be included in G'. Consider sites $s \in C$ and $t \in D$. Since P is a neighbor-connected component, there is a path in G between s and t consisting of edges that are all no longer than $2\,\mathit{Cell\_Diameter}$. Therefore, since $\|s - t\|$ is greater than $2\,\mathit{Cell\_Diameter}$, the edge corresponding to the pair of sites s and t is the uniquely longest edge in a cycle of G. By Fact 1.3, such a longest edge is not in any MST of G. It follows that when finding edges between child components of P, *Find_MST_Edges* need only consider edges between sites in cells that are no farther apart than $2\,\mathit{Cell\_Diameter}$.

Suppose C is in child component P_i, D is in child component P_j, $i \neq j$, and $d_{\min}(C, D) \leq 2\,\mathit{Cell\_Diameter}$. Then *Find_MST_Edges* returns only $O(1)$ pairs of sites from C and D as edges of G'. These are edges from *edge-cell* pairs. Edge-cells are defined as follows:

An edge-cell of a cell $C \in P$ is a largest descendent cell of C that has a diameter less than half the sidelength of C.

That is, if A is an edge-cell of C, *Diameter*(A) is less than half the side length of C, and the parent of A has a diameter greater than this value. Note that if A is an edge-cell of C, and B is an edge-cell of D, then $d_{\min}(A, B)$ is larger than *Diameter*(A) = *Diameter*(B), since C and D are in different child connected components. We will show that only one edge between A and B need be included in G'. Suppose that $\{s,t\}$ and $\{s',t'\}$, with $s, s' \in A$ and $t, t' \in B$, are two edges between A and B. Then $\|s' - s\| < \|s - t\|$, and $\|t - t'\| < \|t' - s'\|$, so that $\{s,t\}$ and $\{s',t'\}$ are the two longest edges in a cycle of G. Therefore, by Fact 1.3, at most one of these edges need be included in G'. This reasoning implies that only the edge corresponding to an NFN pair for A and B need be included in G'.

We have shown that when processing component P, the only edges that procedure *Find_MST_Edges* need find between child components are those corresponding to NFN pairs from the pairs of cells A and B, where: A is an edge-cell of C, B is an edge-cell of D, C and D are in different child components, and $d_{\min}(C, D) \leq$ 2*Cell_Diameter*. The following lemma results by induction:

Lemma 2.7. When called with quadtree cell $\{T\}$ as input and using an exact NFN algorithm, procedure *Find_MST_Edges* finds an MST supergraph for the sites contained in T.

Suppose that instead of an exact NFN algorithm, an approximate one is used. Note that NFN pairs are found between edge-cells that are farther apart than their diameter. The absolute error in the NFN pair found, measured as a fraction of the input cell diameter, is therefore a relative error in the length of the edge found. Instead of an MST supergraph, an "approximate" MST supergraph results:

Lemma 2.8. If an NFN algorithm is used in *Find_MST_Edges* that has an absolute error α relative to the problem scale, then the set of edges returned will contain a spanning tree with weight within $1 + \alpha$ of the minimum.

The size of the supergraph produced is $O(n \log \sigma)$, where $n = |$*Contents*$(T)|$. It is easy to see that each edge-cell contains endpoints of a bounded number of edges of the resulting MST supergraph, and the number of edge-cells is proportional to the number of quadtree cells.

2.3.2 Bounding the time for NFN subproblems

The reduction above, when applied using a linear time NFN algorithm, can take $\Omega(n^2)$ time, since there are $\Omega(n)$ subproblems to be solved for the MST problem, some with $\Omega(n)$ sites involved. In this section a modification of the above reduction, together with a more careful accounting, will yield an MST algorithm with the bound claimed. We will first sketch the main idea behind this modification, assuming that the L_1 NFN algorithm of the previous section is used.

The modification to *Find_MST_Edges* is a pruning step that removes certain quadtree cells from consideration in the course of solving NFN pair subproblems.

As a result of this pruning step, each quadtree cell (down to singleton-cell size) is examined within NFN pair subproblems a total number of times that is bounded by a constant dependent on the dimension. The time required by *Find_MST_Edges* therefore is proportional to the time required per quadtree cell by the NFN pair algorithm, multiplied by the number of quadtree cells examined, which is $O(n \log \sigma)$.

The crucial effect of the pruning step, that implies the bound on the number of times a quadtree cell is examined, is that a certain region near an examined quadtree cell contains no sites. For example, suppose that an NFN pair is desired between edge-cells A and B, and cell a in A may have an NFN pair with cell b in B. Then the region $\mathcal{B}_a \cap \mathcal{B}_b$, as shown in Figure 2.13 (p. 32) will not contain any sites (if the pruning step is used). (The nature of regions $\mathcal{B}_a$ and $\mathcal{B}_b$ will be described below. For the L_2 MST problem, these regions will be spheres, as shown. For the L_1 case, these regions will be L_1 balls, which are octahedra.) This implies that no NFN subproblem will be solved for another (smaller) edge-cell containing a, and an edge-cell in $\mathcal{B}_a \cap \mathcal{B}_b$. After a bounded number of NFN subproblems examining a, a region completely surrounding a will be empty of sites, and there can be no edge-cells for which there is an NFN pair problem involving a.

We will first show how the NFN algorithm used by *Find_MST_Edges* may be modified so that a quadtree cell a in edge-cell A is examined only if there is a region in A that is empty of sites. We next show how a pruning step involving several edge-cells may be used, so that a larger region near an examined quadtree cell will be empty of sites.

When an L_1 MST is desired, and algorithm *NFN* of §2.2 is used, then the modification mentioned is simply that described at the end of §2.2.2. With this change, at each level of refinement of *NFN*, only those cells are considered that contain sites that are approximately nearest neighbors. Suppose the NFN pair problem is being solved for edge-cells A and B, where A is in some neighbor-connected component P_i. Then for cell a in A examined at a step, a region near a and containing cell B is free of sites of A. If a contains an approximate NFN pair with cell b in B, this region consists of points known to be closer (within the current accuracy) to sites in b than are the sites in a. If there were a site in that region, a would have been eliminated earlier in the processing of the algorithm, and not examined. Thus, further NFN calls for which a is examined cannot come from that region in A.

When another approximation algorithm for the NFN pair problem is used, this condition is not so directly obtained, since such an algorithm may not used quadtrees at all. However, a similar condition may still be obtained, by using the given algorithm as a "black box" within a quadtree-based algorithm. In such an approach, when edge-cells A and B are input, the black box is called with an input consisting of representative points of the quadtree children of A and B, and the NFN pair distance determined for those points. Such representative points might be, for example, the centers of those quadtree children. Using the NFN pair distance estimate resulting, some of the quadtree children may be eliminated from consid-

```
    *For all edge-cell pairs of a given size in processing a connected component, determine
    the required NFN values in parallel as follows. Include in the MST supergraph all edges
    between NFN pairs for active edge-cells.*
for $i \leftarrow 1$ to $number\_of\_generations$ do
  for each active edge-cell pair $\{A, B\}$ do $pathmax(A, B) \leftarrow NFN_{\max}(A, B)$ od;
  for each active edge-cell pair $\{A, B\}$
  and edge-cell $A'$ with $d_{\max}(A, A') \leq 2Diameter(A)$ do
    if $NFN_{\min}(A, B) > pathmax(A', B)$ then
      make $\{A, B\}$ inactive;
      $pathmax(A, B) \leftarrow pathmax(A', B)$;
    fi;
  od;
od;
```

Figure 2.11. The MST pruning step.

eration, if they provably cannot contain a site from an NFN pair. This process is then repeated for these children of those cells, and so on. Although some time may be wasted at each step, since the black box algorithm works from scratch each time, some speed may be gained due to the described elimination of "fruitless" quadtree cells. In addition, just as with the modified L_1 NFN algorithm described above, for each examined quadtree cell there will be an empty region near it in the edge-cell containing it.

Thus no matter what NFN pair approximation algorithm is available, the emptiness condition for an examined quadtree cell will hold. We will assume in the following that an algorithm *Approximate_NFN* is available, with processing that results in this condition. Now we will show that a further pruning step will result in the stronger condition described above.

The pruning step is shown in Figure 2.11, under the following assumptions. When processing a connected component, first *Find_MST_Edges* determines all pairs of edge-cells for which *Approximate_NFN* will be called to find an NFN pair. This set of edge-cell pairs is processed by *Approximate_NFN* "in parallel," so that the algorithm proceeds in phases of refinement steps. At each refinement step, upper and lower bounds, denoted $NFN_{\max}(A, B)$ and $NFN_{\min}(A, B)$, will be known for the NFN pair distance between edge-cells A and B for which a NFN pair is being found. The difference between these bounds will be twice the diameter of the cells being examined at that step in the "parallel" NFN algorithm. At each refinement step, it will be possible to eliminate some edge-cell pairs from consideration in the NFN algorithm because an edge between them cannot be in an MST. Such pairs will be termed *inactive*. Associated with an inactive pair $\{A, B\}$ is a value $pathmax(A, B)$, indicating that there are paths from every site in A to every site in B, all with edges shorter than $pathmax(A, B)$. This value will also be defined for active edge-cell pairs $\{A, B\}$, taking $pathmax(A, B)$ as $NFN_{\max}(A, B)$.

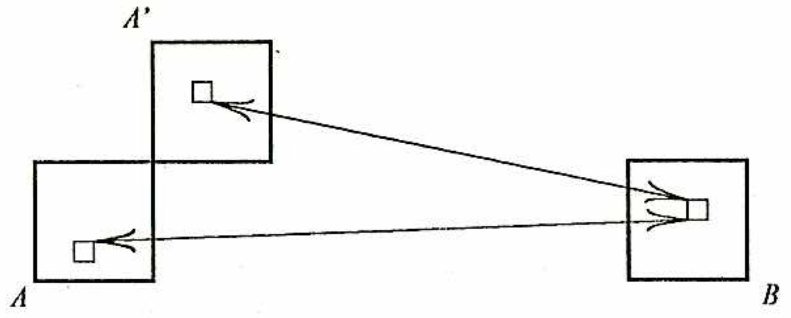

Figure 2.12. Some edge-cell pairs can be pruned away.

We will first show that the pruning step preserves correctness, and then show that the condition wanted for the time bound holds.

> **Lemma 2.9.** If *Find_MST_Edges* is modified to incorporate the pruning step of Figure 2.11, then the set of edges returned will still contain an MST.

Proof. Suppose, at some point, that $NFN_{\min}(A, B) > pathmax(A', B)$, for edge-cell pairs $\{A, B\}$ and $\{A', B\}$, and in addition, $d_{\max}(A, A') \leq 2Diameter(A)$. (As in Figure 2.12.) Since the edge-cells have a diameter less than half the side-length of the component cell size, it follows that $pathmax(A', B) \geq d_{max}(A, A')$. As a result, there is a path by way of A' between sites in A and in B with edges no longer than $pathmax(A', B)$. This path forms a cycle with any edge between A and B, and since

$$NFN_{\min}(A, B) > pathmax(A', B),$$

that edge is the longest on the cycle. By Fact 1.3, such an edge is not in any MST. Therefore the pair $\{A, B\}$ may be made inactive, and the set of edges returned will still contain an MST. ∎

As suggested above, the following lemma allows an improved time bound for finding an MST.

> **Lemma 2.10.** If procedure *Find_MST_Edges* is modified to incorporate the pruning step, then a quadtree cell a will be explicitly considered in an NFN pair computation $O(1)$ times.

Proof. The lemma follows from the fact that, because of the modification to procedure *Approximate_NFN* mentioned, and the addition of this pruning step, there is a conical region near a cell examined in an NFN call that is free of sites. At the end of a pruning step, if an edge-cell pair $\{A, B\}$ is not made inactive, then for any edge-cell pair $\{A', B\}$ with $d_{\max}(A', A) \leq 2Diameter(A)$, it will be true that

$$NFN_{\min}(A, B) \leq pathmax(A', B).$$

This means that if $\{A', B\}$ is active, then

$$NFN_{\min}(A,B) \leq NFN_{\max}(A',B) \leq NFN_{\min}(A',B) + \alpha,$$

where α is the current accuracy to which the NFN is known. This accuracy is proportional to *Diameter*(a). If $\{A', B\}$ is inactive, then $NFN_{\min}(A,B)$ is no larger than the last computed value of $NFN_{\min}(A',B)$. Therefore, in either case, no site in A' is closer than $NFN_{\min}(A,B) - \alpha$ to any site in B. A cell a in A is considered at a step only if there is a cell b in B with

$$d_{\min}(a,b) \leq NFN_{\max}(A,B) \leq NFN_{\min}(A,B) + \alpha.$$

Therefore, there are no sites in the intersection of the interiors of balls $\mathcal{B}_a$ and $\mathcal{B}_b$, defined by:

$\mathcal{B}_a$ is a ball centered at the center of cell a, with radius at least the edge-cell diameter, and

$\mathcal{B}_b$ is a ball centered at the center of cell b, with radius no less than the distance between a and b, less twice *Diameter*(a).

(See Figure 2.13.) Now, because the bounded ratio of the edge-cell size to the size of the component cells currently being processed, it follows that only a bounded number of *Approximate_NFN* calls in which a is touched will occur before all such calls are due to sites in $\mathcal{B}_a$. Furthermore, only a bounded number of NFN calls will occur that are due to sites that are farther from a than 2*Diameter*(a). This follows from the fact that the intersection of a narrow cone $\mathcal{C}$ with $\mathcal{B}_a$ will contain no sites that are in $\mathcal{B}_b$: The radius of $\mathcal{B}_a$ is less than that of $\mathcal{B}_b$, and so by the definition of narrow, any site in $\mathcal{C} \cap \mathcal{B}_a$ is also in $\mathcal{B}_b$. Finally, only a bounded number of NFN calls involving a can come from cells closer to a than 2*Diameter*(a), so that there are only a constant total number of times that a quadtree cell a will be touched in an NFN call over the course of the MST algorithm. ∎

Using the previous three lemmas, the following theorem results.

Theorem 2.11. If the nearest foreign neighbor problem can be solved for site sets S and T in time $O(f(|S|+|T|, \alpha))$ with absolute error α, then the geometric minimum spanning tree problem can be solved with relative error α in $O(f(n,\alpha)\log\sigma)$.

When algorithm *NFN* in particular is used, the previous lemmas and the time bound for *NFN* imply the following theorem.

Theorem 2.12. Procedure *Find_MST_Edges*, employing algorithm *NFN* and modified to include the pruning step, requires $O(n\log^{d-1}(1/\rho)\log\sigma)$ time to find a set of edges whose MST has total weight within $1+\rho$ of the MST weight.

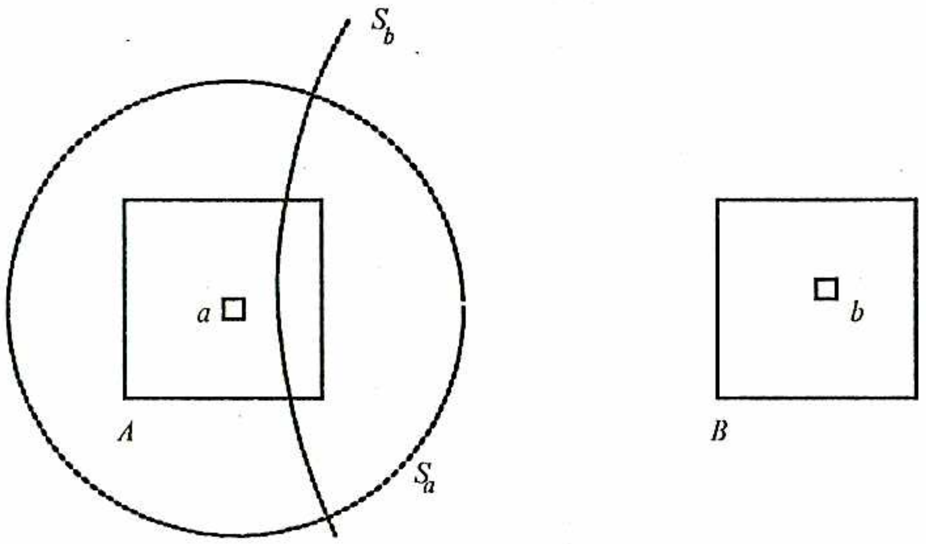

Figure 2.13. The lune from a is free of sites of P_i.

2.3.3 The 3D Euclidean case

As an example of Theorem 2.11, for the three-dimensional Euclidean case the MST problem can be solved in $O(n(\log n + 1/\alpha)\log\sigma)$ time by using an approximation algorithm for the NFN problem based on two-dimensional Voronoi diagrams. In such an algorithm, a quadtree cell is split into slabs α thick. In each slab, the sites are projected to a square. To solve the post office problem approximately for the sites in the slab, project a 3-d query site onto the plane containing the square for the sites, and solve the post office problem for the projected site. (See Figure 2.14.) Thus the post office problem for n sites in three dimensions can be solved with $O(n\log n)$ preprocessing and $O(1/\alpha + \log n)$ query time. Therefore, the NFN problem can be solved in time bounded by $O((|S| + |T|)(\log n + 1/\alpha))$, yielding a corresponding MST algorithm. When $(\log\sigma)/\alpha$ is a function of n that is $o(n^{4/5})$, this algorithm compares favorably in running time with Yao's [Y1], which has a running time of $O((n\log n)^{9/5})$.

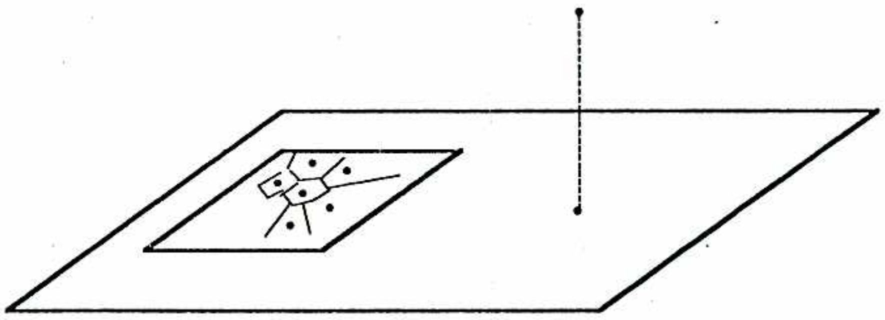

Figure 2.14. Solving the 3-d post office problem by projection.

Chapter 3.
A Probabilistic Algorithm

In this chapter, a probabilistic algorithm for solving the all nearest neighbors problem will be described. Probabilistic algorithms employ randomization, such as random sampling, so that their running time for a fixed input is a random variable. Their performance is usually measured by the expected time required, averaging over that randomization. In an influential paper by Rabin [Tra], it is shown that the closest pair of sites in a set can be found in linear expected time, using randomization. In many cases, there are deterministic algorithms with worst-case performance nearly matching the expected performance of probabilistic algorithms. However, probabilistic algorithms are often notable for their simplicity and practicality.

The algorithm presented here has two main phases: the construction of an auxiliary data structure, called a *celltree*, in probabilistic $O(n \log n)$ time, and the use of that data structure to solve the ANN in $O(n)$ worst-case time. Before these phases are described in §3.1 and §3.2, the motivation for the use of celltrees will be indicated, and also their basic structure.

While algorithm ANN_a, described in §2.1, has several appealing features, the dependence of its running time on the numerical parameter σ is perhaps inelegant and potentially could result in slow performance. If the sites are evenly distributed, then σ will be about $n^{-1/d}$, and ANN_a will require $O(n \log n)$ time. Unfortunately, such a smooth spread of sites is not necessarily present, as a cell may have a concentration of a large number of sites in a small volume, so that a cell may have only one descendent for some arbitrarily large number of generations. That is, the quadtree explored by the algorithm may have arbitrarily long paths without any branching. To use the basic approach of ANN_a, but avoid this problem, a celltree will be used. A celltree is simply a quadtree, but with the non-branching paths replaced by single edges. A simple quadtree, and the corresponding celltree, are shown in Figure 3.1. The leaf nodes of the celltree, with no celltree children, are singleton cells. These leaf cells are the only singleton cells in the celltree. Note that celltrees are related to quadtrees in basically the same way that "Patricia" search tries are to digital search tries ([Knu], §6.3).

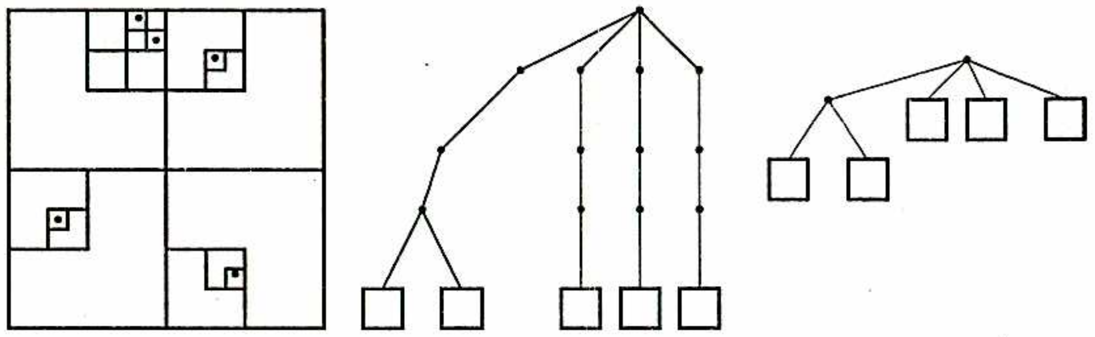

Figure 3.1. Example quadtree and celltree.

§3.1 Building a Celltree

A celltree can be built using the divide-and-conquer algorithm *Build_Celltree*, shown in Figure 3.2. At each call of this algorithm, the input is a quadtree cell T and a set of objects (sites and celltrees) within T. Initially, the set of objects is just the set of sites, but as the algorithm proceeds, some parts of the celltree are built recursively. These parts, which are celltrees themselves, then form part of the input to the algorithm.

In the dividing step of *Build_Celltree*, a quadtree cell called a *splitting cell* is found. A splitting cell satisfies two conditions:

▸It is the smallest quadtree cell containing the objects within it, and also
▸it contains between ϵ and $1-\epsilon$ of the objects in T, where $\epsilon = 1/(2^d+1)$.

Having found a splitting cell C, a celltree rooted at C is built, and the celltree for T is built with the objects in C represented by C alone. This approach is similar to that of the sorting procedure Quicksort ([Hoa],[Knu], p. 114). In Quicksort, values are partitioned into those smaller and those larger than some partitioning value. In this case, the objects are partitioned into those inside and those outside the splitting cell.

The first condition for a splitting cell ensures that the result of the algorithm is actually a celltree, and has no nonbranching paths. The second condition guarantees that each of the two subproblems solved is relatively small. As in Quicksort, if the total time for the dividing step is $O(n)$, for input size n, then the total time required is $O(n \log n)$. The function *Find_Splitting_Cell*, shown in Figure 3.2 and described below, requires $\Theta(k^2)$ time in the worst case to find a splitting cell for k objects. However, when this function is applied to a random sample of the objects, with a nonzero probability the result is a splitting cell for the whole set of objects. This technique is analogous to the use in Quicksort of the median of three randomly chosen values as the partitioning value. The result is that *Build_Celltree* requires $O(n \log n)$ time on the average, for any input set of sites, as shown below.

```
function Build_Celltree(T : cell, Objects : set_of_objects)
  return celltree_root;
co It should be true that $T = Smallest\_Cell(Objects)$ oc;
co Function Smallest_Cell is described in the text. oc;
begin
  if $|Objects| \leq K$
    then return Build_Celltree_Deterministically(T, Objects)
    else
      choose random $S \subset Objects$;
      co should have $\epsilon |S| > 1$ to avoid degeneracy oc
      $C \leftarrow Find\_Splitting\_Cell(T, S)$;
      $Objects\_In\_C \leftarrow \{x \mid x \in Objects,$ x geometrically inside $C\}$;
      $T' \leftarrow Build\_Celltree(C, Objects\_In\_C)$;
      return $Build\_Celltree(T, Objects \cup T' \setminus Objects\_In\_C)$;
  fi;
end;


function Find_Splitting_Cell(T : cell, S : set_of_objects)
  return celltree_root;
  co It should be true that the number of objects in T is greater than $(1 - \epsilon)|S|$ oc;
begin
  for $D \in Children(T)$ do
    $Objects\_In\_D \leftarrow \{x \mid x \in S,$ x geometrically inside $D\}$;
    $Proportion\_In\_D \leftarrow |Objects\_In\_D| \,/\, |S|$;
    $C \leftarrow Smallest\_Cell(Objects\_In\_D)$;
    if $Proportion\_In\_D > 1 - \epsilon$ then return $Find\_Splitting\_Cell(C, S)$ fi;
    if $\epsilon \leq Proportion\_In\_D \leq 1 - \epsilon$ then return C fi;
  od;
end;
```

Figure 3.2. Function Build_Celltree.

The size $s = |S|$ of the sample used for *Find_Splitting_Cell* must be larger than $1/\epsilon$, to avoid the degenerate case where a cell containing one site might be a splitting cell. (In this degenerate case, the splitting cell cannot satisfy the second condition above without being an "infinitely small" quadtree cell.) Furthermore, there is a tradeoff between the time needed by *Find_Splitting_Cell* for a larger sample, and the improvement in running time for *Build_Celltree* due to the higher probability of a splitting cell being found. Also, for small input to *Build_Celltree*, it may be better to build a celltree by "brute force." Thus for a small constant $K > |S|$, a celltree is built deterministically.

Before giving a proof of the time bound for *Build_Celltree*, we will consider the processing within *Find_Splitting_Cell* in more detail. The heart of this function is the function *Smallest_Cell*, that answers the following query:

Given a set of sites, what is the smallest quadtree cell containing those sites?

To process such a query, *Smallest_Cell* requires the use of the floor, logarithm, and bitwise exclusive-or functions, and is the only algorithm in this chapter that does so. The purpose of *Smallest_Cell* is to help in skipping over the nonbranching paths in a quadtree.

The function operates in the following way. Assume without loss of generality that the root cell of the a quadtree has side length 1. Observe that every quadtree cell of side length 2^{-k} has the form $\bigotimes_{1\le j\le d}[i_j 2^{-k}, (i_j+1)2^{-k})$, where each i_j satisfies $0 \le i_j < 2^k$. Let point $p^{\min}$ have jth 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 $i_j 2^{-k} \le p_j^{\min} \le p_j^{\max} < (i_j+1)2^{-k}$. Thus if function $f(x,y)$ computes such a value k given $p^{\min}$ and $p^{\max}$, the subdivision level b of the cell to be found is equal to $\min_{1\le j\le d} f(p_j^{\min}, p_j^{\max})$. Given this value b, the cell desired has $i_j = \lfloor p_j^{\min} 2^b \rfloor$.

Another way to describe $f(x,y)$ is that it is the largest k for which the integer parts of $2^k x$ and $2^k y$ are equal. This is the number of common leading bits of x and y. In the exclusive-or $x \oplus y$, these leading bits result in zeros, and the first pair of different bits results in a one. The number of leading zeros in $x \oplus y$ may be recovered as $-\lfloor \lg x \oplus y \rfloor - 1$, for $0 \le x, y < 1$. The function $f(x,y)$ might be termed the *least common ancestor* of x and y: If the binary representation of a number corresponds to a node in a binary tree, then the function $f(x,y)$ corresponds to the level of the least common ancestor in that tree of the nodes corresponding to x and y.

Given *Smallest_Cell*, the function *Find_Splitting_Cell* works as follows. In the general recursive step, the input cell has at least $(1-\epsilon)$ of the objects in S. This implies that one of its children D must have at least ϵs objects, since there are only $2^d = 1/\epsilon - 1$ children. Either D contains no more than $(1-\epsilon)s$ objects, and *Smallest_Cell*(D) is a splitting cell, or the algorithm may be applied recursively to *Smallest_Cell*(D). By applying *Smallest_Cell* to the contents of D before the recursive call, the invariant is maintained that every cell input to *Find_Splitting_Cell* has at least two children, each containing at least one object from S. Therefore the number of objects from S in the cell examined decreases by at least one at each recursive call. Since the execution of *Find_Splitting_Cell* requires $O(s)$ time, except for the recursive call, we have the following lemma.

> **Lemma 3.1.** When applied to a set of s objects, function *Find_Splitting_Cell* requires time proportional to s^2 (hence $O(1)$) to find a splitting cell, where the constant is dependent on the dimension.

For example, in the square in Figure 3.3, with a total of 15 sites, a square is sought with more than 3 and less than 12 sites. No child of the root square will do, so the heavy upper right square with 13 sites is examined. Within this square, the lower left square will do, so we take as C' the smallest quadtree square containing all of the sites within that lower left square. At the second recursive call, with 7 objects in the square (6 sites 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.

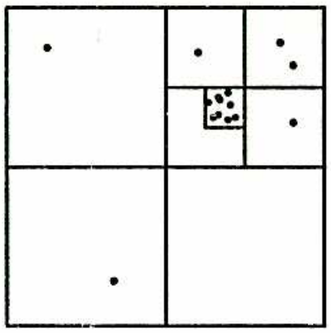

Figure 3.3. Finding a splitting cell.

Now for a proof of the time bound for *Build_Celltree*:

Theorem 3.2. Procedure *Build_Celltree* requires $O(n \log n)$ expected time to build a celltree for n sites.

Proof. As for the correctness of the procedure, note that if the input cell T satisfies the condition that $T = \mathit{Smallest\_Cell}(\mathit{Objects})$, then this condition will always be true for the inputs to *Build_Celltree* and *Find_Splitting_Cell*. Therefore, assuming the correctness of *Build_Celltree_Deterministically*, the tree returned will be a celltree.

As for the time bound, note first that all processing requires linear time, except for the recursive calls to *Build_Celltree*. Let T_n denote the expected time required by *Build_Celltree* to find a celltree for n objects. If α_k denotes the probability that the cell found by *Find_Splitting_Cell* has k objects, then

$$T_n \le C_1 n + \sum_{1<k<n} \alpha_k [T_k + T_{n-k+1}],$$

where $C_1 n$, for some $C_1 > 0$, bounds the time required, except for the recursive call. We know that $\alpha_1 = \alpha_n = 0$ because the splitting cell returned will always have at least two sample objects in it, and will not contain all of the sample objects. Note that the α_k probabilities will be different for different sets of n objects. However, let m denote some value greater than 1 that will be chosen later. Let $\hat{\phi}_m$ denote the smallest possible value of ϕ_m, which will defined as

$$\sum_{k_m \le k \le n-k_m} \alpha_k$$

for a given set of n objects, where $k_m = \lfloor \epsilon n/m \rfloor$. A lower bound for $\hat{\phi}_m$ that is independent of n will be proven shortly. This will provide a time bound for *Build_Celltree*, by the following lemma.

Lemma 3.3. For sufficiently large n, $T_n \le C_2 n \ln n$, where

$$C_2 = 16 \frac{C_1}{\hat{\phi}_m} \frac{m}{\epsilon}.$$