Spatial Tessellations: Concepts and Applications of Voronoi Diagrams
Edition by ATSUYUKI OKABE & BARRY BOOTS and SUNG NOK CHIU

CHAPTER 4

Algorithms for Computing Voronoi Diagrams

In this chapter we consider practical methods for constructing Voronoi diagrams. Here, we mainly concentrate our attention on the most basic case, namely the Voronoi diagram for points in a plane. In Section 4.1 we present a naive method which follows directly from the definition of the Voronoi diagram. The naive method is however inefficient, and many efforts have been made in the field of computational geometry to make efficient algorithms, which are also surveyed in this section. In Section 4.2 we present a typical data structure for representing the Voronoi diagram and thus make clear what is meant by the 'construction' of the Voronoi diagram. We also derive a lower bound of the time complexity for the construction of the Voronoi diagram. Then we present three typical algorithms, respectively, in Sections 4.3, 4.4 and 4.5. These algorithms are designed on the assumption that no numerical error takes place in the course of computation, so that a direct translation of such an algorithm to a computer program does not necessarily give a numerically valid program; such a program may fail because of geometric inconsistency caused by numerical errors. In order to make a practically valid computer program, we have to consider numerical errors. Hence, in Section 4.6 we present two typical techniques for avoiding inconsistency due to numerical errors. The last three sections, Sections 4.7, 4.8 and 4.9, are concerned with generalized Voronoi diagrams. In Section 4.7 we present a typical algorithm for constructing higher dimensional Voronoi diagrams, while in Section 4.8 we briefly touch upon algorithms for various kinds of generalized Voronoi diagrams in the plane. Finally in Section 4.9 we present approximation algorithms for generalized Voronoi diagrams.

4.1 COMPUTATIONAL PRELIMINARIES

As in the previous chapter, let $P = \{p_1, \ldots, p_n\}$ be the set of generators, and $\mathcal{V} = \{V(p_1), \ldots, V(p_n)\}$ denote the Voronoi diagram for P. The construction

of the Voronoi diagram is a procedure for generating $\mathcal{V}$ from P. In the previous chapter we saw several equivalent conditions for characterizing the Voronoi diagram. Among them the characterization based on the half planes (Definition V3) directly gives us a naive method for constructing the diagram. The Voronoi polygon for the generator p_i is the intersection of all the half planes defined by the perpendicular bisectors of p_i and the other generators. According to this definition, we can construct the Voronoi polygons one by one. Thus we get the following method.

Naive method

Input: n generators $p_1, p_2, \ldots, p_n$.
Output: Voronoi diagram $\mathcal{V} = \{V(p_1), V(p_2), \ldots, V(p_n)\}$.
Procedure :
Step 1. For each i such that $i = 1, 2, \ldots, n$, generate $n-1$ half planes $H(p_i, p_j)$, $1 \leq j \leq n, j \neq i$, and construct their common intersection $V(p_i)$.
Step 2. Report $\{V(p_1), V(p_2), \ldots, V(p_n)\}$ as the output and stop.

In this chapter we present a number of geometric algorithms and name them with sequential numbers. However, we do not want to call the above method a practical 'algorithm', because it is 'insufficient'. Let us see how 'insufficient' the above method is. To evaluate an algorithm, we have to consider at least correctness and efficiency, and if it involves numerical computation, we also have to consider robustness against numerical errors. The above method is correct, because it is a restatement of the definition of the Voronoi diagram. So, we consider the other two aspects.

First we consider efficiency. To construct the half plane $H(p_i, p_j)$ for two given points p_i and p_j requires only constant time. Hence for each p_i the time required for constructing $n-1$ half planes is proportional to $n-1$, say $a(n-1)$, where a is a positive constant. To construct the intersection of the half planes, let us consider the following simple procedure. First we construct the intersection of two half planes, obtaining a polygon with two sides; next construct the intersection of this polygon with the third half plane, and so on. In the kth step of this procedure, we have to find the intersection of a k-sided polygon (in the worst case) with another half plane, and this step requires time at least proportional to k if we check whether each of the k sides crosses the boundary line of the half plane. Thus, constructing the intersection of $n-1$ half planes requires time proportional to $1 + 2 + \ldots + (n-2) = (n-2)(n-1)/2$, say $b(n-2)(n-1)$, where b is another positive constant. We have to repeat this process for all generators, and consequently the total time required by the naive method is

$$T(n) = n\{a(n-1) + b(n-2)(n-1)\} = O(n^3).$$

This means that for a computer program based on the above procedure, the time for processing becomes eight ($= 2^3$) times larger as the size of the input data becomes twice as large.

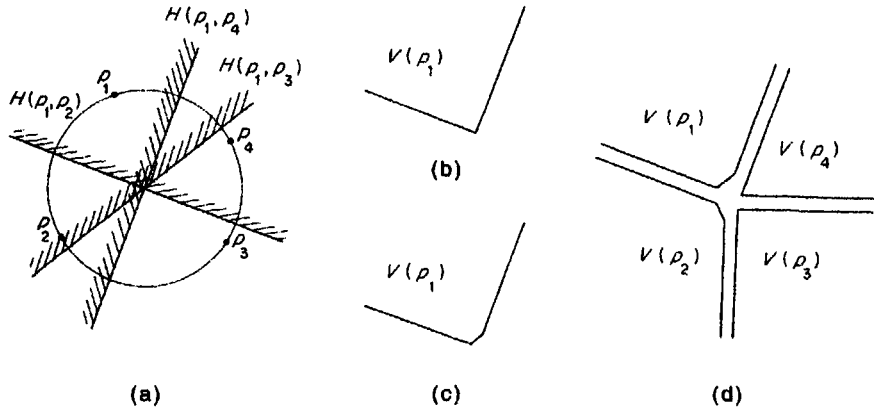

Figure 4.1.1 Inconsistency caused by numerical errors: (a) four generators on a common circle; (b) possible Voronoi polygon for p_1; (c) another possible Voronoi polygon for p_1; (d) inconsistent set of Voronoi polygons.

Employing a little more sophisticated technique, we can construct the intersection of $n-1$ half planes in $O(n \log n)$ time (Preparata and Shamos, 1985). Hence the time complexity of the naive method can be decreased to $O(n^2 \log n)$.

As we will see in the next section, any algorithm for constructing the Voronoi diagram requires at least $O(n \log n)$ time in the worst case, and at least $O(n)$ time on average. Moreover, there exist algorithms whose time complexities attain these lower bounds, and some of them actually run very fast even for small values of n (although the order itself represents the behaviour of the algorithm as n approaches infinity). The algorithm presented in Section 4.3 requires $O(n)$ time on average and $O(n^2)$ time in the worst case, while the algorithms presented in Sections 4.4 and 4.5 require $O(n \log n)$ time in the worst case. Compared with these algorithms, the naive algorithm is not very satisfactory from the time-complexity point of view.

Another important point of view for evaluating the algorithm is robustness against numerical errors. From this point of view too the naive method is not satisfactory. This can be understood by the following example. Suppose that there are only four generators p_1, p_2, p_3 and p_4 and they come very near to a common circle. Then, as shown in Figure 4.1.1(a), the three perpendicular bisectors between p_1 and the other generators pass near to the centre of the circle; if numerical error takes place, it is difficult to decide the relative locations of the points of intersections of these bisectors. Hence the result of intersecting the three half planes associated with p_1 may be a two-sided polygon as in (b) or a three-sided polygon as in (c); which result comes out depends on numerical errors. The other three Voronoi polygons have similar instability. In the naive method, each Voronoi polygon is computed independently, so that the result may become topologically inconsistent, as shown in Figure 4.1.1(d). This figure is contradictory, because the Voronoi

polygon $V(p_1)$ shows that there is a Voronoi edge between p_1 and p_3, but the Voronoi polygon $V(p_3)$ shows that such a Voronoi edge does not exist; we cannot glue the boundaries of the Voronoi polygons together.

From a theoretical point of view all the Voronoi polygons give a tessellation of the plane, but in actual computation the output of the naive method does not necessarily give a tessellation of the plane. Thus the naive method, if translated to a computer program in a straightforward manner, is not robust against numerical errors. Its output may be topologically inconsistent. We need careful consideration in order to make a numerically robust algorithm. Basic techniques for this purpose will be presented in Section 4.6.

Here, we will briefly survey typical algorithms for constructing the ordinary Voronoi diagram in the plane. First of all, we should note that some algorithms are described in terms of the Voronoi diagram itself (e.g. Shamos and Hoey, 1975; Sibson, 1980a; Ohya *et al.*, 1984a), while others are described in terms of the Delaunay diagram (e.g. Lee and Schachter, 1980; Guibas and Stolfi, 1985). As we saw in Section 2.2, however, these two diagrams are duals of each other, and we can transform one to the other and vice versa. Moreover, the description of the algorithm itself can be transformed from Voronoi diagram terminology to Delaunay diagram terminology, and vice versa. Hence the choice of the terminology does not make any essential difference.

The naive method presented above is one of the easiest methods to understand (Rhynsburger, 1973) because it is a direct translation of the definition of the Voronoi diagram. Although this method is not recommended for general purposes, it may be used for some special types of simulation where the goal is not to construct the Voronoi diagram itself, but to generate many samples of Voronoi polygons to obtain statistical data such as the distribution of the number of edges per polygon (Crain, 1978; Boots and Murdoch, 1983; Quine and Watson, 1984). Bentley *et al.* (1980) combined this method with a bucketing technique and presented an algorithm that runs in $O(n)$ time on average, in which generators relevant to each Voronoi polygon are searched for from bucket to bucket in a spiral order (the bucketing technique will be described in Section 4.3). However, the output of the naive method is a simple collection of Voronoi polygons, and it does not include explicit information about the topological structure of the diagram. As we will see in more detail in the next section (Section 4.2), if we want to extract various kinds of information from the Voronoi diagram, the naive method is not suitable.

The second method, which is another naive method, can be categorized as a 'walking method', in which Voronoi vertices and Voronoi edges (or, equivalently, Delaunay polygons and Delaunay edges) are constructed one by one just in the order in which a traveller walks along the edges of the diagram. This method was described in Delaunay diagram terminology by Lawson (1977), and in Voronoi diagram terminology by Brassel and Reif (1979) and Cromley and Grogan (1985). While they did not consider the time complexity point of view, Maus (1984) utilized the bucketing technique to construct an

$O(n)$ algorithm in the average sense, in which the algorithm is described in terms of the Delaunay triangulation.

The third method may be called a 'flip method', in which an initial diagram is constructed first, and then it is modified step by step until it converges to the Voronoi diagram. Sibson (1978) showed that, starting with any triangulation of $CH(P)$, we can modify it to the Delaunay triangulation by flipping the diagonals of convex quadrilaterals according to the local max-min angle criterion, i.e. the diagonal of a convex quadrilateral is replaced by the other diagonal if the resulting pair of triangles gives a larger value for the minimum angle. As we saw in Section 2.4, Edelsbrunner (1988) gave an intuitive and simple interpretation of the procedure in the context of the lift-up transformation, and showed that $O(n^2)$ flipping of the diagonals is enough to obtain the Delaunay triangulation from any initial triangulation.

The fourth method, a simple but still very powerful method, is an incremental method, in which we start with a simple Voronoi diagram for two or three generators, and modify it by adding generators one by one; the method was presented in terms of the Voronoi diagram by Green and Sibson (1978), Sibson (1980a,b), Lee and Schachter (1980), Shapiro (1981), Tipper (1990a), and in terms of the Delaunay diagram by Watson (1981), Devijver and Dekesel (1982), Gowda *et al.* (1983), Correc and Chapuis (1987), Palacios-Velez and Renaud (1990), Jünger *et al.* (1991) and Tsai (1993). In the worst case, each addition of a generator requires time proportional to the number of generators added so far, and consequently the total time complexity is of $O(n^2)$. However, the average time complexity can be decreased to $O(n)$ by the use of special data structures such as buckets and a quaternary tree (Ohya, 1983; Iri *et al.*, 1984; Ohya *et al.*, 1984a; Maus, 1984). Randomization techniques are also useful to decrease the average time complexity (de Berg *et al.*, 1995; Guibas *et al.*, 1992). Also a numerically robust version of this method was proposed by Sugihara and Iri (1988, 1989b,d). Thus, this method is one of most practical from both the time-complexity and the robustness points of view. This method will be presented in detail in Section 4.3 and the numerically robust version will be shown in Section 4.6.

The fifth method, another typical method for constructing the Voronoi diagram, is the use of the divide-and-conquer paradigm, in which the set of generators is recursively divided into smaller subsets and the Voronoi diagrams for those subsets of generators are merged into the final diagram. This method was first proposed in terms of the Voronoi diagram by Shamos and Hoey (1975); an early attempt to implement this method was made, for example, by Horspool (1979). Later this method was written in a simpler form using Delaunay triangulation terminology by Drysdale and Lee (1978), Lee and Schachter (1980), Guibas and Stolfi (1985) and Elbaz and Spehner (1990). The worst-case time complexity of this method is $O(n \log n)$, which is the best possible as we will see in the next section. However, as was shown by Ohya *et al.* (1984a), the original divide-and-conquer method requires $O(n \log n)$ time also in the average sense. The average-case time complexity was lessened to $O(n \log \log n)$ by Dwyer (1987), and further to $O(n)$ by

Katajainen and Koppinen (1988). A robust version of the method was also constructed (Ooishi, 1990; Sugihara *et al.*, 1990; Ooishi and Sugihara, 1995). The divide-and-conquer method will be shown in Section 4.4.

The sixth method is the plane sweep method (Fortune, 1986, 1987). The plane sweep is one of fundamental techniques in computational geometry, by which a two-dimensional problem can be reduced to an almost one-dimensional problem. A vertical line, called a sweep line, is moved over the plane from left to right, and the Voronoi diagram is constructed along this line. This method also attains the worst-case optimal time complexity, $O(n \log n)$. This method will be presented in Section 4.5.

The seventh method may be called a 'lift-up method', which utilizes the relationship between a two-dimensional Voronoi diagram and a three-dimensional convex hull (Properties D7 and D8). In this method, first the generators in the plane are transformed to certain points in three-dimensional space, then their convex hull is generated, and finally the convex hull is inversely transformed to the original plane to obtain the Delaunay diagram. Brown (1979, 1980), Aurenhammer and Edelsbrunner (1984) and Buckley (1988) used a transformation from the plane to a sphere (Property D7), and Edelsbrunner and Seidel (1986) and O'Rourke *et al.* (1986) used a transformation from the plane to a paraboloid of revolution (Property D8). This method can be extended to any dimension, and hence we will see this method in the context of the construction of higher-dimensional Voronoi diagrams (Section 4.7).

Other directions of algorithmic research include the use of a matrix representation (Fang and Piegl, 1992), construction of the Voronoi diagram for convexly located generators (Agarwal *et al.*, 1989a), digital-image approximation of the Voronoi diagram (Toriwaki *et al.*, 1982; Mark, 1987; Yoshitake *et al.*, 1987), parallel computation of the Voronoi diagram (Lu, 1986; Saxena *et al.*, 1990; Adamatzky, 1993; Cole *et al.*, 1990, 1996; Evans and Stojmenovic, 1989; Garga and Bose, 1994; Goodrich *et al.*, 1993; Blelloch *et al.*, 1996), and constructing the dynamic Voronoi diagram (Tokuyama, 1988; Bajaj and Bouma, 1990; Fu and Lee, 1991; Huttenlocher *et al.*, 1992a; Devillers *et al.*, 1992; Roos, 1993). An attempt to decrease storage requirements was also studied by Kartashov and Folk (1995).

For a while (i.e. from Section 4.2 to Section 4.5) we make the following assumptions.

Assumption A4.1.1 Numerical computation is carried out in precise arithmetic.

Assumption A4.1.2 No four generators align on a common circle.

The first assumption is necessary to consider algorithms in a 'theoretically closed world', and has been adopted, implicitly or explicitly, in almost all discussions for designing geometric algorithms. The second assumption is to avoid degeneracy. Note that this assumption is not equivalent to the

non-cocircularity assumption; the non-cocircularity assumption avoids an empty circle passing through four points, whereas Assumption A4.1.2 avoids any circle passing through four points; Assumption A4.1.2 is stronger than the non-cocircularity assumption. This assumption is necessary to avoid degeneracy in the course of constructing the Voronoi diagram, because we treat many Voronoi diagrams generated by subsets of the generator set. Whether a case is special or not sometimes depends on which algorithm we consider. Indeed, we will make other assumptions when we discuss other algorithms.

In Section 4.6, on the other hand, we remove all such assumptions and construct algorithms that are valid in the real world.

4.2 DATA STRUCTURE FOR REPRESENTING A VORONOI DIAGRAM

Before starting our consideration of algorithms, we have to make clear what is meant by 'construction of a Voronoi diagram'. Naively it might be thought that constructing a Voronoi diagram is equivalent to drawing it on a sheet of paper. Indeed, to draw the Voronoi diagram is a typical way to present it to the human eye. However, the drawn diagram itself, or a set of numerical data equivalent to the drawn diagram, does not convey enough information for many applications. Suppose that a computer program gives as output a list of line segments corresponding to the Voronoi edges. It is easy to draw the diagram from this list, but it is not easy to retrieve, for example, a sequence of Voronoi vertices on the boundary of one Voronoi polygon, or a set of Voronoi polygons contiguous to one Voronoi polygon. Thus, to 'draw' the Voronoi diagram is not enough in many applications. What we need to construct is a 'richer structure' from which we can extract a variety of information necessary for applications.

Another extremal way of representing the Voronoi diagram is to give all necessary data explicitly. For example, for each Voronoi polygon we may have the list of all Voronoi vertices on its boundary, the list of all Voronoi edges on its boundary, the list of all contiguous Voronoi polygons, and so on. However, this way of representation requires much space. Hence we have to choose some data structure in between which is concise but is still rich enough for us to retrieve various aspects of the diagram.

Here we choose one standard data structure, called a winged-edge data structure or a polygon data structure, which is common in geometric modelling and computer graphics (Baumgart, 1975; Ballard and Brown, 1982; Hoffmann, 1989; see also Guibas and Stolfi, 1985, for an alternative data structure). As will be shown in what follows, this data structure conveys explicit information about local incidence relations among Voronoi vertices, Voronoi edges and Voronoi polygons, so that a variety of information can be retrieved relatively easily.

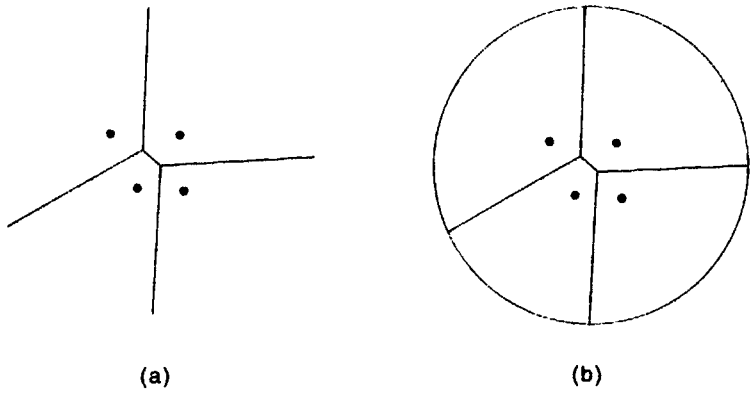

Figure 4.2.1 Voronoi diagram and the associated geometric graph: (a) Voronoi diagram; (b) geometric graph associated with the Voronoi diagram in (a).

The winged-edge data structure is a way of representing a geometric graph in the plane. However, the Voronoi diagram is slightly different from a geometric graph in that some Voronoi edges extend infinitely. To consider the Voronoi diagram as a geometric graph, we introduce a closed curve which is large enough to surround all the Voronoi vertices and consider that the infinite Voronoi edges have their terminal points on this closed curve. These terminal points are intuitively considered as points at infinity. By this convention, the Voronoi diagram shown in Figure 4.2.1(a) is converted to the diagram in (b). At these terminal points the closed curve itself is divided into curved line segments; these line segments are also considered as Voronoi edges. Thus, by this convention the Voronoi diagram is converted into a geometric graph. We call this geometric graph *the augmented geometric graph* associated with the Voronoi diagram. (Note that the augmented geometric graph is a little different from the geometric graph of the Voronoi diagram introduced in Section 2.3; in the geometric graph all the infinite Voronoi edges are connected to a common additional vertex, whereas in the augmented geometric graph infinite Voronoi edges are connected to mutually different vertices 'at infinity'. We adopt this structure because we want to store the direction of the infinite edges as the 'coordinates' of these vertices, as we will see soon.)

Consider the Voronoi diagram for the set $P = \{p_1, \ldots, p_n\}$ of n generators. By the above convention the plane is divided into $n+1$ regions, n finite regions corresponding to the n Voronoi polygons and one outermost infinite region. We introduce a virtual generator, say p_∞, and consider that the outermost infinite region is the Voronoi region of p_∞. Thus the augmented geometric graph associated with the Voronoi diagram for P has $n+1$ regions, corresponding to $p_1, p_2, \ldots, p_n$ and p_∞. Let n_e and n_v be the number of edges and vertices, respectively, of the augmented geometric graph. Since at least

three edges meet at a vertex, we get $3n_v \leq 2n_e$. From this inequality together with Euler's formula $(n + 1) - n_e + n_v = 2$, we get

$$n_e \leq 3n - 3 \quad \text{and} \quad n_v \leq 2n - 2, \tag{4.2.1}$$

where the equality holds when no degeneracy takes place. (Note that these inequalities are a little different from those in Property V11 in Section 2.3; this is because the 'pseudo' Voronoi edges and the 'pseudo' Voronoi vertices on the outermost closed curve are also counted here.)

To represent the augmented geometric graph associated with the Voronoi diagram, we create the winged-edge data structure in the following way. First, for each edge we choose and fix the direction of the edge arbitrarily, and thus convert the geometric graph to a directed geometric graph. Next we arbitrarily choose and fix a linear order from 1 to n_v to vertices, and similarly choose and fix a linear order from 1 to n_e to edges. Furthermore, we call the Voronoi polygon associated with p_i, *polygon* i ($i = 1, 2, \ldots, n, \infty$). The incidence relations among vertices, edges and polygons are represented by ten integer-value arrays, one for polygons, one for vertices and eight for edges, which are defined in the following way.

For polygon i ($i = 1, 2, \ldots, n, \infty$):

edge.around.polygon[i]: the ordinal number of an edge on the boundary of the polygon i.

For vertex j ($j = 1, 2, \ldots, n_v$):

edge.around.vertex[j]: the ordinal number of an edge incident to the vertex j.

For each edge k ($k = 1, 2, \ldots, n_e$) (Figure 4.2.2 shows the following eight geometric objects represented by these arrays for edge k):

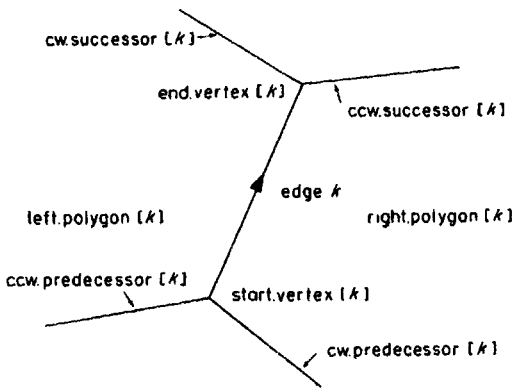

Figure 4.2.2 Geometric objects represented by the eight arrays associated with an edge.

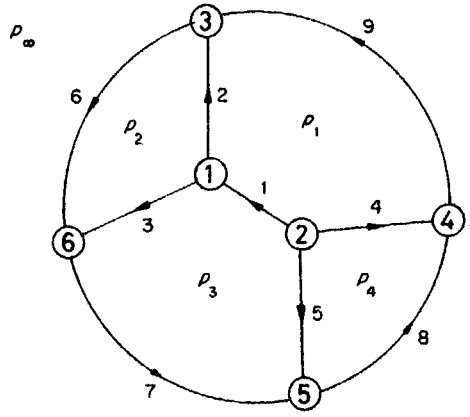

Figure 4.2.3 Directed augmented geometric graph associated with the Voronoi diagram in Figure 4.2.1.

right.polygon[k]: the ordinal number of the polygon that is to the right of the edge k,
left.polygon[k] : the ordinal number of the polygon that is to the left of the edge k,
start.vertex[k]: the ordinal number of the start vertex of the edge k,
end.vertex[k]: the ordinal number of the end vertex of the edge k,
cw.predecessor[k]: the ordinal number of the edge next to edge k clockwise around the start vertex,
ccw.predecessor[k]: the ordinal number of the edge next to edge k counterclockwise around the start vertex,
cw.successor[k]: the ordinal number of the edge next to edge k clockwise around the end vertex,
ccw.successor[k]: the ordinal number of the edge next to edge k counterclockwise around the end vertex,

where cw and ccw indicate '*cw*ise' and '*ccw*ise', respectively, and 'predecessor' and 'successor' represent edges around the start vertex and the end vertex, respectively. For each polygon i, we simply store in edge.around.polygon[i] one of the edges incident to the polygon, and similarly for each vertex j we store in edge.around.vertex[j] one of the edges incident to the vertex. Most information is stored in the eight arrays for edges (Figure 4.2.2).

Figure 4.2.3 shows a directed geometric graph obtained from the graph in Figure 4.2.1(b) by choosing directions arbitrarily and by ordering the vertices and the edges arbitrarily, and Table 4.2.1 shows the contents of the ten arrays representing this graph. Here we use the symbol ∞ to represent the outermost infinite region; this is simply because we want to emphasize that this region does not correspond to an actual Voronoi polygon, and in practical implementations the symbol ∞ may be replaced by the integer $n+1$.

The above ten arrays together represent the topological structure of the Voronoi diagram, but the metric aspect of the Voronoi diagram should also be represented. For this purpose, we use three more arrays, w.vertex[j], x.vertex[j] and y.vertex[j], attached to the vertices. The array w.vertex[j] represents one-bit information in such a way that

$$\text{w. vertex}[j]=\begin{cases}1 & \text{if vertex } j \text{ is an ordinary point,}\\ 0 & \text{if vertex } j \text{ is a point at infinity.}\end{cases}$$

If vertex j is an ordinary point, x.vertex[j] and y.vertex[j] represent the x and the y coordinates of this vertex, while if vertex j is a point at infinity, x.vertex[j] and y.vertex[j] represent the x and the y components of the unit vector designating the direction in which the associated (infinite) Voronoi edge runs. Thus, each entry of the array w.vertex requires one bit of memory space, while each entry of x.vertex or y.vertex requires one word to represent a floating-point number. The triple (w.vertex[j], x.vertex[j], y.vertex[j]) is a special case of what is usually called 'homogeneous coordinates', which are useful for representing ordinary points and points at infinity in a unified manner (e.g. Pedoe, 1970).

The collection of the above thirteen arrays is called the *winged-edge data structure*. In what follows by the construction of the Voronoi diagram $\mathcal{V}(P)$ we mean the construction of the winged-edge data structure for the augmented geometric graph associated with $\mathcal{V}(P)$.

In the augmented geometric graph associated with the Voronoi diagram, there are exactly $n+1$ polygons, and at most $3n-3$ edges and $2n-2$ vertices. Hence in total the winged-edge data structure requires memory space for $n + 1 + (2n - 2) + 8(3n - 3) = 27n - 25$ integers, $2n - 2$ bits and $2(2n - 2)$

Table 4.2.1 Entries of the ten integer arrays in the winged-edge data structure associated with the augmented geometric graph shown in Figure 4.2.3.

k (edge number)	1	2	3	4	5	6	7	8	9
right.polygon[k]	1	1	2	4	3	∞	∞	∞	∞
left.polygon[k]	3	2	3	1	4	2	3	4	1
start.vertex[k]	2	1	1	2	2	3	6	5	4
end.vertex[k]	1	3	6	4	5	6	5	4	3
cw.predecessor[k]	4	1	2	5	1	9	6	7	8
ccw.predecessor[k]	5	3	1	1	4	2	3	5	4
cw.successor[k]	3	6	7	9	8	3	5	4	2
ccw.successor[k]	2	9	6	8	7	7	8	9	6
i (polygon number)			1	2	3	4	∞		
edge.around.polygon[i]			1	2	1	4	6		
j (vertex number)			1	2	3	4	5	6	
edge.around.vertex[j]			1	1	2	4	5	3	

$= 4n - 4$ floating-point numbers. To represent the Voronoi diagram completely, we also need $2n$ more floating-point numbers for the x and the y coordinates of the generators in P, which is not the output of the algorithm but the input to the algorithm.

Once we obtain the winged-edge data structure, we can easily retrieve a variety of fundamental information about the Voronoi diagram. For example, if we want to obtain the list of edges and polygons that are incident to vertex j, we can use the next algorithm. In the algorithm, a 'list' means data containing a sequence of items, and 'to add an item to the tail of a list' means to augment the list by adding the item as the last element of the sequence. For example, if we add b to the tail of list $(b_1, b_2, \ldots, b_k)$, we get $(b_1, b_2, \ldots, b_k, b)$.

Algorithm 4.2.1 (Retrieve the edges and polygons incident to a vertex)
Input: Winged-edge data structure of $\mathcal{V}(P)$ and vertex j.
Output: List L_e of edges and list L_r of polygons that surround vertex j counterclockwise.
Procedure:
Step 1. $L_e \leftarrow$ empty list and $L_r \leftarrow$ empty list.
Step 2. $k \leftarrow$ edge.around.vertex[j], and *kstart* $\leftarrow k$.
Step 3. Add k to the tail of the list L_e.
Step 4. If j = start.vertex[k] then add left.polygon[k] to the tail of L_r and $k \leftarrow$ ccw.predecessor[k],
else add right.polygon[k] to the tail of L_r and $k \leftarrow$ ccw.successor[k].
Step 5. If k = *kstart* then return L_e and L_r, else go to Step 3.

In this description of the algorithm, we use several conventions common in computer science (Aho *et al.*, 1974). '$x \leftarrow y$' represents 'replace the value of variable x by the value of y', 'if C then X else Y' represents 'do X if the condition C is true, and do Y otherwise', and 'return x' represents 'report x as the output and exit from the procedure'.

In this algorithm, Steps 1 and 2 are done once, and Steps 3, 4 and 5 are repeated as many times as the number of edges (or, equivalently, the number of polygons) incident to vertex j. Hence, the total running time of this algorithm is proportional to the degree of vertex j.

If we want to get the list of edges and vertices that surround polygon i counterclockwise, we can use the next algorithm.

Algorithm 4.2.2 (Retrieve the edges and vertices surrounding a polygon)
Input: Winged-edge data structure of $\mathcal{V}(P)$ and polygon i.
Output: List L_e of edges and list L_v of vertices that surround polygon i counterclockwise.
Procedure:
Step 1. $L_e \leftarrow$ empty list, and $L_v \leftarrow$ empty list.
Step 2. $k \leftarrow$ edge.around.polygon[i], and *kstart* $\leftarrow k$.

Step 3. Add k to the tail of list L_e.
Step 4. If i = left.polygon[k]
then add end.vertex[k] to the tail of L_v and $k \leftarrow$ cw.successor[k],
else add start.vertex[k] to the tail of L_v and $k \leftarrow$ cw.predecessor[k].
Step 5. If $k = kstart$ then return L_e and L_v, else go to Step 3.

This algorithm runs in time proportional to the number of edges on the boundary of polygon i, because Steps 1 and 2 are done once, and Steps 3, 4 and 5 are repeated as many times as the number of edges on the boundary of polygon i.

In particular, if polygon ∞ is chosen as the input polygon, this algorithm gives the list of edges and vertices on the boundary of the outermost infinite region, i.e. the list of edges and vertices on the boundary of the convex hull CH(P) (recall Property V2).

Sometimes we want to retrieve the structure of the Delaunay diagram, the dual of the Voronoi diagram. The winged-edge data structure is also convenient for this purpose. Indeed, if we exchange the roles of the vertices and polygons, the winged-edge data structure for a geometric graph itself can be considered as the winged-edge data, structure for the dual graph. For example, edge k corresponds to the Delaunay edge connecting two Delaunay vertices (i.e. generators) represented by right.polygon[k] and left.polygon[k], and the Delaunay polygon corresponding to the jth Voronoi vertex can be obtained as the output of Algorithm 4.2.1. Thus, what we have to do to retrieve information about the Delaunay diagram is simply to rename the geometric concepts by their dual concepts. That is, we consider that the edge k is a Delaunay edge connecting two generators right.polygon[k] and left.polygon[k], and that the Voronoi vertex i is a Delaunay triangle one of whose edges is edge.around.vertex[i], etc.

Before closing this section let us see the lower bound of the time complexity for constructing the Voronoi diagram. We can see that the lower bound of the worst-case time complexity for constructing the Voronoi diagram for n points is O($n$log n). This lower bound can be derived from the well-known fact that any algorithm for sorting n real numbers in increasing order using comparisons requires at least O($n \log n$) time in the worst case (Aho *et al.*, 1974).

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n real numbers. We create the set $P = \{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}$ of n points in the plane and construct the Voronoi diagram for P. From this Voronoi diagram we obtain the cyclic sequence of points on the boundary of the convex hull CH(P) by Algorithm 4.2.2 in O(n) time. Since all the points in P are on the boundary of CH(P), as shown in Figure 4.2.4, we obtain the increasing order of the n numbers in X. Thus, if the Voronoi diagram for P could be constructed faster than in O($n \log n$) time, then we could sort numbers in X faster than in O($n \log n$) time, which is a contradiction. Hence, any algorithm for constructing the Voronoi diagram for n points takes at least O($n \log n$) time in the worst case.

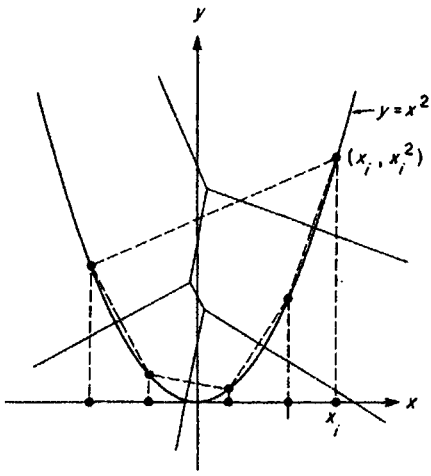

Figure 4.2.4 Relation between the sorting problem and the convex hull problem.

This proof also contains the proof of the property that any algorithm for constructing the cyclic sequence of points on the boundary of the convex hull of n points takes at least $O(n \log n)$ time in the worst case.

It may seem that some kind of 'sorting' is a bottleneck in constructing the Voronoi diagram. However, even if the sorted order of the generators in one direction is given, the construction of the Voronoi diagram still requires $O(n \log n)$ time (Djidjev and Lingas, 1995).

On the other hand, the lower bound of the average-case time complexity for constructing the Voronoi diagram is obviously $O(n)$; this is because any algorithm should create the winged-edge data structure whose size is $O(n)$.

As we will see, both of the lower bounds are tight in the sense that we can actually construct an algorithm whose worst-case time complexity is $O(n \log n)$ and an algorithm whose average-case time complexity is $O(n)$.

4.3 THE INCREMENTAL METHOD

The incremental method is one of the most important methods because it is conceptually simple and its average time complexity can be decreased to $O(n)$ by some algorithmic techniques. This method starts with a simple Voronoi diagram for a few generators, say two or three generators, and modifies the diagram by adding other generators one by one. For $l = 1, 2, \ldots, n$, let $\mathcal{V}_l$ denote the Voronoi diagram for the first l generators $p_1, p_2, \ldots, p_l$. The main part of the incremental method is to convert $\mathcal{V}_{l-1}$ to $\mathcal{V}_l$ for each l.

Figure 4.3.1 shows an example of the addition of a generator. Suppose that we have already constructed the Voronoi diagram $\mathcal{V}_{l-1}$ as shown by the solid lines, and that we now want to add a new generator p_l. First, we find the generator, say p_i, whose Voronoi polygon contains p_l, and draw the perpendicular bisector between p_l and p_i. The bisector crosses the boundary

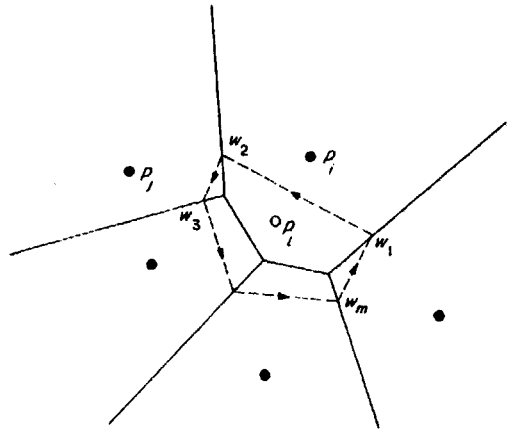

Figure 4.3.1 Addition of a new generator p_l.

of $\mathcal{V}(p_i)$ at two points; let the two points be w_1 and w_2 in such a way that p_l is to the left of the directed line segment $\overrightarrow{w_1w_2}$. The line segment $\overrightarrow{w_1w_2}$ divides the Voronoi polygon $V(p_i)$ into two portions, the one on the left belonging to the Voronoi polygon of p_l. Thus, we get a Voronoi edge on the boundary of the Voronoi polygon of p_l.

Starting with the edge $\overrightarrow{w_1w_2}$, we 'grow' the boundary of the Voronoi polygon of p_l by the following procedure, which we call the *boundary growing procedure*. The bisector between p_i and p_l crosses the boundary of $V(p_i)$ at w_2, entering the adjacent Voronoi polygon, say $V(p_j)$. So, we next draw the perpendicular bisector between p_l and p_j, and find the point (other than w_2) at which the bisector crosses the boundary of $V(p_j)$; let this point be w_3. In a similar way, we find the sequence of segments of perpendicular bisectors of p_l and the neighbouring generators until we reach the starting point w_1. Let this sequence be $(\overline{w_1w_2}, \overline{w_2w_3}, \ldots, \overline{w_{m-1}w_m}, \overline{w_mw_1})$. This sequence forms a counterclockwise boundary of the Voronoi polygon of the new generator p_l. Finally, we delete from $\mathcal{V}_{l-1}$ the substructure inside the new Voronoi polygon, and thus obtain $\mathcal{V}_l$.

The above description is a rough sketch of the incremental method. To make an exact and efficient algorithm we must be careful at several points.

First, we have to take care that the boundary growing procedure works well only when the Voronoi polygon of the new generator p_l is finite, although a Voronoi polygon is not necessarily finite. An infinite region, if it happens, requires an exceptional branch of processing. However, such an exception can be avoided by a simple trick.

Recall that, in the Voronoi diagram $\mathcal{V}$ for the generator set P, the Voronoi polygon $V(p_i)$ is infinite if and only if p_i is on the boundary of the convex hull of P (Property V2). Keeping this in mind, let us renumber the generators as $p_4, p_5, \ldots, p_n$ (now n is the number of generators plus three), and introduce three additional generators p_1, p_2 and p_3 in such a way that the

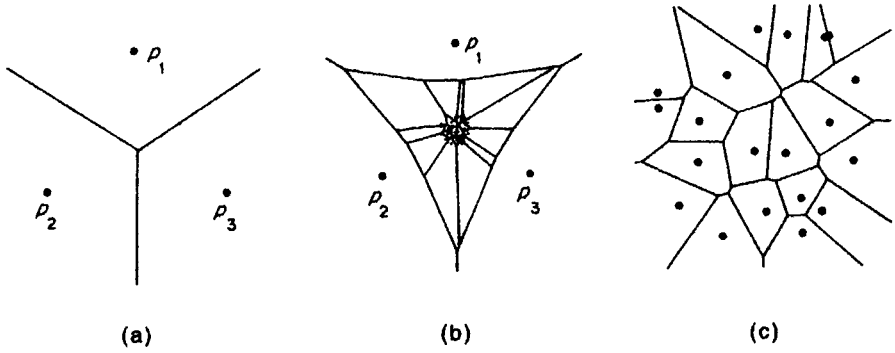

Figure 4.3.2 Avoidance of infinite Voronoi polygons: (a) Voronoi diagram for three additional generators; (b) Voronoi diagram for all the generators; (c) Voronoi diagram for the original generators.

triangle $p_1p_2p_3$ contains all the original generators $p_4, p_5, \ldots, p_n$ in its interior. The Voronoi diagram for p_1, p_2 and p_3 is a simple one, as shown in Figure 4.3.2(a). We start with this Voronoi diagram, and add other generators $p_4, p_5, \ldots, p_n$ one by one. The convex hull of $p_1, p_2, \ldots, p_l$ is the triangle $p_1p_2p_3$ itself for any $l = 4, 5, \ldots, n$, and consequently the Voronoi polygon of p_l in the Voronoi diagram $\mathcal{V}_l$ is always finite, as shown in Figure 4.3.2(b). Thus, we can avoid the occurrence of an infinite Voronoi polygon in the boundary growing procedure.

The Voronoi diagram constructed in this way is not the same as the Voronoi diagram for the original generators. However, the difference due to the additional three generators can be eliminated substantially if a sufficiently large triangle is chosen. For example, if we magnify the central portion of Figure 4.3.2(b), we get (c), where the effect of the additional generators does not appear. Without loss of generality, let us assume that all the original generators $p_4, p_5, \ldots, p_n$ are located in the unit square $S = \{(x, y) \mid 0 \le x, y \le 1\}$, and that we are interested in the Voronoi diagram in this square area. Then, we can choose the three additional generators, for example, by

$$\begin{aligned} p_1 &= (0.5, 3\sqrt{2}/2 + 0.5), \\ p_2 &= (-3\sqrt{6}/4 + 0.5, -3\sqrt{2}/4 + 0.5), \\ p_3 &= (3\sqrt{6}/4 + 0.5, -3\sqrt{2}/4 + 0.5). \end{aligned} \tag{4.3.1}$$

These three points form a sufficiently large regular triangle which contains the square area $\{(x, y) \mid 0 \le x, y \le 1\}$; see Figure 4.3.2(b). We can see that no boundary of the Voronoi polygon of the additional generators goes through the interior of the square S unless the set of original generators is empty (actually, we can prove that no such boundary goes through the interior of the circle circumscribing S; the proof is left to the reader).

The second point we should take care over is the time complexity. The first task in the addition of a new generator p_l is to find the old

generator p_i whose Voronoi polygon contains p_l. Obviously p_i satisfies $d(p_l, p_i) \leq d(p_l, p_j)$ for any $j = 1, 2, \ldots, l-1$. Hence, p_i can be found if the distances from p_l to all the other generators are computed. However, this requires $O(l)$ time, so that the total time complexity will become $O(1 + 2 + \ldots + (n-1)) = O(n^2)$.

A slightly more intelligent way is to start with an initial guess and to search its neighbours for a closer generator. This can be done using the next algorithm.

Algorithm 4.3.1 (Nearest neighbour search)

Input: l generators $p_1, p_2, \ldots, p_l$, Voronoi diagram $\mathcal{V}_{l-1}$, and initial guess $p_i (1 \leq i \leq l-1)$.

Output: The generator (other than p_l) that is the closest to p_l.

Procedure:

Step 1. Among the generators adjacent to p_i, find the one, say p_j, with the minimum distance to p_l:

$$d(p_j, p_l) = \min_k d(p_k, p_l),$$

where the minimum is taken over all generators p_k whose Voronoi polygons are adjacent to $V(p_i)$.

Step 2. If $d(p_i, p_l) \leq d(p_j, p_l)$, return p_i, else $p_i \leftarrow p_j$ and go to Step 1.

The algorithm always terminates in a finite number of steps, because each replacement of p_i at Step 2 gives a closer generator than before. Moreover, the algorithm gives the closest generator as the output, because, if p_i is not closest to p_l, there always exists an adjacent generator closer to p_l (indeed, if we move from p_i in the direction to p_l, we cross the boundary of $V(p_i)$ and visit the neighbouring polygon, say the Voronoi polygon of p_j, which is closer to p_l; the proof is left to the reader).

Note that the time required by the algorithm depends greatly on the initial guess p_i; if it is near to the closest generator, we reach the closest generator quickly. Hence, it is important to make a good initial guess. For this purpose we employ a bucketing technique. Before describing this technique, however, we present another important point.

The final point we should be careful of is the size of the substructure to be deleted, because, if the substructure is large and complicated, we cannot find and delete it in constant time. As we have seen, the average number of edges on the boundary of a Voronoi polygon is six or less (recall Property V13 in Section 2.3). Consequently, we can expect that the average size of the substructure to be deleted is almost constant. However, we should not be too optimistic, because the average number of edges of a Voronoi polygon being six does not necessarily imply that the average number of edges of a 'new' Voronoi polygon will be six.

A counterexample is shown in Figure 4.3.3, where the generators align along a spiral and are numbered from outermost inward. The Voronoi polygon of the new generator, the innermost one, is adjacent to all the other

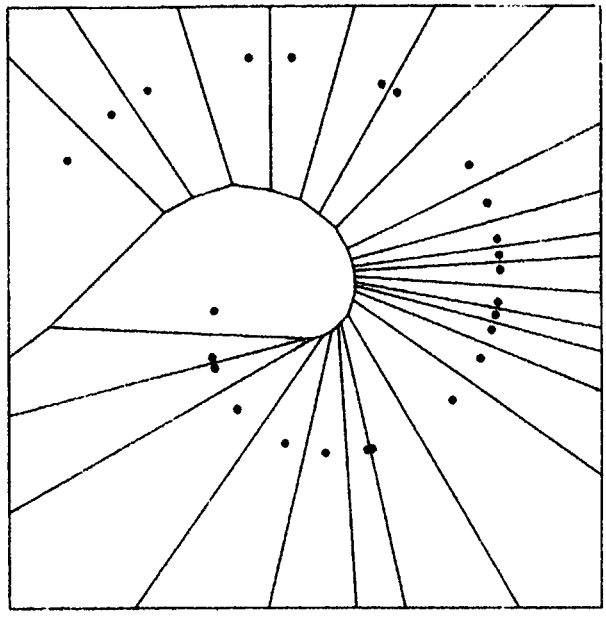

Figure 4.3.3 Voronoi diagram for points on a spiral.

polygons, so that the size of the substructure to be deleted is proportional to the number of generators added so far.

To restrict the average size of the substructure to be constant, the generators should be numbered in such a way that for any $l = 4, 5, \ldots, n$, the generators $p_4, p_5, \ldots, p_l$ distribute as uniformly as possible. Hence before starting the incremental method, we have to reorder the generators to meet this property as nearly as possible. For this purpose also, we use the bucketing technique.

A *rooted tree* is a tree in which one node is specified as the *root* (see Figure 4.3.4). When we draw a rooted tree, we put the root at the top, and draw other nodes downward in such a way that nodes at the same distance from the root align horizontally (where the distance between two nodes is the number of links on the path connecting them). For each node u, the unique one-level upper node adjacent to u is called the *parent* of u, and one-level lower nodes adjacent to u (if any) are called *children* of u. The root is the only node that does not have a parent. A node without children is called a *leaf*. A node which is neither the root nor a leaf is called an *intermediate node*. If node u is on the path from node v to the root, u is called an *ancestor* of v and v a *descendant* of u. For any node u, the subgraph consisting of u and all its descendants is itself a rooted tree with the root u; this subgraph is called the *subtree* rooted at u. A rooted tree is called a *quaternary tree* if every node other than a leaf has exactly four children.

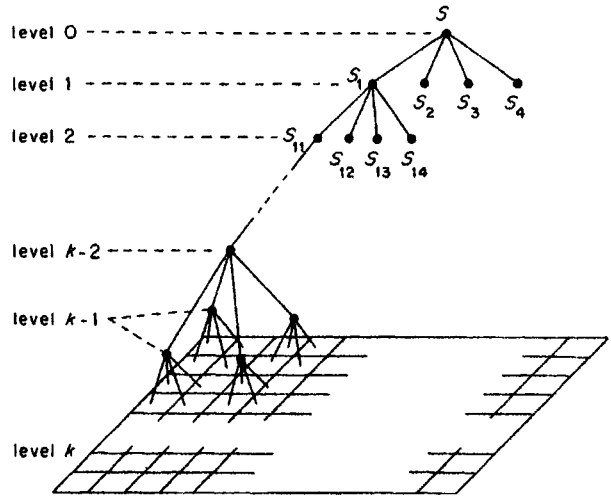

Figure 4.3.4 Quaternary tree with buckets as leaves.

Let k be a positive integer such that 4^k is nearest to n. Let us divide the unit square S into four smaller squares, which we shall call quadrants; hence we divide the square into the left upper quadrant S_1, the right upper quadrant S_2, the left lower quadrant S_3 and the right lower quadrant S_4. We repeat this process k times recursively. That is, first S is divided into S_1, S_2, S_3 and S_4, and next S_i $(1 \le i \le 4)$ is divided into S_{i1}, S_{i2}, S_{i3} and S_{i4}, and so on. The division procedure can be represented by a quaternary tree; the root corresponds to the entire region S, and for each node, its four child nodes correspond to the four smaller squares generated by the partition of the corresponding square, as shown in Figure 4.3.4. The first division process generates 4 level-one nodes, the second division process generates 4^2 level-two nodes, and so on; finally the kth process generates 4^k level-k nodes. The smallest squares, the squares corresponding to the level-k nodes, are called *buckets*.

This quaternary tree structure is useful both for finding a good initial guess for Algorithm 4.3.1 and for reordering the generators. For these purposes, we need a little more preprocessing.

With each node u of the quaternary tree we associate a generator, say $g[u]$, according to the following procedure. First, for all $l = 4, 5, \ldots, n$, we find the bucket containing p_l (since S is the unit square, the bucket containing p_l can be found from its coordinates by multiplying 2^k and then truncating off the fractional parts). Next, we put $g[u] \leftarrow p_1$ for the root node u. Thirdly, we visit the buckets one by one, and at each non-empty bucket we choose one generator, say p_i, contained in the bucket and put $g[u] \leftarrow p_i$ for all nodes u with $g[u]$ undefined on the path from this bucket to the root. This is all we have to do for the preprocessing.

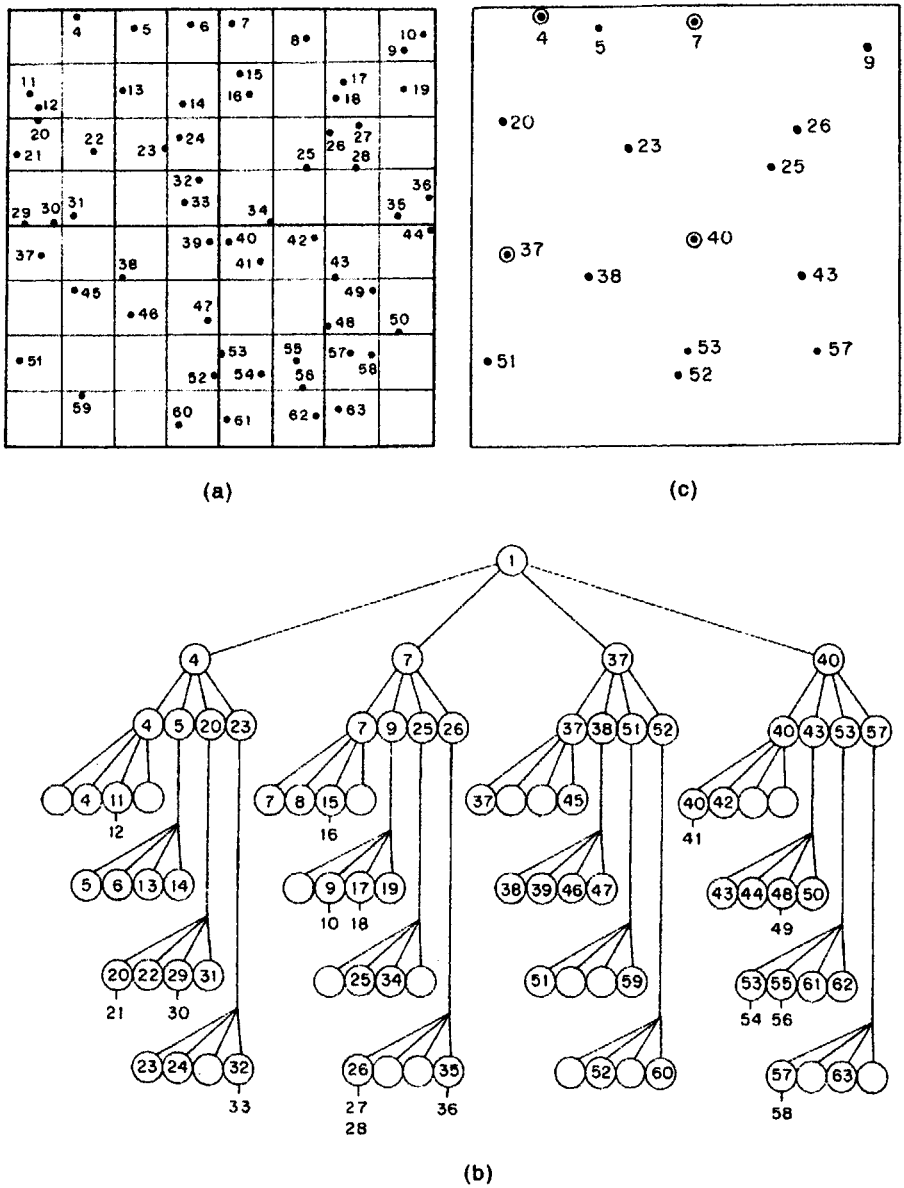

Figure 4.3.5 Randomly distributed points and the associated quaternary tree: (a) 60 points distributed randomly in a square and 64 buckets covering the square; (b) quaternary tree associated with the points and the buckets in (a); (c) points added when the nodes with level two or less are scanned.

Figure 4.3.5 shows an example of the quaternary tree structure. Panel (a) shows 60 points randomly located in a square. Since 60 is near to 64 ($= 4^3$), the square region is divided into 64 buckets, and the points are numbered from 4 to 63 in the order in which we encounter them when we visit the buckets from the top row downward and from left to right at each row; points in the same buckets are numbered randomly. Scanning the buckets in the above order, we construct the quaternary tree, as shown in (b), where for each node the four child nodes correspond, from left to right, to the left upper quadrant, the right upper quadrant, the left lower quadrant and the right lower quadrant, respectively. The number attached to each node represents the generator number $g[u]$. For a bucket containing two or more generators, the corresponding leaf node contains one of them and the other generators are stored in an additional list attached to the node.

Note that some buckets may contain two or more generators and some buckets may contain no generator, and hence some generators may not be assigned to any node and $g[u]$ may not be defined for some node u. In the main processing of the incremental method, we visit the nodes of the quaternary tree from the level-one nodes to the leaves in a breadth-first manner, that is, we visit all the level-i nodes before visiting a level-$(i+1)$ node. Remember that we defined $g[u] = p_1$ for the root node, and that we start with the initial Voronoi diagram generated by p_1, p_2 and p_3. Hence, we skip the root node, and start the addition at the level-one nodes. At each node u, if the associated generator $g[u]$ is new in the sense that we come across the generator $g[u]$ for the first time in the breadth-first visit, we add it to the current Voronoi diagram. Let u' be the parent node of u. Then the generator $g[u]$ is new if and only if $g[u] \neq g[u']$. If the visited node u is a bucket (i.e. a leaf node of the tree), and if the bucket contains generators other than $g[u]$, we add them to the current Voronoi diagram in an arbitrary order.

Adding generators in this order will keep the uniformness of the distribution of the generators relatively well, because at the end of visiting all the level-i nodes, exactly one generator is chosen from each non-empty square corresponding to that level. We call the above method for reordering *quaternary reordering*.

The quaternary tree is also useful for finding a good initial guess for Algorithm 4.3.1. Suppose that we are visiting node u. Let node u' be the parent of u. The generator $g[u]$ is added at this point if and only if $g[u] \neq g[u']$. We use $g[u']$ as the initial guess for Algorithm 4.3.1. If u is a leaf node and if the corresponding bucket contains two or more generators, we use $g[u]$ as the initial guess in the addition of the other generators in the bucket. This choice seems to give a good initial guess, because it lies either in the same square or in a one-level-larger square. We call this method for choosing the initial guess for Algorithm 4.3.1 *quaternary initial guessing*.

Figure 4.3.5(c) represents the points that are added by the end of scanning up to level-two nodes of the quaternary tree in (b). The four points designated by double circles are those added by the time the level-zero and level-one nodes are scanned, while the other points, represented by small

filled circles, correspond to those added in the scan of the level-two nodes. We can see that the 16 points shown in (c) are distributed relatively uniformly in the square.

Point 4 is used as the initial guess for the nearest neighbour search in the addition of points 5, 20 and 23, and similarly point 7 is used as the initial guess in the addition of points 9, 25 and 26, and so on. We see that the initial guess is either the nearest point itself or a point very near to the nearest point.

Summing up the above considerations, we obtain the next algorithm.

Algorithm 4.3.2 (Quaternary incremental method)

Input: Set $\{p_4, p_5, \ldots, p_n\}$ of $n-3$ generators located in the unit square $S = \{(x, y) \mid 0 \le x, y \le 1\}$.

Output: Voronoi diagram $\mathcal{V}$ for n generators $\{p_1, p_2, \ldots, p_n\}$, where p_1, p_2 and p_3 are the additional generators defined by equation (4.3.1).

Procedure:

Step 1. Find positive integer k such that 4^k is closest to n, divide S into 4^k square buckets, and construct the quaternary tree having the buckets as leaves.

Step 2. Renumber the generators by the quaternary reordering, and let the resultant order be $p_4, p_5, \ldots, p_n$.

Step 3. Construct the Voronoi diagram $\mathcal{V}_3$ for the three additional generators p_1, p_2 and p_3 defined by equation (4.3.1), as shown in Figure 4.3.2(a).

Step 4. For $l = 4, 5, \ldots, n$, do 4.1, ..., 4.4.

4.1. By Algorithm 4.3.1 with the initial guess given by the quaternary initial guessing, find the generator p_i $(1 \le i \le l-1)$ closest to p_l.

4.2. Find the points w_1 and w_2 of intersections of the perpendicular bisector of p_i and p_l with the boundary of $V(p_i)$.

4.3. By the boundary growing procedure, construct the closed sequence $(\overline{w_1w_2}, \overline{w_2w_3}, \ldots, \overline{w_{m-1}w_m}, \overline{w_mw_1})$ of segments of perpendicular bisectors forming the boundary of the Voronoi polygon of p_l.

4.4. Delete from $\mathcal{V}_{l-1}$ the substructure inside the closed sequence, and name the resultant diagram $\mathcal{V}_l$.

Step 5. Return $\mathcal{V} = \mathcal{V}_n$.

This algorithm does not refer to the associated winged-edge data structure explicitly, but it is not difficult to understand how to create and modify the winged-edge data structure accordingly; the initial winged-edge data structure for $\mathcal{V}_3$ is created in Step 3 and is modified in Step 4.4. To describe Steps 3 and 4.4 in terms of the winged-edge data structure is left as an exercise. Obviously, Steps 1 and 2 are done in $O(n)$ time, and Steps 3 and 5 in $O(1)$ time. Because of the quaternary reordering and the quaternary initial guessing, we can expect that Steps 4.1, ..., 4.4 can be done in $O(1)$ time on

average for each l. Thus, in total, Algorithm 4.3.2 runs in $O(n)$ time on average if the generators are distributed at random in a square.

The use of buckets with the quaternary tree was proposed by Ohya, Iri and Murota (Ohya, 1983; Ohya *et al.*, 1984a,b), by which the average time complexity of the incremental method decreases from $O(n^2)$ to $O(n)$. They also recommended a certain specific order of visiting the buckets in the preprocessing and of visiting tree nodes in the main processing (see Ohya *et al.*, 1984a, for the details of their reordering). Guibas *et al.* (1992) and de Berg *et al.* (1995) studied the random-order insertion.

4.4 THE DIVIDE-AND-CONQUER METHOD

In this and the next sections we give two $O(n\log n)$ algorithms for constructing the Voronoi diagram, which are optimal in a worst-case sense. They are indeed important from a theoretical point of view. However, since we have in practice already obtained a more efficient method in the previous section, we describe only the basic ideas of these algorithms, skipping details for implementation.

'Divide-and-conquer' is one of the fundamental paradigms for designing efficient algorithms. In this paradigm the original problem is recursively divided into several simple subproblems of almost equal size, and the solution of the original problem is obtained by merging the solutions of the subproblems.

For our particular problem, the generators are sorted in increasing order of the x coordinates, and are represented by the 'list' $P = (p_1, p_2, \ldots, p_n)$ (unlike the incremental method, we do not use any additional generators; $p_1, \ldots, p_n$ are all original generators). This can be done in $O(n \log n)$ time by any optimal sorting algorithm such as a heap sort or a merge sort (cf. Knuth, 1973, or Aho *et al.*, 1974). In addition to Assumptions A4.1.1 and A4.1.2, here we assume the following.

Assumption A4.4.1 No two generators align vertically.

Hence the order $(p_1, p_2, \ldots, p_n)$ is defined uniquely. Next, the divide-and-conquer paradigm is applied in the following way (we first state the algorithm formally and then give an example).

Algorithm 4.4.1 (Divide-and-conquer method)

Input: Number n of generators and list $P = (p_1, p_2, \ldots, p_n)$ of the generators arranged in increasing order of the x coordinates.

Output: Voronoi diagram $\mathcal{V}$ for P.

Procedure:

Step 1. If $n \leq 3$, then construct the Voronoi diagram $\mathcal{V}$ for P directly and go to Step 3.

Step 2. Otherwise do the following.

2.1. Let t be the integral part of $n/2$, and divide P into $P_L = (p_1, \ldots, p_t)$ and $P_R = (p_{t+1}, \ldots, p_n)$.
2.2. Construct the Voronoi diagram $\mathcal{V}_L$ for P_L by Algorithm 4.4.1.
2.3. Construct the Voronoi diagram $\mathcal{V}_R$ for P_R by Algorithm 4.4.1.
2.4. Merge $\mathcal{V}_L$ and $\mathcal{V}_R$ into the Voronoi diagram $\mathcal{V}$ for P (this can be done by Algorithm 4.4.3.).

Step 3. Return $\mathcal{V}$.

Note that the generators in P_L are to the left of the generators in P_R. We call elements of P_L *left generators* and those of P_R *right generators*. $\mathcal{V}_L$ is called the *left Voronoi diagram* and $\mathcal{V}_R$ the *right Voronoi diagram*.

The essential portion of this algorithm is Step 2.4, the step for merging two Voronoi diagrams. In what follows we consider how to do this step.

Suppose that we have obtained the left Voronoi diagram $\mathcal{V}_L$ and the right Voronoi diagram $\mathcal{V}_R$, as shown in Figure 4.4.1, where the left generators are represented by filled circles, the right generators by unfilled circles, $\mathcal{V}_L$ by dot-and-dash lines and $\mathcal{V}_R$ by broken lines. Now we have to merge them by generating new Voronoi edges and new Voronoi vertices and by deleting superfluous portions from $\mathcal{V}_L$ and $\mathcal{V}_R$. The new edges and the new vertices come from the interaction between left generators and right generators; that

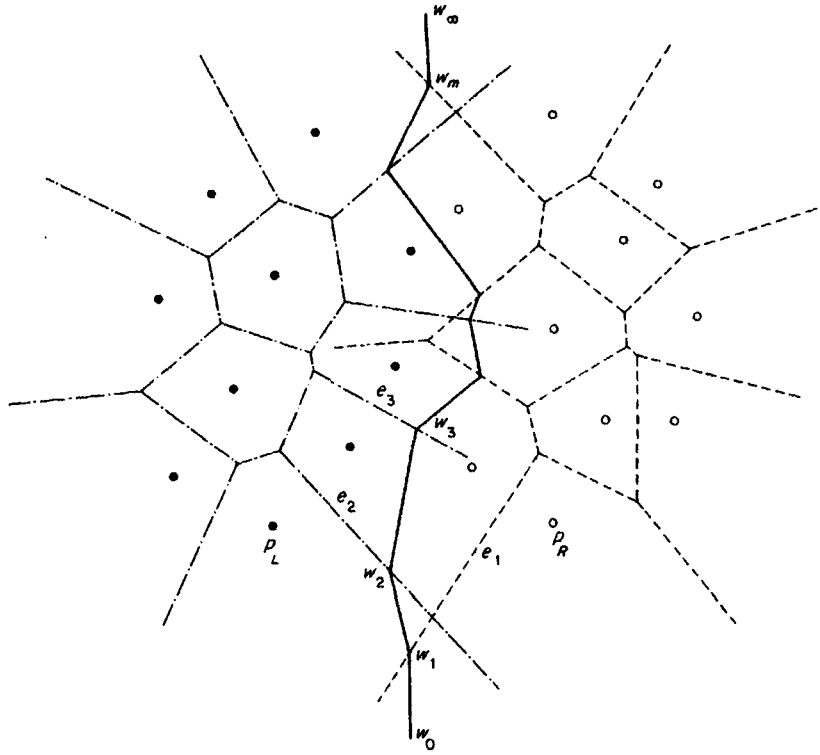

Figure 4.4.1 Merger of left and right Voronoi diagrams.

is, each new edge is part of the perpendicular bisector between a left generator and a right generator, and each new vertex is equidistant either from two left generators and one right generator or from one left generator and two right generators.

For any point p in the plane, let $d(p, P_L)$ denote the minimum distance from p to the generators in P_L, and $d(p, P_R)$ the minimum distance from p to the generators in P_R. Then, any point p on a new Voronoi edge satisfies $d(p, P_L) = d(p, P_R)$. Consider a point p that moves from left to right along an arbitrary horizontal line. Since P_L is to the left of P_R, there is a unique point p^* such that $d(p^*, P_L) = d(p^*, P_R)$ and $d(p, P_L) < d(p, P_R)$ before p reaches p^* and $d(p, P_L) > d(p, P_R)$ after p passes through p^*. As the horizontal line moves from the bottom upward, p^* moves continuously, tracing a polygonal line running between the left generators and the right generators. Our task is to construct this polygonal line and to remove from V_L and V_R the portions that lie to the right and to the left, respectively, of the polygonal line.

We construct the polygonal line from the bottom upward. To find the bottommost edge, we need one more preparation. We present an algorithmic tool for finding a line such that it touches two disjoint convex polygons and that all the other points of the polygons are on the same side of the line.

For any two points p and q, the directed line passing through p and q in this order is denoted by $L(p, q)$, and the directed line segment starting at p and ending at q is denoted by $\overrightarrow{pq}$.

Let U be a convex polygon defined by the cyclic list $(u_1, u_2, \ldots, u_s, u_1)$ of vertices which surround the polygon counterclockwise. For the vertex u in U, let cnext[u] and ccnext[u] be the immediate predecessor and the immediate successor, respectively, of u in the list (cnext[u] is the *c*lockwise *next* vertex of u and ccnext[u] is the *c*ounterclockwise *next* vertex of u).

Let U_L and U_R be two convex polygons, U_L being to the left of U_R, i.e. the x coordinates of vertices in U_L are smaller than the x coordinates of vertices in U_R. For the vertex u in U_L and the vertex w in U_R the line $L(u, w)$ is called a *common support* of U_L and U_R if all the nodes in U_L and U_R are on the same side of the line or on the line. U_L and U_R admit two common supports; the upper one is called the *upper common support* and the lower one the *lower common support*. For instance, for two polygons U_L and U_R represented by the solid lines in Figure 4.4.2, the lower common support is the line connecting the left vertex d and the right vertex g. The lower common support can be found by the next algorithm (an example of the behaviour of the algorithm follows the algorithm).

Algorithm 4.4.2 (Lower common support)

Input: Two convex polygons U_L and U_R such that the maximum x coordinate over all vertices in U_L is smaller than the minimum x coordinate over all vertices in U_R.

Output: Pair consisting of the vertex u in U_L and the vertex w in U_R such that $L(u, w)$ forms the lower common support of U_L and U_R.

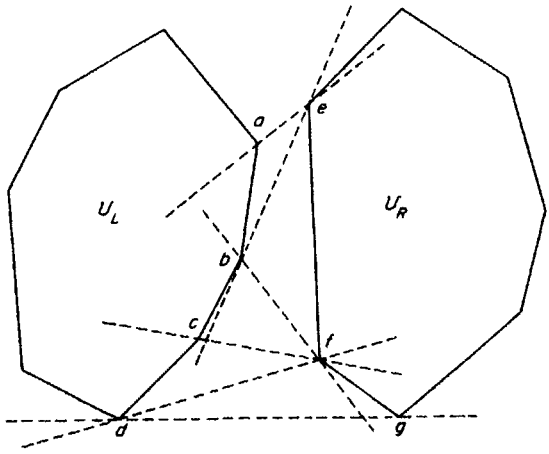

Figure 4.4.2 Search for the lower common support.

Procedure:

Step 1. Find the vertex u in U_L with the largest x coordinate, and the vertex w in U_R with the smallest x coordinate.

Step 2. Do 2.1 and 2.2 alternately until u and w are not changed any more.

2.1. While vertex cnext[u] is lower than $L(u, w)$, repeat $u \leftarrow$ cnext[u].

2.2. While vertex ccnext[w] is lower than $L(u, w)$, repeat $w \leftarrow$ ccnext[w].

Step 3. Return $L(u, w)$.

An example of the behaviour of the algorithm is shown in Figure 4.2.2. First, the pair (u, v) is set to be (a, e) in Step 1. Next, the pair is changed to (b, e) in Step 2.1, to (b, f) in Step 2.2, through (c, f) to (d, f) in Step 2.1, and finally to (d, g) in Step 2.2.

This algorithm terminates in a finite number of steps and gives the lower common support as the output. This can be understood in the following way. Let $(u_1, u_2, \ldots, u_s, u_1)$ be the list of vertices in U_L appearing counterclockwise on the boundary. Without loss of generality, suppose that u_1 is the vertex on the lower common support. Then, for any vertex w in U_R such that w is below the line $L(u_1, u_2)$, the vertex cnext[u_1] is not lower than the line $L(u_1, w)$. Thus, once u reaches u_1 in Step 2.1, it will not be updated any more. Similarly, once w reaches the vertex on the common lower support in Step 2.2, it will not be updated. Therefore, Step 2 terminates in a finite number of repetitions. Moreover, the termination condition for Step 2 actually implies that $L(u, w)$ is the lower common support.

Let n denote the total number of vertices in U_L and U_R. In Step 1, the initial pair (u, w) can be found by scanning the lists of vertices. In Step 2, the total number of updates of the pair (u, w) does not exceed n. Hence,

Algorithm 4.4.2 runs in $O(n)$ time. Actually there is an $O(\log n)$ algorithm for finding the common support (Overmars and Leeuwen, 1981), but an $O(n)$ algorithm is enough for our purpose because the subproblem is not a bottleneck of the total time complexity.

Now we are ready to consider our main problem. We want to construct the polygonal line consisting of new Voronoi edges and new Voronoi vertices. As shown in Figure 4.4.1, let p_L and p_R be the pair of the left generator and the right generator forming the lower common support $L(p_L, p_R)$ of the convex hull of P_L and that of P_R, then $b(p_L, p_R)$, the perpendicular bisector of $\overline{p_L p_R}$, gives the bottommost edge of the polygonal line, because $\overline{p_L p_R}$ is an edge of the convex hull of $P_L \cup P_R$ but is not an edge of the convex hull of P_L or of P_R. Hence, we start with this pair (p_L, p_R).

We consider a point w moving along the perpendicular bisector between p_L and p_R, from the bottom upward. If the y coordinate of the point w is sufficiently small, w belongs to both $V(p_L)$ and $V(p_R)$. Let w_0 be the point at infinity in the negative y direction along the bisector, and let w_1 be the point at which w crosses a Voronoi edge, say e_1, of $\mathcal{V}_L$ or $\mathcal{V}_R$ for the first time (as in Figure 4.4.1). Then, the portion of the bisector along which w has moved so far, i.e. the half line $\overline{w_0 w_1}$, is the first new Voronoi edge, and w_1 is the first new Voronoi vertex.

At w_1 the moving point w changes the region either in $\mathcal{V}_L$ or in $\mathcal{V}_R$, depending on whether e_1 is an edge of $\mathcal{V}_L$ or of $\mathcal{V}_R$. If e_1 is an edge of $\mathcal{V}_L$, we replace p_L with the left generator that is on the other side of e_1. If e_1 is an edge of $\mathcal{V}_R$ (as in Figure 4.4.1), we replace p_R with the other right generator generating e_1. Next, we draw the perpendicular bisector between the updated pair of p_L and p_R. This bisector has w_1 on it. Suppose that w moves from w_1 along the new bisector upward until it crosses a Voronoi edge of $\mathcal{V}_L$ or $\mathcal{V}_R$ again. Let the crossing point be w_2. Thus we obtain the second new Voronoi vertex w_2 and the second new Voronoi edge $\overline{w_1 w_2}$.

We repeat similar procedures until we reach the pair (p_L, p_R) forming the upper common support. Thus, we obtain the next algorithm.

Algorithm 4.4.3 (Merger of two Voronoi diagrams)

Input: Two Voronoi diagrams $\mathcal{V}_L$ and $\mathcal{V}_R$ for generator sets P_L and P_R, respectively, such that the generators in $\mathcal{V}_L$ have smaller x coordinates than those in $\mathcal{V}_R$.

Output: Voronoi diagram for $P_L \cup P_R$.

Procedure:

Step 1. Construct the convex hull of P_L and that of P_R.

Step 2. Find the lower common support $L(P_L, P_R)$ by Algorithm 4.4.2.

Step 3. $w_0 \leftarrow$ the point at infinity downward on $b(p_L, p_R)$, and $i \leftarrow 0$.

Step 4. While $L(p_L, p_R)$ is not the upper common support, repeat 4.1, ..., 4.4.

4.1. $i \leftarrow i + 1$.

4.2. Find the point a_L (other than w_{i-1}) of the intersection of $b(p_L, p_R)$ with the boundary of $V(p_L)$.

4.3. Find the point a_R (other than w_{i-1}) of the intersection of $b(p_L, p_R)$ with the boundary of $V(p_R)$.

4.4. If a_L has a smaller y coordinate than a_R,
$w_i \leftarrow a_L$, and
$p_L \leftarrow$ the generator on the other side of the Voronoi edge containing a_L.
Otherwise
$w_i \leftarrow a_R$, and
$p_R \leftarrow$ the generator on the other side of the Voronoi edge containing a_R.

Step 5. $m \leftarrow i$.
$w_{m+1} \leftarrow$ the point at infinity upward on $b(p_L, p_R)$.

Step 6. Add the polygonal line $(\overline{w_0w_1}, \overline{w_1w_2}, \ldots, \overline{w_mw_{m+1}})$, and delete from $\mathcal{V}_L$ the part to the right of the polygonal line and delete from $\mathcal{V}_R$ the part to the left of the polygonal line. Return the resultant diagram.

Step 1 is done in $O(n)$ time because we have already obtained the Voronoi diagrams $\mathcal{V}_L$ and $\mathcal{V}_R$ (recall Algorithm 4.1.2). Step 2 is done in $O(n)$ time by Algorithm 4.4.2, and Step 3 in constant time. Step 4 is repeated at most $O(n)$ times, because there are at most $O(n)$ new Voronoi edges. Steps 4.1 and 4.4 require only constant time.

Steps 4.2 and 4.3 require time proportional to the number of edges on the boundary of $V(p_L)$ and to that of $V(p_R)$, respectively, which are not bounded by any fixed constant. However, we can decrease the total number of edges to be examined in Steps 4.2 and 4.3 over all repetitions of Step 4. For this purpose, we examine edges on the boundary of $V(p_L)$ counterclockwise, and those of $V(p_R)$ clockwise. Suppose that, in a certain repetition of Step 4, we have found the intersections a_L and a_R, as shown in Figure 4.4.3. Since a_L is lower than a_R in this example, p_L is replaced by p_L'; as a result of this, the polygonal line bends to the right. Therefore, when we search for the intersection a_R' of $b(p_L', p_R)$ with the boundary of $V(p_R)$, we need not examine the edges to the left of a_R; we can start searching at a_R clockwise. Thus we can avoid the repeated check of irrelevant edges. Similarly, if a_R is lower than a_L, p_R is replaced and the polygonal line bends to the left, so that we can avoid unnecessary checks of edges by examining them counterclockwise. By doing so, we can execute Step 4 in $O(n)$ time.

Step 5 requires constant time. Step 6 requires $O(n)$ time. Hence, in total Algorithm 4.4.3 runs in $O(n)$ time.

Next, let us consider the time complexity of our main algorithm, Algorithm 4.4.1. The algorithm is recursive in the sense that the procedure calls itself at Steps 2.2 and 2.3. To evaluate the time complexity of such an algorithm, we first consider the other steps. Obviously, Steps 1, 2.1 and 3 can be done in constant time. Step 2.4 can be done in $O(n)$ time by Algorithm 4.4.3. Thus, all steps other than the recursive calls require $O(n)$ time.

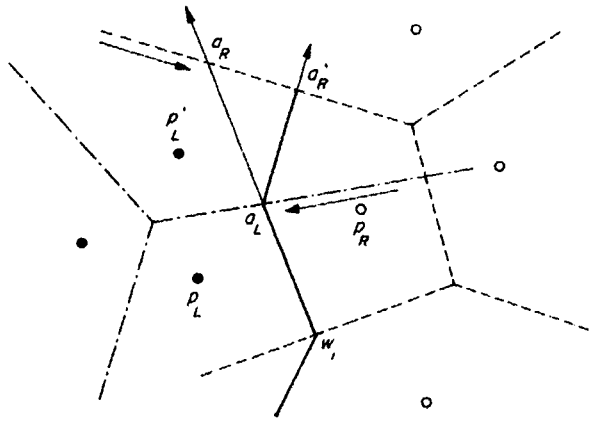

Figure 4.4.3 Search for the next Voronoi vertex on the boundary of the left and the right Voronoi diagrams.

Now, let $T(n)$ denote the time required by Algorithm 4.4.1 for n generators. Then, we get

$$T(n) = 2T(n/2) + O(n),$$

because the algorithm calls itself twice using half of the original input as the new input. With the consideration that $T(1)$ and $T(2)$ are constant, this equation implies that $T(n) = O(n \log n)$ (Aho *et al.*, 1974).

Recall that any algorithm for constructing a Voronoi diagram requires $O(n \log n)$ time in the worst case (Section 4.2). Thus, Algorithm 4.4.1 is optimal from the viewpoint of the worst-case time complexity.

The divide-and-conquer technique was first applied to this problem by Shamos and Hoey (1975); since then many variants have been proposed (Lee and Schachter, 1980; Guibas and Stolfi, 1985). It is also known that a divide-and-conquer method of this type actually requires $O(n \log n)$ time not only in the worst case but also on average (Ohya *et al.*, 1984a). Thus, in the average sense, this method is not optimal. Recently, quite different types of divide-and-conquer methods have been proposed; for uniformly distributed generators, the algorithm proposed by Dwyer (1987) runs in $O(n \log \log n)$ time on average, and the algorithm proposed by Katajainen and Koppinen (1988) runs in $O(n)$ time on average.

4.5 THE PLANE SWEEP METHOD

'Plane sweep' is one of the fundamental techniques used to solve two-dimensional geometric problems. In this technique we use a special line called a *sweepline*, which is conventionally vertical, and sweep the plane with this line, say from left to right. As the sweepline moves, it hits geometric objects one by one. Each time an event happens, a portion of the problem is solved

on the sweepline. Thus, this technique enables us to solve two-dimensional problems by a sequence of almost one-dimensional processing. Hence, if successfully applied, the plane sweep technique gives a conceptually simple algorithm.

However, the plane sweep technique cannot be applied to our problem directly, because to construct Voronoi edges and Voronoi vertices on the sweepline we have to 'foresee' the locations of generators in the half plane that have not yet been swept; the sweepline reaches a Voronoi polygon before hitting the corresponding generator. If each generator p_i were to exist at the leftmost point of the Voronoi polygon $V(p_i)$, which is of course not the case, the plane sweep method could be applied. Indeed, there exists a transformation by which this 'dream' is realized (Fortune, 1986, 1987).

Suppose that the set $P = \{p_1, p_2, \ldots, p_n\}$ of n generators is given. In addition to Assumptions A4.1.1, A4.1.2 and A4.4.1, here we make the following assumption.

Assumption A4.5.1 Any generator and any Voronoi vertex on its boundary do not align horizontally.

This assumption is not critical, but makes the description of the algorithm much simpler.

For any point p, we define $r(p) = \min\{d(p, p_i) \mid 1 \leq i \leq n\}$, i.e. $r(p)$ is the radius of the largest empty circle centred at p. Let $x(p)$ and $y(p)$ denote the x coordinate and the y coordinate, respectively, of p.

We define the map φ from $\mathbb{R}^2$ to itself by

$$\varphi(p) = (x(p) + r(p), y(p)) \tag{4.5.1}$$

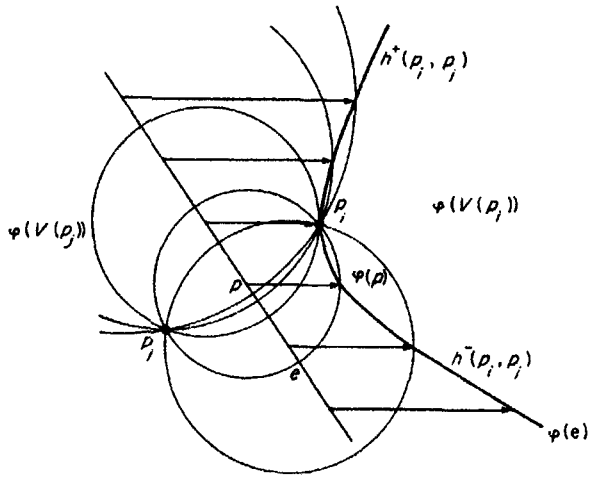

Figure 4.5.1 Perpendicular bisector and its image.

for point p (see Figure 4.5.1). If $p \in V(p_i)$, we get $\varphi(p) = (x(p) + d(p, p_i), y(p))$ because p_i is the nearest generator to p. $\varphi(p) = p$ if and only if p is a generator. For the Voronoi vertex q, the Voronoi edge e, the Voronoi polygon $V(p_i)$ and the Voronoi diagram $\mathcal{V}$, their images using φ are denoted by $\varphi(q)$, $\varphi(e)$, $\varphi(V(p_i))$ and $\varphi(\mathcal{V})$, respectively.

As shown in Figure 4.5.1, let p_i and p_j be two generators such that $x(p_i) > x(p_j)$, and let e be the Voronoi edge between p_i and p_j. Then, φ maps e to part of a curved line whose leftmost point is p_i. Indeed, it is an easy exercise to see that the image $\varphi(e)$ of e is part of a hyperbola open rightward with the leftmost point p_i. The image of each Voronoi polygon is a connected region surrounded by segments of hyperbolas.

The mapping φ can be interpreted in the context of 'Voronoi diagrams in a river' where the boat runs at the same speed as the water flow (Sugihara, 1992a). Suppose that the generators are point-like islands in a river, and each island has a boat of the same speed. Suppose that $p \in V(p_i)$. If there is no flow of water, the boat starting at island p_i reaches p faster than any other boat. Now assume that the water in the river flows from left to right (i.e. in the positive x direction) at the same speed as the boat. Then, during the sailing of the boat from p_i to p, the boat is carried to the right by the same distance as it sails; hence the boat reaches the point $\varphi(p) = (x(p) + r(p), y(p))$ instead of the point p. Thus, $\varphi(p)$ represents the point reached by the boat that leaves the island nearest to p facing toward p. Hence, $\varphi(V)$ can be interpreted as the tessellation on the surface of the river into 'territories' of the islands such that any point in a territory can be reached by the boat starting at the corresponding island faster than any other boat (see Section 3.7.5).

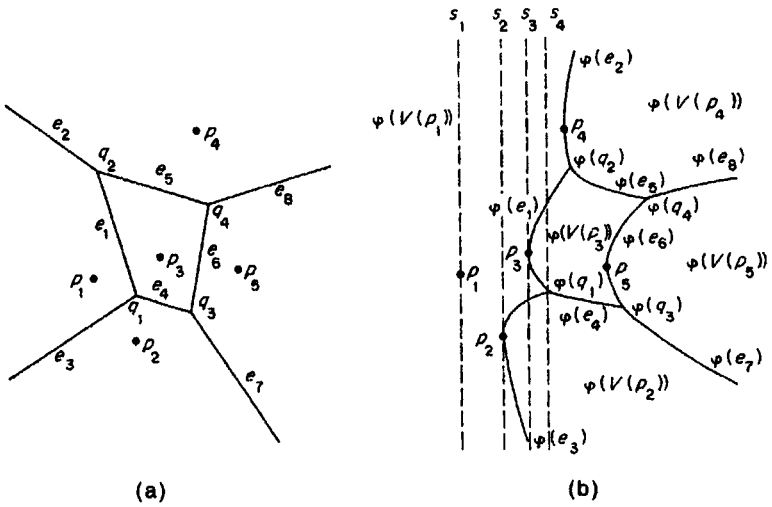

Figure 4.5.2 Voronoi diagram and its image: (a) Voronoi diagram $\mathcal{V}$ for five generators; (b) its image $\varphi(\mathcal{V})$.

An example of the map φ applied to the Voronoi diagram is shown in Figure 4.5.2, where (a) is the Voronoi diagram for five points and (b) is its image by φ. (If we want to interpret diagram (b) as the tessellation on the surface of the river, the vertical line passing through p_1, the broken line s_1, is also necessary, because no boat can reach the region to the left of s_1.)

Note that for any generator p_i other than the leftmost one, the point p_i is the farthest to the left in the transformed region $\varphi(V(p_i))$. The images of all the Voronoi edges and Voronoi vertices related to p_i appear to the right of p_i. Hence we can apply the plane sweep technique for constructing $\varphi(\mathcal{V})$.

In what follows we use the terminology for Voronoi diagrams to designate objects in $\varphi(\mathcal{V})$. For instance, the image of a Voronoi vertex is called a Voronoi vertex of $\varphi(\mathcal{V})$, and the image of a Voronoi edge is called a Voronoi edge of $\varphi(\mathcal{V})$.

Let us note at this point that, because of Assumption A4.5.1, the image $\varphi(q)$ of any Voronoi vertex q is not the leftmost point of any Voronoi region in $\varphi(\mathcal{V})$, and consequently at $\varphi(q)$ exactly one Voronoi edge extends to the right and the other two Voronoi edges extend to the left. For instance, at the vertex $\varphi(q_1)$ in Figure 4.5.2(b), the edge $\varphi(e_4)$ extends to the right and the edges $\varphi(e_1)$ and $\varphi(e_3)$ extend to the left.

The plane sweep method moves a vertical sweepline across the plane from left to right, and each time an event happens, the structure of the Voronoi diagram $\varphi(\mathcal{V})$ is updated on the sweepline. There are two types of events. An event of the first type occurs when the sweepline hits a generator, as the lines s_1, s_2 and s_3 in Figure 4.5.2(b). An event of the second type occurs when the sweepline hits a Voronoi vertex of $\varphi(\mathcal{V})$, as the line s_4 in Figure 4.5.2(b).

Let p_i and p_j be two generators. The perpendicular bisector of p_i and p_j is transformed by φ to a hyperbola. Let us cut this hyperbola at the leftmost point, and let $h^+(p_i, p_j)$ and $h^-(p_i, p_j)$ be the upper part and the lower part, respectively, of the hyperbola; we call them the upper and lower half hyperbolas (see Figure 4.5.1). Note that $h^+(p_i, p_j) = h^+(p_j, p_i)$ and $h^-(p_i, p_j) = h^-(p_j, p_i)$.

To represent the structure of $\varphi(\mathcal{V})$ along the sweepline, we use an alternating list L of regions and boundary edges which appear on the sweepline from the bottom to the top. We also use Q, which is a set of points in the plane. Q represents the potential points at which events may happen in future. At the beginning, Q is the set of all generators, and candidates of Voronoi vertices are added to Q whenever they are found. The algorithm is as follows (the example that follows the algorithm may help the reader to understand the algorithm).

Algorithm 4.5.1 (Plane sweep method)
Input: Set $P = \{p_1, p_2, \ldots, p_n\}$ of n generators.
Output: Voronoi diagram $\varphi(\mathcal{V})$.
Procedure:
Step 1. $Q \leftarrow P$.
Step 2. Choose and delete the leftmost point, say p_i, from Q.

Step 3. $L \leftarrow$ the list consisting of a single region $\varphi(V(p_i))$.

Step 4. While Q is not empty, repeat 4.1, 4.2 and 4.3.

4.1. Choose and delete the leftmost point w from Q.

4.2. If w is a generator, say $w = p_i$, do 4.2.1, 4.2.2 and 4.2.3.

4.2.1. Find region $\varphi(V(p_j))$ on L containing p_i.

4.2.2. Replace $\varphi(V(p_j))$ on L by the subsequence $(\varphi(V(p_j)), h^-(p_j, p_i), \varphi(V(p_i)), h^+(p_i, p_j), \varphi(V(p_j)))$.

4.2.3. Add to Q the intersection of $h^-(p_j, p_i)$ with the immediate lower half hyperbola on L, and the intersection of $h^+(p_i, p_j)$ with the immediate upper half hyperbola on L.

4.3. If w is an intersection, say $w = \varphi(q_t)$, do 4.3.1, . . . , 4.3.4 (where we suppose that $\varphi(q_t)$ is the intersection of $h^{\pm}(p_i, p_j)$ and $h^{\pm}(p_j, p_k)$).

4.3.1. Replace subsequence $(h^{\pm}(p_i, p_j), \varphi(V(p_j)), h^{\pm}(p_j, p_k))$ on L by $h = h^-(p_i, p_k)$ or $h = h^+(p_i, p_k)$ appropriately.

4.3.2. Delete from Q any intersections of $h^{\pm}(p_i, p_j)$ or $h^{\pm}(p_j, p_k)$ with others.

4.3.3. Add to Q any intersections of h with its immediate upper half hyperbola and its immediate lower half hyperbola on L.

4.3.4. Mark $\varphi(q_t)$ as a Voronoi vertex incident to $h^{\pm}(p_i, p_j)$, $h^{\pm}(p_j, p_k)$ and h.

Step 5. Report all the half hyperbolas ever listed on L, all the Voronoi vertices marked in Step 4.3.4 and the incidence relations among them.

Steps 1, 2 and 3 are for initialization. In the example in Figure 4.5.2, Q is set to be $\{p_1, p_2, \ldots, p_5\}$ in Step 1, $p_i = p_1$ is chosen and Q is changed to $\{p_2, \ldots, p_5\}$ in Step 2, and L is set to be

$$L = (\varphi(V(p_1)))$$

in Step 3. Steps 2 and 3 correspond to the event that the sweepline hits the leftmost generator (the vertical line s_1 in the figure). The list $L = (\varphi(V(p_1)))$ implies that if we move along the sweepline from the bottom upward, we always lie in the Voronoi region for p_1.

Step 4 is the main part of the algorithm. In Step 4.1 we choose a point at which the next event occurs. In the example shown in Figure 4.5.2, the algorithm goes in the following way. We have $Q = \{p_2, \ldots, p_5\}$ at the end of Step 3, and hence at Step 4.1 point $w = p_2$ is chosen from Q and Q is changed to $Q = \{p_3, p_4, p_5\}$. This corresponds to the event that the sweepline hits p_2, the sweepline being at s_2 in Figure 4.5.2(b).

According to the type of the event, Step 4.2 or 4.3 is done; Step 4.2 is for an event of the first type and Step 4.3 is for an event of the second type. Since $w = p_2$ is a generator, Step 4.2 is done. First, we find the region $\varphi(V(p_j))$ containing the new generator p_i in Step 4.2.1; next we create a new Voronoi

region $\varphi(V(p_i))$ and a new Voronoi edge consisting of $h^-(p_i, p_j)$ and $h^+(p_j, p_i)$ in Step 4.2.2; and finally we insert candidates of Voronoi vertices which may be end points of $h^-(p_i, p_j)$ and $h^+(p_i, p_j)$. In our example, in Step 4.2.1 $\varphi(V(p_j))$ is recognized as $\varphi(V(p_1))$, and in Step 4.2.2 L is changed to

$$L = \big(\varphi(V(p_1)), h^-(p_1,p_2), \varphi(V(p_2)), h^+(p_2,p_1), \varphi(V(p_1))\big).$$

The list L at this point implies that, if we move along the sweepline from the bottom upward, we traverse the region $\varphi(V(p_1))$, cross the edge $h^-(p_1, p_2)$, traverse the region $\varphi(V(p_2))$, cross the edge $h^+(p_2, p_1)$ and traverse the region $\varphi(V(p_1))$ in this order. This is the situation that we have immediately after the sweepline hits p_2. Since there is no half hyperbola below $h^-(p_1, p_2)$ or above $h^+(p_2, p_1)$, nothing is done in Step 4.2.3.

In the second repeat of Step 4.1, $w = p_3$ is chosen and Q is changed to $Q = \{p_4, p_5\}$. This corresponds to the event that the sweepline hits p_3 (the line s_3 in Figure 4.5.2(b)). Hence, Step 4.2 is done again.

The generator p_3 lies above the upper hyperbola $h^+(p_1, p_2)$, and hence in Step 4.2.2 the second appearance of $\varphi(V(p_1))$ in L is replaced by the sublist $(\varphi(V(p_1)), h^-(p_1, p_3), \varphi(V(p_3)), h^+(p_3, p_1), \varphi(V(p_1)))$, which changes L to

$$L = \big(\varphi(V(p_1)), h^-(p_1,p_2), \varphi(V(p_2)), h^+(p_2,p_1),$$
$$\varphi(V(p_1)), h^-(p_1,p_3), \varphi(V(p_3)), h^+(p_3,p_1), \varphi(V(p_1))\big).$$

The new list L represents the situation obtained immediately after the sweepline hits p_3. In L there is a half hyperbola $h^+(p_2, p_1)$, which is immediately below $h^-(p_1, p_3)$, and consequently in Step 4.2.3 the point $\varphi(q_1)$ of intersection of these two half hyperbolas is added to Q; thus Q is changed to $Q = \{\varphi(q_1), p_4, p_5\}$.

In the third repeat of Step 4.1, $w = \varphi(q_1)$ is chosen and Q is changed to $Q = \{p_4, p_5\}$. This corresponds to the event that the sweepline comes to s_4 in Figure 4.5.2(b). Since $w = \varphi(q_1)$ is a Voronoi vertex, Steps 4.3.1–4.3.4 are done; throughout this step we should take the same superscript + or − for all appearances of $h^\pm(p_i, p_j)$ as we should do for all appearances of $h^\pm(p_j, p_k)$. In Step 4.3.1 the old region $\varphi(V(p_j))$ and its two boundaries $h^\pm(p_i, p_j)$ and $h^\pm(p_j, p_k)$ are deleted from L, and a new Voronoi edge h is inserted into L, where h is either $h^-(p_i, p_k)$ or $h^+(p_i, p_k)$, depending on the configuration around. At this point, $\varphi(q_1)$ is recognized as the end point of the edges $h^\pm(p_i, p_j)$ and $h^\pm(p_j, p_k)$. All the other candidates for the end points of them are deleted from Q in Step 4.3.2, and the candidates of the end point of the new edge h are added to Q in Step 4.3.3. In Step 4.3.4 the incidence structure around $\varphi(q_1)$ is recorded.

In our example, in Step 4.3.1 the subsequence $(h^+(p_2, p_1), \varphi(V(p_1)), h^-(p_1, p_3))$ is replaced by $h^-(p_2, p_3)$, changing L to

$$L = \big(\varphi(V(p_1)), h^-(p_1,p_2), \varphi(V(p_2)), h^-(p_2,p_3),$$
$$\varphi(V(p_3)), h^+(p_3,p_1), \varphi(V(p_1))\big).$$

Nothing is done in Step 4.3.2 because Q contains no point of intersection of two half hyperbolas. Nothing is done in Step 4.3.3, either, because the half hyperbola $h^-(p_2, p_3)$ has no point of intersection with $h^-(p_1, p_2)$ or with $h^+(p_3, p_1)$. In Step 4.3.4 the incidence relations among the Voronoi vertices, the Voronoi edges and the Voronoi regions around $\varphi(q_1)$ are registered.

The algorithm treats the other events one by one in a similar manner until Q becomes empty. The Voronoi diagram $\varphi(\mathcal{V}(P))$ is obtained by collecting all the half hyperbolas ever listed in L, all the Voronoi vertices marked in Step 4.3.4 and the incidence relations among them.

We can understand the correctness of the algorithm by noting that (i) the generator p_i, other than the leftmost one, lies at the leftmost point of the Voronoi region $\varphi(V(p_i))$; (ii) at each Voronoi vertex in $\varphi(\mathcal{V})$ exactly one edge emanates to the right and the other two edges to the left; and that (iii) for each Voronoi vertex in $\varphi(\mathcal{V})$, the two edges emanating to the left become neighbours on L at some time during the sweep (and hence the points inserted in Steps 4.2.3 and 4.3.3 include all the Voronoi vertices). See Fortune (1986, 1987) for the details of the proof.

To execute the procedure efficiently, we need some techniques in data structure. The basic operations on Q are the insertion of points and the deletion of the minimum point with respect to the x coordinate. A storage requiring these operations is called a *priority queue*, and can be implemented, for instance, by a data structure called a heap. Indeed, using a heap, we can carry out each of the above operations in $O(\log n)$ time, where n is the number of points stored in Q (Aho *et al.*, 1974). In Step 4.3.2 we also have to delete from Q some points that are not necessarily minimum in the x coordinate. However, these points can be accessed in constant time if we store and maintain pointers at the associated half hyperbolas, and can be deleted in $O(\log n)$ time in the same manner as the insertion and the minimum deletion.

The other important storage L is an alternating list of regions and half hyperbolas along the sweepline. The intersections of the regions with the sweepline give intervals on the sweepline, and the intersections of the half hyperbolas with the sweepline give boundary points of the intervals. The interval corresponding to each region $\varphi(V(p_i))$ changes as the sweepline moves. The basic operations in L are to search for a relevant interval in Step 4.2.1 and the deletion and insertion of substrings in Steps 4.2.2 and 4.3.1. A storage requiring these operations is called a *dictionary*, and is implemented efficiently by a data structure such as a 2–3 tree or an AVL tree (Aho *et al.*, 1974; AVL stands for the initials of the coauthors of the paper). With such a data structure, we can execute each operation in $O(\log n)$ time.

Employing those techniques in data structure, we can execute Algorithm 4.5.1 in $O(n \log n)$ time. Indeed, Step 1 requires $O(n \log n)$ time, Step 2 $O(\log n)$ time and Step 3 constant time. Note that the total number of elements ever listed on L is of $O(n)$; this is because the number of Voronoi edges and Voronoi vertices are both of $O(n)$. The total number of elements ever stored in Q is also of $O(n)$, because there are only $O(n)$ pairs of Voronoi

edges that ever become neighbours on L and hence there are only $O(n)$ candidates for Voronoi vertices. Therefore, Steps 4.1, 4.2 and 4.3 require $O(\log n)$ time, and they are repeated $O(n)$ times in Step 4. Thus, the total time complexity is of $O(n \log n)$.

The plane sweep technique has been applied to many geometric problems, such as point location, the extraction of intersections between line segments or between polygons and the construction of an arrangement consisting of lines (Preparata and Shamos, 1985). Fortune (1986) was the first to apply this technique to the construction of Voronoi diagrams and obtained an algorithm equivalent to Algorithm 4.5.1. The plane sweep technique was also applied to a Voronoi diagram for line segments (Fortune, 1986, 1987; Section 3.5), to an additively weighted Voronoi diagram (Fortune, 1986, 1987; Section 3.1.2) and to a Voronoi diagram on a cone (Dehne and Klein, 1987; Section 3.7.8). Different types of sweep procedure are also used for the ordinary Voronoi diagrams (Dehne and Klein, 1987, 1988, 1997).

4.6 PRACTICAL TECHNIQUES FOR IMPLEMENTING THE ALGORITHMS

4.6.1 Inconsistency caused by numerical errors

So far we have assumed that neither degeneracy nor numerical error takes place in computation. However, algorithms designed in such an artificial world are not necessarily valid in practice, because in computers numerical values are treated only in finite-precision arithmetic. In degenerate or nearly degenerate cases, numerical error sometimes causes inconsistency in topological structures, and thus makes algorithms invalid.

An example of topological inconsistency we often encounter in the incremental method is shown in Figure 4.6.1. The boundary-growing procedure employed in the incremental method is based on the property that the boundary of a Voronoi polygon forms a closed sequence of segments of perpendicular bisectors. However, if the bisectors pass near to a Voronoi vertex, the topological relations between the boundaries and the Voronoi vertex are sometimes misjudged, resulting in a sequence in which the end point does not coincide with the start point, as shown in Figure 4.6.1. If such an inconsistency happens, the incremental algorithm cannot accomplish the task.

In this section, we remove all the assumptions, Assumptions A4.1.1, A4.1.2, A4.4.1 and A4.5.1, which we made for the sake of simplicity, and consider how to translate the incremental method into computer programs that work for any input data, degenerate or non-degenerate, in finite-precision computation. We present two quite different principles. The first one, presented in Section 4.6.2, is for constructing an error-free closed world in which all the topological structures are judged correctly. In this world, degeneracy is also detected exactly, so that we have another problem, i.e. the problem of

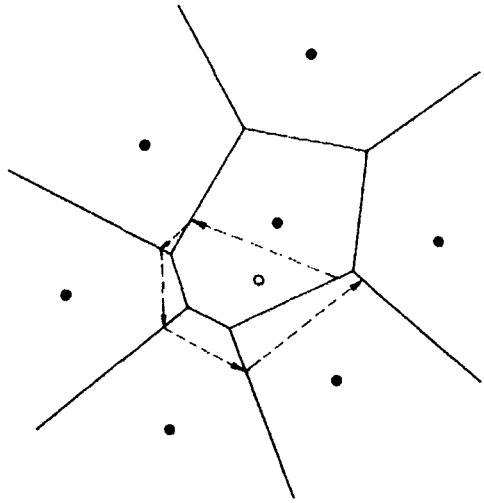

Figure 4.6.1 Unclosed boundary caused by numerical errors.

preparing complicated branches of exceptional procedure for treating degenerate cases. To circumvent this problem, we also present a method for avoiding degeneracy by symbolic perturbation. In Section 4.6.3, on the other hand, we consider a world with numerical errors and present the second principle, in which we avoid inconsistency by placing higher priority on topological structures than on numerical values. This principle automatically enables us to avoid degeneracy because it is impossible to detect degeneracy by computation with errors.

4.6.2 Construction of an error-free world

The first principle is to compute numerical values so precisely that they are good enough always to judge topological structures correctly.

Suppose that we fix the set $P = \{p_1, \ldots, p_n\}$ of generators. Let (x_i, y_i) denote the coordinates of the generator p_i. Let us assume that the coordinates are represented by integers whose absolute values are not greater than a certain fixed integer K:

$$-K \leq x_i, y_i \leq K \quad \text{for } i = 1, 2, \ldots, n. \tag{4.6.1}$$

This assumption is not unrealistic, because in a digital computer the coordinates are usually represented by a fixed number of bits, so that they can be transformed into integers if a sufficiently large-scale factor is multiplied.

Suppose that we have constructed a Voronoi diagram $\mathcal{V}_{l-1}$ for the first $l-1$ generators $p_1, p_2, \ldots, p_{l-1}$ and now want to add a new generator p_l. For this purpose we need to find Voronoi vertices of $\mathcal{V}_{l-1}$ that should be contained in the Voronoi polygon of the new generator.

Let q_{ijk} denote the Voronoi vertex incident to three Voronoi polygons $V(p_i)$, $V(p_j)$ and $V(p_k)$ counterclockwise in this order. For three generators p_i, p_j, p_k and a point $p = (x, y)$, let $H(p_i, p_j, p_k, p)$ be as defined by equation (2.4.10). The equation $H(p_i, p_j, p_k, p) = 0$ represents the circle that passes through p_i, p_j and p_k, and this circle contains p if and only if $H(p_i, p_j, p_k, p) < 0$. Recall that q_{ijk} belongs to the Voronoi polygon of the new generator p_l if and only if p_l is contained in this circle, which means that q_{ijk} belongs to $V(p_l)$ if and only if $H(p_i, p_j, p_k, p_l) < 0$.

Hence, what we have to do to find the points $w_1, w_2, \ldots, w_m$ in Step 4.3 of Algorithm 4.3.2 is to find Voronoi edges having $H(p_i, p_j, p_k, p_l)$ with opposite signs at their two end points.

Note that $H(p_i, p_j, p_k, p_l)$ is an integer, so that it can be computed correctly unless it overflows memory. From the Hadamard inequality (which says that the absolute value of the determinant of a matrix is not greater than the multiplication of the lengths of all the column vectors; see, for instance, Greub, 1975) and the assumption (4.6.1), we obtain

$$\begin{aligned} &|H(p_i,p_j,p_k,p_l)| \\ &\quad \le \sqrt{1^2+1^2+1^2+1^2}\sqrt{x_i^2+x_j^2+x_k^2+x_l^2}\sqrt{y_i^2+y_j^2+y_k^2+y_l^2} \\ &\qquad \times \sqrt{(x_i^2+y_i^2)^2+(x_j^2+y_j^2)^2+(x_k^2+y_k^2)^2+(x_l^2+y_l^2)^2} \qquad (4.6.2) \\ &\quad \le 2\cdot 2K\cdot 2K\cdot 4k^2 \\ &\quad = 32K^4. \end{aligned}$$

If we compute the value of $H(p_i, p_j, p_k, p_l)$ according to the Laplace expansion of the right-hand side of equation (4.6.2), the absolute value of any number that appears in the course of computation does not exceed the bound $32K^4$, either. Hence the integer $H(p_i, p_j, p_k, p_l)$ can be computed without overflow if the computation is carried out so precisely that a number as large as $32K^4$ is representable.

Let $N = [\log_2 (K + 1)]$, where $[x]$ is the smallest integer not less than x. Then each coordinate can be represented by $N+1$ bits, N bits for the absolute value and 1 bit for the sign. Any integer between $-32K^4$ and $32K^4$ can be represented by $4N+6$ bits, $4N+5$ bits for the absolute value and 1 bit for the sign. Hence, using $4N+6$ bits for computation, we can always correctly determine the Voronoi vertices to be deleted in the addition of the new generator. Thus we can avoid misjudgement in the topological structure of the Voronoi diagram.

For example, if we use 64 bits (2 words in an ordinary computer) to represent each integer appearing in the computation (i.e. 1 bit for the sign and 63 bits for the absolute value), we can represent integers as large as $M = 2^{63} - 1$. In this case, consequently, we can choose $K = 23\,170$, because

$$M = 2^{63} - 1 = 2^5 \times 2^{58} - 1 = 32(2^{14.5})^4 - 1 > 32(23170)^4,$$

which seems to give sufficient resolution for ordinary purposes in point pattern analysis.

By the above method we can indeed avoid misjudgement in topological structures, but we still have another problem, i.e. the treatment of degenerate cases. Degeneracy takes place when $H(p_i, p_j, p_k, p_l) = 0$. In this case the boundary of the Voronoi polygon of the new generator passes through the Voronoi vertex of $\mathcal{V}_{l-1}$, giving a Voronoi vertex with four or more edges. Such degeneracy usually requires exceptional branches of processing, which makes construction and maintenance of a computer program difficult.

Fortunately, however, we can avoid degeneracy by a very simple trick; intuitively, we 'shrink' the Voronoi polygon of the new generator infinitesimally and thus avoid creating degenerate Voronoi vertices.

From a topological point of view, the Voronoi diagram can be represented by an augmented geometric graph, a planar graph embedded in the plane; recall Section 4.2. Let G_i denote the augmented geometric graph associated with the Voronoi diagram $\mathcal{V}_i$ for the first i generators. G_i represents the topological structure of $\mathcal{V}_i$. If no degeneracy takes place, the degree of any vertex (i.e. the number of edges incident to the vertex) in G_i is three. In the case of degeneracy, on the other hand, G_i has vertices with degree four or more.

To represent G_i, we adopt the convention that a vertex with degree four or more, if any, is replaced with a microstructure consisting of degree-three vertices and 'length-zero' edges. For example, if four generators p_i, p_j, p_k and p_l are on a common circle and give a degenerate Voronoi vertex with degree four, as shown in Figure 4.6.2(a), we generate two degree-three vertices and one length-zero edge, as shown in (b) or (c); we can choose either of the two alternative structures. This convention implies that we need not generate a vertex of degree more than three even if degeneracy takes place.

Our idea to avoid exceptional processing for degeneracy is just to treat the case where $H(p_i, p_j, p_k, p_l) = 0$ as if $H(p_i, p_j, p_k, p_l) > 0$. That is, we judge

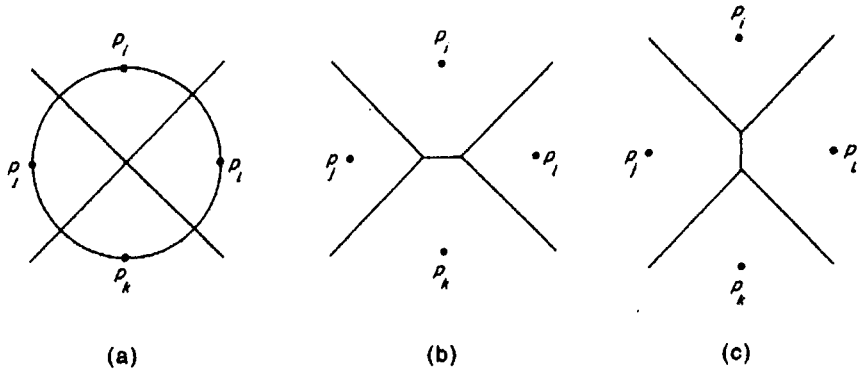

Figure 4.6.2 Degeneracy and perturbation: (a) four generators on a common circle; (b) introduction of a length-zero edge; (c) another way of introducing a length-zero edge.

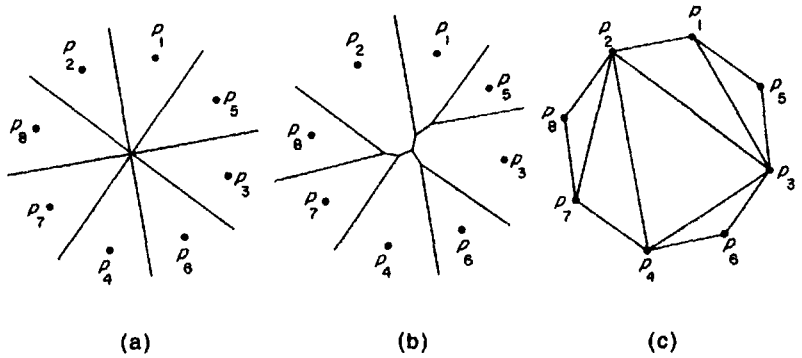

Figure 4.6.3 Degeneracy avoidance: (a) degenerate vertex of degree eight; (b) result of avoiding degeneracy; (c) the corresponding Delaunay triangulation.

that q_{ijk} is inside $V(p_l)$ if $H(p_i, p_j, p_k, p_l) < 0$, and outside if $H(p_i, p_j, p_k, p_l) \geq 0$; note that the latter inequality allows the equality. This strategy intuitively means that each time degeneracy takes place, we shrink the Voronoi polygon of the new generator only slightly so as not to disturb other parts of the Voronoi diagram. Hence every Voronoi vertex of G_{l-1} is judged either inside $V(p_l)$ or outside; no vertex is 'on' the boundary of $V(p_l)$.

Figure 4.6.3 shows an example of how our method avoids degeneracy. Here we assume that eight generators $p_1, p_2, \ldots, p_8$ lie on a common circle, so that in an ordinary sense a Voronoi vertex of degree eight will appear, as shown in (a). Suppose that the generators $p_1, p_2, \ldots, p_8$ are added in this order. Then the resultant topological structure will be as shown in (b), where the degenerate vertex is replaced by a tree structure consisting of six vertices and five length-zero edges. This structure is the result of our strategy. This may be understood more easily when we consider the Delaunay triangulation, dual to the Voronoi diagram, shown in (c). Since the primal diagram (b) consists of degree-three vertices, the dual diagram (c) is composed of triangles. These triangles are generated in such a way that every time a new generator is added, it is connected to the old generators to form a triangle. This is the interpretation of our shrink strategy in terms of the dual graph.

This strategy can avoid degeneracy because, if G_{l-1} is a geometric graph with degree three, then so is G_l. Moreover, the strategy does not disturb our original purpose in the sense that, when length-zero edges are contracted, the resulting diagram gives the correct topological structure of the Voronoi diagram.

Summing up the above considerations, we get the next principle for implementation.

Implementation Principle 4.6.1 (Error-free world) Choose and fix the upper bound K of the absolute values of the coordinates of the generators, and put $N = [\log_2 (K + 1)]$. Represent the x and y coordinates of the generators by

$N+1$ bit integers (N bits for the absolute value and 1 bit for the sign), and compute the values of $H(p_i, p_j, p_k, p_l)$ in $4N+6$ bits. Judge that the Voronoi vertex q_{ijk} is inside the Voronoi polygon of the new generator p_l if $H(p_i, p_j, p_k, p_l) < 0$, and outside if $H(p_i, p_j, p_k, p_l) \geq 0$.

This principle is based on the recognition that 'if the input data are represented by a finite number of bits, the sign of the result of an algebraic computation over the data can be determined correctly in a certain finite-precision arithmetic'. Recently this principle has gradually been recognized and utilized in several fields of computer sciences such as linear programming (Khachian, 1980), computational algebra (Lenstra, 1984) and solid modelling (Sugihara and Iri, 1989a).

The shrink strategy presented here is a kind of 'symbolic perturbation'. Symbolic perturbation is a technique for avoiding degeneracy originally used in linear programming by Bland (1977). Recently the same idea was introduced in computational geometry, and general frameworks are discussed by Edelsbrunner and Mücke (1987) and Yap (1988), by which we can obtain a non-degenerate configuration that is topologically equivalent to a configuration resulting from some perturbation of the input numerical data. The strategy presented in this section can be considered as a symbolic perturbation in the context of the Voronoi diagram with the Laguerre metric (Sugihara, 1992b). The lift-up method was also implemented using the exact arithmetic and symbolic perturbation (Sugihara, 1997).

4.6.3 Topology-oriented approach

We next consider what we can do if we are not allowed to compute in sufficiently high precision. In poor precision arithmetic, numerical errors are inevitable, so that judgment based on numerical values cannot be relied upon; we can rely only on combinatorial computation. Hence, it is natural to place higher priority on topological structures than on numerical values. In this subsection we treat topological structures as higher-priority information than numerical values and thus make the incremental method robust against numerical errors.

As we saw in Section 4.2, the topological structure of the Voronoi diagram $\mathcal{V}_l$ can be considered as the augmented geometric graph G_l. From a topological point of view, therefore, the main task in the incremental method is to convert G_{l-1} to G_l for $l = 4, 5, \ldots, n$. We first characterize this task in terms of combinatorial computation only.

For $i = 3,4, \ldots, n$, let us define $\boldsymbol{A}_i$ as the set of augmented geometric graphs G satisfying the following conditions.

(C4.6.1) The degree of any vertex in G is exactly three.

(C4.6.2) G divides the plane into $i+1$ regions; let us call these regions cells.

(C4.6.3) Every cell except for the outermost infinite one is simply connected, i.e. it is connected and does not have a hole.

(C4.6.4) Two cells share at most one common edge.

(C4.6.5) The outermost circuit of G (i.e. the boundary of the outermost infinite cell) consists of exactly three edges and exactly three vertices.

If the Voronoi diagram $\mathcal{V}_i$ is not degenerate, the augmented geometric graph G_i associated with $\mathcal{V}_i$ belongs to A_i. Indeed, (C4.6.1) holds first because $\mathcal{V}_i$ is non-degenerate so that all the Voronoi vertices are of degree three, and secondly because the infinite Voronoi edges were connected to the outermost closed curve (recall Section 4.2). Condition (C4.6.2) holds because $\mathcal{V}_i$ divides the plane into i Voronoi polygons and we introduced the outermost closed curve to surround the diagram. (C4.6.3) and (C4.6.4) hold because Voronoi polygons are convex. To see that (C4.6.5) holds, recall that the additional three generators p_1, p_2 and p_3 were chosen in such a way that they formed a triangle containing all the original generators $p_4, p_5, \ldots, p_n$. Hence as shown in Figures 4.3.2(a) and (b), only p_1, p_2 and p_3 have infinite Voronoi polygons; the three cells corresponding to these three Voronoi polygons only are incident to the outermost circuit of G. Thus, G_i satisfies (C4.6.5).

Let us represent the augumented geometric graph G_i by the triple $G_i = (W_i, E_i, C_i)$, where W_i, E_i and C_i are the set of vertices, edges and cells, respectively, of G_i. For any subset $T \subset W_i$, let $G_i(T)$ denote the subgraph of G_i consisting of vertices in T and those edges in E_i that connect two vertices in T. For any cell $c \in C_i$, let $W_i(c)$ denote the set of vertices on the boundary of the cell c. In the addition of a new generator p_l, the incremental method changes the topological structure of the Voronoi diagram from $G_{l-1} = (W_{l-1}, E_{l-1}, C_{l-1})$ to $G_l = (W_l, E_l, C_l)$. As shown by an example in Figure 4.6.4(a), this task can be done by first choosing a set T of vertices to be deleted (the vertices represented by filled circles in the figure), then creating new vertices (represented by unfilled circles) on all edges connecting a vertex in T and a vertex not in T, and new edges (represented by broken lines) forming a circuit enclosing T, and finally deleting the substructure inside the circuit.

Let $T (\subset W_{l-1})$ be the set of vertices that is deleted in the change from G_{l-1} to G_l. Then T should satisfy the following conditions.

(C4.6.6) T is non-empty.

(C4.6.7) T does not contain a vertex on the outermost circuit of G_{l-1}.

(C4.6.8) $G_{l-1}(T)$ is a tree, i.e. a connected acyclic graph.

(C4.6.9) For any $c \in C_{l-1}$, $G_{l-1}(T \cap W_{l-1}(c))$ is connected.

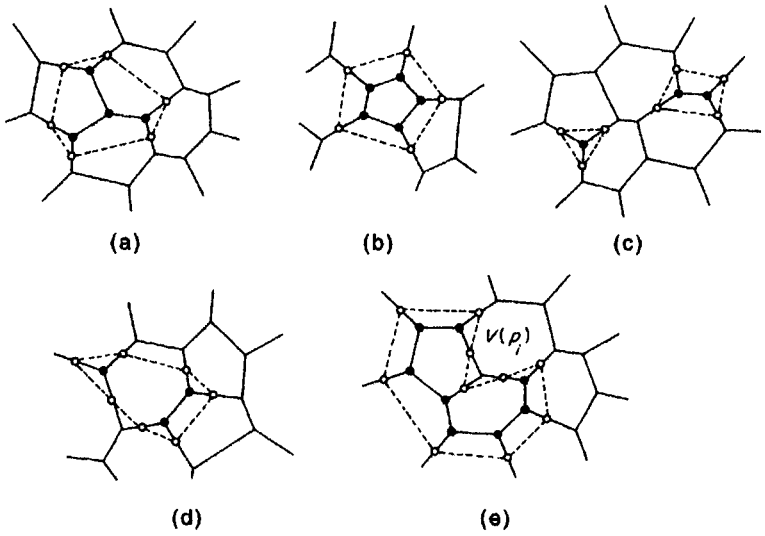

Figure 4.6.4 Topological aspect of the incremental method: (a) deletion of a tree; (b) deletion of a cycle; (c) deletion of two or more connected components; (d) another way of deleting two or more components; (e) deletion of a subgraph that violates condition (C4.6.9).

Condition (C4.6.6) is obvious because the new generator p_l should have its own non-empty Voronoi polygon. Condition (C4.6.7) holds because we introduced the three additional generators and hence the Voronoi polygon of p_l ($l = 4, 5, \ldots, n$) is finite. If $G_{l-1}(T)$ has a circuit, then the Voronoi region of an old generator is removed entirely, as shown in Figure 4.6.4(b), which contradicts the basic property that every generator has its own non-empty region. If $G_{l-1}(T)$ is disconnected, then either the Voronoi region of p_l has two or more components, as shown in (c), or an old Voronoi region has two or more components, as shown in (d). Hence T satisfies (C4.6.8). If $G_{l-1}(T \cap W_{l-1}(c))$ is not connected as in (e) (where c corresponds to the boundary of Voronoi region $V(p_i)$), then the Voronoi regions $V(p_i)$ and $V(p_l)$ have two or more edges in common, which contradicts condition (C4.6.4). Thus, (C4.6.9) is satisfied.

Note that conditions (C4.6.6)–(C4.6.9) are written in terms of combinatorial properties only. Hence we can check the conditions by combinatorial computation; we need not employ numerical computation. Moreover, it is easy to see that if G_{l-1} belongs to $\boldsymbol{A}_{l-1}$ and T satisfies (C4.6.6)–(C4.6.9), then the augmented geometric graph G_l belongs to $\boldsymbol{A}_l$.

However, there is freedom in the choice of the subset T. Hence we employ numerical computation to choose the most promising subset T, and for this purpose only. Thus we obtain the next algorithm for changing G_{l-1} to G_l, which should be used in place of the boundary-growing procedure.

Algorithm 4.6.1 (Tree expansion and deletion)
Input: Generators $p_1, p_2, \ldots, p_l$ and geometric graph G_{l-1} ($\in \boldsymbol{A}_{l-1}$).
Output: Geometric graph G_l ($\in \boldsymbol{A}_l$).
Procedure:
Step 1. Find the generator p_i ($1 \leq i \leq l-1$) such that $d(p_i, p_l)$ is minimum.
Step 2. Among the Voronoi vertices q_{ijk}'s on the boundary of $V(p_i)$, find the one that gives the smallest value of $H(p_i, p_j, p_k, p_l)$. Let T be the set consisting of this Voronoi vertex alone.
Step 3. Repeat 3.1 until T cannot be augmented any more.
  3.1. For each Voronoi vertex q_{ijk} that is connected by a Voronoi edge to an element of T, add q_{ijk} to T both if $H(p_i, p_j, p_k, p_l) < 0$ and if the resultant T satisfies conditions (C4.6.7)–(C4.6.9).
Step 4. For every edge connecting a vertex in T with a vertex not in T, create a new vertex on the edge and thus divide the edge into two edges.
Step 5. Create new edges connecting the vertices created in Step 4 in such a way that the new edges form a circuit enclosing the vertices in T.
Step 6. Remove the vertices in T and the edges incident to them (and consider the interior of the circuit as the Voronoi polygon of p_l), and let the resultant geometric graph be G_l.

This algorithm corresponds to Steps 4.1–4.4 in Algorithm 4.3.2. More specifically, Step 1 in this algorithm is equivalent to Step 4.1 in Algorithm 4.3.2, Steps 2, 3 and 4 correspond to Steps 4.2 and 4.3, and Steps 5 and 6 correspond to Step 4.4. Hence if there is no numerical error, and if G_{l-1} corresponds to the Voronoi diagram $\mathcal{V}_{l-1}$, the output G_l correctly corresponds to the Voronoi diagram $\mathcal{V}_l$. The main difference is that in this algorithm the initial vertex found in Step 2 is used as a 'seed' from which a tree is expanded, while in the boundary-growing procedure the boundary enclosing this tree is traced. This difference is essential, because even if computed values of $H(p_i, p_j, p_k, p_l)$ contain numerical errors, the subset T obtained in the algorithm satisfies (C4.6.6)–(C4.6.9). We need not worry about inconsistency; the algorithm always carries out its task and the output G_l is consistent in the sense that it belongs to $\boldsymbol{A}_l$.

Note that degeneracy can be avoided automatically in the algorithm. We judge the Voronoi vertex q_{ijk} as being inside the Voronoi polygon of the new generator if and only if $H(p_i, p_j, p_k, p_l) < 0$. This is equivalent to the shrink strategy presented in the previous subsection.

The conditions (C4.6.8) and (C4.6.9) can be checked efficiently in the following manner. We assign three labels, 'in', 'out' and 'undecided', to the Voronoi vertices; 'in' is assigned to the Voronoi vertices that are added to T, 'out' to the Voronoi vertices that are judged not to be added to T, and 'undecided' to the Voronoi vertices that are not yet judged. Also, we assign two labels 'incident' and 'non-incident' to the Voronoi polygons; 'incident' to the Voronoi polygons whose boundary has a vertex in T, and 'non-incident'

to the other Voronoi polygons. In Step 3.1, q_{ijk} is chosen among the Voronoi vertices that are labelled 'undecided' and that are adjacent to a vertex in T. Then, $G_{l-1}(T \cup \{q_{ijk}\})$ is a tree if and only if

(C4.6.10) q_{ijk} is not adjacent to two or more 'in' Voronoi vertices.

For any cell c of G_{l-1}, the subgraph $G_{l-1}((T \cup \{q_{ijk}\}) \cap W_{l-1}(c))$ is connected if and only if

(C4.6.11) for any 'incident' Voronoi polygon having q_{ijk} on its boundary, q_{ijk} is adjacent to an 'in' Voronoi vertex on this boundary.

Hence conditions (C4.6.8) and (C4.6.9) can be examined locally by checking the labels of the Voronoi polygons incident to q_{ijk} and those of the Voronoi vertices adjacent to q_{ijk}.

For the initial geometric graph G_3, we assign 'out' labels to the three degree-one vertices, and these labels are permanently fixed. This is because the Voronoi polygon of a new generator is always finite, so that these three vertices should not belong to T. At the start of Algorithm 4.6.1 all the other Voronoi vertices are labelled 'undecided', and all the Voronoi polygons are labelled 'non-incident'. Each time a Voronoi vertex is added to T, its label is changed from 'undecided' to 'in' and the labels of the Voronoi polygons incident to the Voronoi vertex, if they are 'non-incident', are changed to 'incident'. Similarly, each time a Voronoi vertex is judged not to be added to T, its label is changed from 'undecided' to 'out'. In Algorithm 4.6.1 the Voronoi vertices and the Voronoi polygons whose labels are changed are listed, and at the end of the algorithm these labels are cleared (that is, 'in' and 'out' are changed to 'undecided', and 'incident' is changed to 'non-incident'). In this way Algorithm 4.6.1 can be executed in time proportional to the size of the substructure that is deleted for the addition of the new generator.

Thus, we obtain the next principle for implementing the incremental method.

Implementation Principle 4.6.2 (Topology-oriented approach)
We concentrate our attention on the construction of the topological structure G_n of the Voronoi diagram (that is, we replace $\mathcal{V}, \mathcal{V}_3, \mathcal{V}_{l-1}, \mathcal{V}_l$ and $\mathcal{V}_n$ in Algorithm 4.3.2 with G, G_3, G_{l-1}, G_l and G_n, respectively), and replace Steps 4.1–4.4 in Algorithm 4.3.2 with Algorithm 4.6.1.

Once G_n is obtained, it is easy to draw the Voronoi diagram itself, because a Voronoi vertex is the centre of the circle passing through the three surrounding generators, and a Voronoi edge is the perpendicular bisector of the two side generators.

An example of the behaviour of the computer program constructed on the basis of this principle is shown in Figure 4.6.5, where (a) is the output constructed in single-precision floating point arithmetic for 20 generators

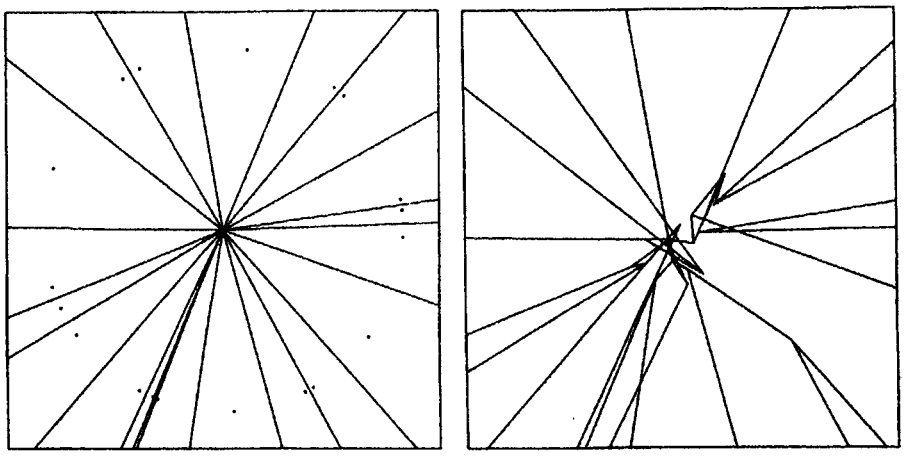

Figure 4.6.5 Behaviour of a computer program based on the topology-oriented approach: (a) output of the program for 20 generators; (b) central portion of (a) magnified by 10^5.

placed at random on a common circle, and (b) represents the central portion of the same output magnified by 10^5. The diagram in (a) seems a correct Voronoi diagram but, as (b) shows, the central portion of the output is far from the correct Voronoi diagram. This set of generators gives high degeneracy, and the criss-cross microstructure of this kind seems natural in single precision arithmetic. It should be noted that even for such a degenerate set of generators, the program carried out its task and gave an output, and that the output is topologically consistent in the sense that it belongs to A_n. A conventional computer program usually fails in processing such a degenerate set of generators.

We place higher priority on topological consistency than on numerical values, but this does not mean that numerical computation is less important. In our algorithm the most important numerical computation is that of $H(p_i, p_j, p_k, p_l)$ defined by (2.4.10). $H(p_i, p_j, p_k, p_l)$ can be rewritten in the following way:

$$H(p_i,p_j,p_k,p_l) = J^2_{ijk}(x_l - x_k) - J^3_{ijk}(y_l - y_k) + J^4_{ijk}((x_l - x_k)^2 + (y_l - y_k)^2), \quad (4.6.3)$$

where

$$J^2_{ijk} = \begin{vmatrix} y_i - y_k & (x_i - x_k)^2 + (y_i - y_k)^2 \\ y_j - y_k & (x_j - x_k)^2 + (y_j - y_k)^2 \end{vmatrix}, \quad (4.6.4)$$

$$J^3_{ijk} = \begin{vmatrix} x_i - x_k & (x_i - x_k)^2 + (y_i - y_k)^2 \\ x_j - x_k & (x_j - x_k)^2 + (y_j - y_k)^2 \end{vmatrix}, \tag{4.6.5}$$

$$J^4_{ijk} = \begin{vmatrix} x_i - x_k & y_i - y_k \\ x_j - x_k & y_j - y_k \end{vmatrix}, \tag{4.6.6}$$

This expression shows that the value of $H(p_i, p_j, p_k, p_l)$ can be computed in terms of the coordinates of the generators with respect to the origin at p_k. The computation based on this expression is stable in a numerical sense, because the translation of the origin of the coordinate system to one of the relevant generators enables us to avoid dealing with unnecessarily large numbers. So this way of computation is recommended. Indeed, by this way of computation a Voronoi diagram for one million generators scattered at random in a unit square can be constructed in single precision arithmetic (Sugihara and Iri, 1989d, 1992).

The principle presented here was proposed by Sugihara and Iri (1988). A FORTRAN program called VORONOI2 was constructed on the basis of this principle, and is open for public use (Sugihara and Iri, 1989c; see Okabe, 1994, for a review).

There are still other algorithms that are robust in finite-precision arithmetic. They include a stable swapping algorithm (Fortune, 1992a, 1995), a topology-oriented divide-and-conquer algorithm (Oishi and Sugihara, 1995) and a topology-oriented lift-up algorithm using a three-dimensional convex hull (Minakawa and Sugihara, 1997).

4.7 ALGORITHMS FOR HIGHER-DIMENSIONAL VORONOI DIAGRAMS

A Voronoi diagram in a three- or higher-dimensional space is more complicated, at least in the sense that the size of the Voronoi diagram for n generators cannot be bounded by $O(n)$. It is known that the number of k-faces of the Voronoi diagram for n generators in an m-dimensional space is of $O(n^{\min\lfloor m+1-k, \lfloor m/2 \rfloor\rfloor})$ for $0 \leq k \leq m$ (Klee, 1980).

In a three-dimensional space, for instance, the Voronoi diagram is a partition of the space into n convex polyhedra, but the total number of vertices of these polyhedra can be as many as $O(n^2)$. An example of such a situation is shown in Figure 4.7.1. Suppose that one half of n generators align on the x-axis and the other half align on the line passing through (0,0,1) parallel to the y-axis. Recall that a Voronoi vertex, and hence its dual a Delaunay tetrahedron, is created by each set of those four generators whose circumscribing sphere is an empty sphere. Hence, each set of two consecutive generators on one line and two consecutive generators on the other line create a Delaunay tetrahedron as shown in the figure, giving $O(n^2)$ tetra-

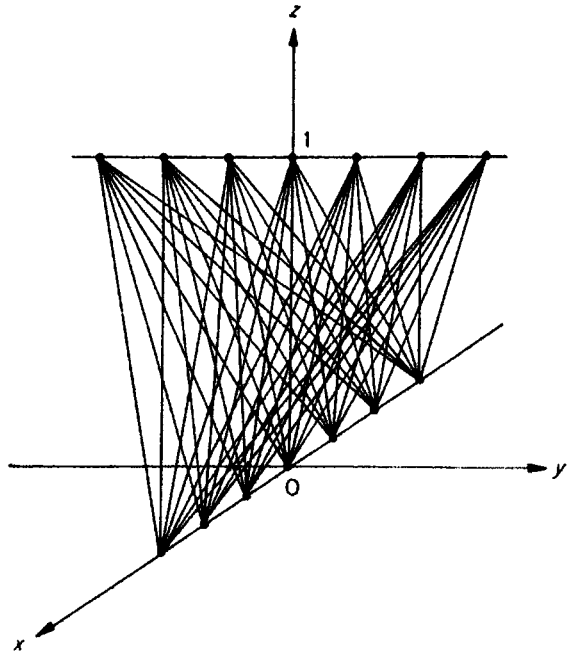

Figure 4.7.1 Three-dimensional Delaunay diagram with $O(n^2)$ tetrahedra.

hedra in total. Thus, $O(n^2)$ is a lower bound of the worst-case time complexity of an algorithm for constructing a Voronoi diagram in the three-dimensional space.

A simple and unifying approach to constructing an m-dimensional Voronoi diagram is to reduce the problem to a convex hull problem in an $(m+1)$-dimensional space.

As we saw in Chapter 2, we can establish a correspondence between the Voronoi diagram in the plane and a convex polyhedron in the three-dimensional space through the lift-up transformation. A similar correspondence is established in any dimension, and this fact can be used to construct the Voronoi diagram in the following way.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n generators in an m-dimensional space, and $(x_{i1}, x_{i2}, \ldots, x_{im})$ be the Cartesian coordinates of p_i. We consider the configuration of $t = m + 1$ generators. To avoid unnecessarily messy expressions, let us choose the first t generators $p_1, p_2, \ldots, p_t$ in P. These generators give a Voronoi vertex if and only if the hypersphere (a circle for $m = 2$, a sphere for $m = 3$, etc.) passing through them contains no other generators in its interior; actually the Voronoi vertex is the centre of this hypersphere.

The hypersphere S in an m-dimensional space that passes through $p_1, p_2, \ldots, p_t$ is represented by the equation

$$\begin{vmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} & x_{11}^2 + x_{12}^2 + \cdots + x_{1m}^2 \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} & x_{21}^2 + x_{22}^2 + \cdots + x_{2m}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_{t1} & x_{t2} & \cdots & x_{tm} & x_{t1}^2 + x_{t2}^2 + \cdots + x_{tm}^2 \\ 1 & x_1 & x_2 & \cdots & x_m & x_1^2 + x_2^2 + \cdots + x_m^2 \end{vmatrix} = 0, \tag{4.7.1}$$

with the variable point $p = (x_1, x_2, \ldots, x_m)$ (note that this equation is the extension of $H(p_i, p_j, p_k, p) = 0$, where $H(p_i, p_j, p_k, p)$ is defined in equation (2.4.10)). Hence the sign of the determinant in the left-hand side changes when the point p moves from inside S to outside.

Let p_i^* be the point in the t-dimensional space with coordinates

$$(x_{i1}, x_{i2}, \ldots, x_{im}, x_{i1}^2 + x_{i2}^2 + \ldots + x_{im}^2),$$

p^* be a variable point in the t-dimensional space with coordinates

$$(x_1, x_2, \ldots, x_m, x_1^2 + x_2^2 + \ldots + x_m^2),$$

and let $P^* = \{p_1^*, p_2^*, \ldots, p_n^*\}$. As illustrated in Figure 4.7.2, which is the case where $m = 2$, p_i^* is obtained by lifting up p_i in the positive direction of the tth coordinate axis until we reach the surface of the paraboloid of revolution

$$x_t = x_1^2 + x_2^2 + \ldots + x_m^2. \tag{4.7.2}$$

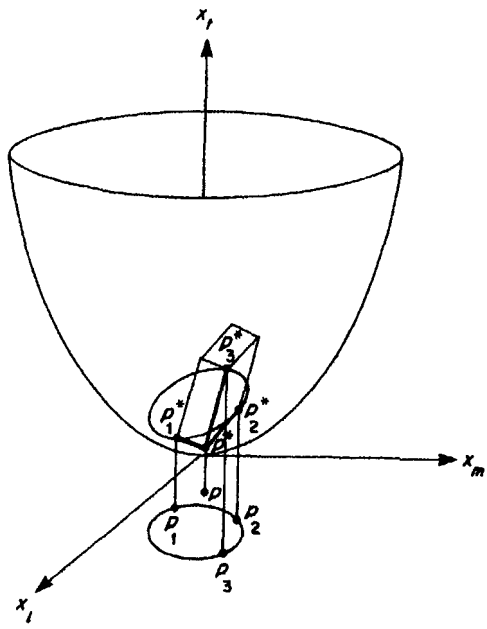

Figure 4.7.2 Lifting transformation that maps the plane to a hyperboloid of revolution.

Let S^* denote the hyperplane in the t-dimensional space that passes through $p_1^*, p_2^*, \ldots, p_t^*$. Then, S coincides with the projection along the tth axis of the intersection of S^* with the surface defined by equation (4.7.2).

The determinant in the left-hand side of equation (4.7.1) can be considered as the signed volume of the t-dimensional parallelepiped defined by t edges $\overline{p^*p_1^*}, \overline{p^*p_2^*}, \ldots, \overline{p^*p_t^*}$. Since the t-dimensional region consisting of points above the surface defined by equation (4.7.2) is convex, we can see that the point p is in the interior of the sphere S in the m-dimensional space if and only if the point p^* is below the hyperplane S^*. This implies that the set of t generators $p_1, p_2, \ldots, p_t$ gives a Voronoi vertex (that is, $p_1, \ldots, p_t$ form the vertices of a Delaunay simplex) if and only if $p_{t+1}^*, \ldots, p_n^*$ are all above the hyperplane S^*. Thus, all the Voronoi vertices (equivalently, all the Delaunay simplices) can be obtained by constructing the convex hull $\mathrm{CH}(P^*)$ of P^* in the t-dimensional space.

Suppose that $p_1^*, p_2^*, \ldots, p_t^*$ form an m-dimensional face (m-face, for short) of the convex hull $\mathrm{CH}(P^*)$ in the $(m+1)$-dimensional space. There is a unique hyperplane passing through these points, and all the other points $p_{t+1}^*, \ldots, p_n^*$ are on the same side of the hyperplane. We call the m-face a *lower* m-face if the other points are above the hyperplane, and an *upper* m-face otherwise. As we have seen, $p_1^*, \ldots, p_t^*$ forming a lower m-face implies that the hypersphere passing through $p_1, \ldots, p_t$ contains no other generators in its interior. Thus, the lower m-faces of $\mathrm{CH}(P^*)$ correspond to Voronoi vertices of $\mathcal{V}(P)$, or equivalently, the lower m-faces of $\mathrm{CH}(P^*)$ correspond to Delaunay simplices.

If $p_1^*, \ldots, p_t^*$ form an upper m-face of $\mathrm{CH}(P^*)$, the corresponding hypersphere in the m-dimensional space contains all the other generators $p_{t+1}, \ldots, p_n$ in its interior, which implies that $p_1, \ldots, p_t$ form a Voronoi vertex of the farthest-point Voronoi diagram for P.

Algorithm 4.7.1 (m-dimensional Voronoi diagram)

Input: Set $P = \{p_1, p_2, \ldots, p_n\}$ of n generators in the m-dimensional space with the coordinates $(x_{i1}, x_{i2}, \ldots, x_{im})$, $i = 1, 2, \ldots, n$.

Output: Delaunay tessellation spanning P.

Procedure:

Step 1. Create set $P^* = \{p_1^*, p_2^*, \ldots, p_n^*\}$ of points in an $(m+1)$-dimensional space, where the first m coordinates of p^*_i coincide with those of p_i and the $(m+1)$st coordinate is $x_1^2 + x_2^2 + \ldots + x_m^2$.

Step 2. Construct the convex hull $\mathrm{CH}(P^*)$ of P^* in the $(m+1)$-dimensional space.

Step 3. Project all the lower m-faces of $\mathrm{CH}(P^*)$ in the direction parallel to the $(m+1)$st coordinate axis onto the original m-dimensional space, and return the resulting diagram.

From the time-complexity point of view, the critical step is Step 2. There are many algorithms for constructing convex hulls in higher dimensions. One popular method is gift wrapping (Chand and Kapur, 1970; Preparata and

Shamos, 1985; Swart, 1985; Sugihara, 1994), where one supporting hyperplane is found initially and the neighbouring m-faces are found one by one just as we wrap the points using an elastic sheet. This method requires time proportional to $O(n^{[t/2]+1})$ for n points in the t-dimensional space, where $[x]$ represents the largest integer not greater than x.

Another popular method is an incremental method (which is sometimes called a beneath–beyond method) (Preparata and Shamos, 1985; Seidel, 1986; Edelsbrunner, 1987), where a convex hull of a small number of points is constructed initially and then the other points are added one by one. When a new point is added, the m-faces of the previous convex hull are divided into those visible from the new point and the others, and the former m-faces are replaced with new m-faces containing the new point as a vertex. This method requires $O(n^{[(t+1)/2]} + n \log n)$ time.

Other higher-dimensional convex hull algorithms include a shelling method, which runs in $O(n^{[t/2]} \log n)$ time (Seidel, 1987), and a divide-and-conquer method, which runs in $O(n \log n + n^{[t/2]})$ time (Buckley, 1988).

Therefore, the incremental method is optimal in the even-dimensional space, while the divide-and-conquer method is optimal in the odd-dimensional space. Hence, for example, the three-dimensional Voronoi diagram can be constructed in $O(n^2)$ time by Algorithm 4.7.1 using the incremental method for the four-dimensional convex hull.

The farthest-point Voronoi diagram (actually the dual diagram of the farthest-point Voronoi diagram) can be obtained if we project all the upper m-faces, instead of the lower m-faces, in Step 3 of Algorithm 4.7.1.

The reduction of the Voronoi diagram construction in the m-dimensional space to a convex hull construction in the $(m+1)$-dimensional space was pointed out by Brown (1979, 1980), where another transformation called inversion (which maps the m-dimensional plane to a hypersphere in the $(m+1)$-dimensional space) was used (see Figure 2.4.8 and equation (2.4.10)). That transformation was used by Aurenhanuner and Edelsbrunner (1984), Edelsbrunner (1987), Buckley (1988) and Mulmuley (1991). The transformation presented in this section (i.e. lift-up to the paraboloid of revolution) was used by Edelsbrunner and Seidel (1986), Edelsbrunner (1987) and Chan *et al.* (1997). This transformation can be considered as an extremal case of the inversion in which the centre of inversion is taken as a point at infinity.

It should be noted that the lift-up transformation can be used also for two-dimensional points (O'Rourke *et al.*, 1986). In this case the problem is converted to the construction of a three-dimensional convex hull, and hence this method runs in $O(n \log n)$ time if the divide-and-conquer method is applied (Preparata and Hong, 1977); this time complexity is optimal in the worst-case sense. The lift-up transformation gives also a linear-time algorithm for the Voronoi diagram for points forming a convex polygon (Agarwal *et al.*, 1989a).

Other algorithms for constructing higher-dimensional Voronoi diagrams include the naive method (Brostow *et al.*, 1978; Finney, 1979), a walking method (Avis and Bhattacharya, 1983), incremental methods (Bowyer, 1981;

Watson, 1981; Devijver and Dekesel, 1983; Tanemura *et al.*, 1983; Inagaki *et al.*, 1992; Borouchaki and Lo, 1995), a flipping method (Joe, 1991a), a randomized method (Dwyer, 1991; Facello, 1995), the bucketing method (Fang and Piegl, 1995), and others (Dwyer, 1991; Medvedev, 1986).

4.8 ALGORITHMS FOR GENERALIZED VORONOI DIAGRAMS

Let us return to two-dimensional Voronoi diagrams. As we saw in Chapter 3, there is a wide variety of generalizations of the concept of the Voronoi diagram. However, we do not have space to treat them individually. Hence we try to present a general framework for constructing Voronoi diagrams of various types.

To represent a generalized Voronoi diagram in a unified manner, we introduce the following notations. Let $\Gamma = \{A_1, A_2, \ldots, A_n\}$ be the set of generators, and for any point z in the plane let $d(z, A_i)$ denote a 'distance metric' representing how far the point z is from the generator A_i (here we call d the 'distance' even if d does not satisfy the distance axiom). As in Chapter 3, we define the bisector of A_i and A_j by

$$b(A_i, A_j) = \{z \mid d(z, A_i) = d(z, A_j)\}, \tag{4.8.1}$$

and the dominance region of A_i over A_j by

$$(A_i, A_j) = \{z \mid d(z, A_i) \leq d(z, A_j)\}. \tag{4.8.2}$$

For any generator A_i, we define the Voronoi region for A_i with respect to the distance d by

$$V(A_i) = \bigcap_{j \neq i} \mathrm{Dom}(A_i, A_j). \tag{4.8.3}$$

The collection of $V(A_1), V(A_2), \ldots, V(A_n)$ is called the *Voronoi diagram for* Γ *with respect to* d.

For an ordinary Voronoi diagram, generators are distinct points, and d is the Euclidean distance. For a weighted Voronoi diagram, generators are distinct points, and d is an additively weighted, a multiplicatively weighted or a compoundly weighted distance. For a farthest-point Voronoi diagram, generators are distinct points, and $d(z, A_i)$ is a monotone decreasing function of the Euclidean distance between z and A_i; for example, the inverse of the Euclidean distance between z and A_i can be used as $d(z, A_i)$. For a Voronoi diagram with obstacles, generators are distinct points, and $d(z, A_i)$ is the minimum length of the obstacle-avoiding path from A_i to z. For a Voronoi diagram of line segments or areas, generators are line segments or areas, respectively, and $d(z, A_i)$ is the minimum of the Euclidean distance from z to a point in A_i. For a Voronoi diagram with a non-Euclidean distance, generators are distinct points, and d is the associated non-Euclidean distance.

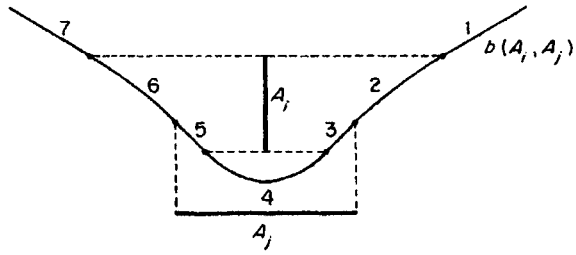

Figure 4.8.1 Bisector of two line segments.

We restrict our consideration to the case where every bisector $b(A_i, A_j)$ is a single curve without self-intersection, closed or extended infinitely in both directions, dividing the plane into two connected regions $\mathrm{Dom}(A_i, A_j)$ and $\mathrm{Dom}(A_j, A_i)$. This case includes almost all the generalized Voronoi diagrams listed in Chapter 3. An example of an exception is a bisector $b(A_i, A_j)$ with respect to an L_∞-distance of two generators aligning horizontally or vertically, where $b(A_i, A_j)$ includes a two-dimensional region. A similar exception arises for the L_1-distance Voronoi diagram and the convex-distance Voronoi diagram.

For an ordinary Voronoi diagram, $b(A_i, A_j)$ is the straight line forming the perpendicular bisector of $\overline{A_iA_j}$. For a Voronoi diagram with a power distance (i.e. a power diagram) and for a farthest-point Voronoi diagram, the bisector $b(A_i, A_j)$, if it exists, is a straight line. In general, however, $b(A_i, A_j)$ is a complicated curve. For instance, the bisector of $b(A_i, A_j)$ of two line segments A_i and A_j with respect to the Euclidean distance can be a concatenation of up to seven segments of parabolas and straight lines, as shown in Figure 4.8.1, where segments 1 and 7 are perpendicular bisectors of two terminal points, segments 2, 4 and 6 are parabolas with one generator being the directorix and a terminal point of the other generator being the focus, and segments 3 and 5 are bisectors of the angles formed by the two generators. The bisector $b(A_i, A_j)$ with respect to a multiplicatively weighted distance is a circle containing the generator with the smaller weight. The bisector $b(A_i, A_j)$ with respect to an additively weighted distance is a hyperbola with A_i and A_j being the foci. Thus, the bisector changes its type of line according to the generators and the distance. In what follows we assume that we can construct $b(A_i, A_j)$ for any pair of generators in constant time, and that we can find the points of intersection of two bisectors also in constant time.

The concepts of the bisector and the dominance region for a pair of generators can be extended to those for a pair of disjoint subsets of generators. Let Γ' and Γ'' be mutually disjoint subsets of Γ. We define

$$b(\Gamma', \Gamma'') = \{z \mid \min_{A_i \in \Gamma'} d(z, A_i) = \min_{A_j \in \Gamma''} d(z, A_j)\} \tag{4.8.4}$$

and call it the *bisector* of Γ' and Γ''. Similarly, we define

$$\mathrm{Dom}(\Gamma', \Gamma'') = \{z \mid \min_{A_i \in \Gamma'} d(z, A_i) \leq \min_{A_j \in \Gamma''} d(z, A_j)\} \tag{4.8.5}$$

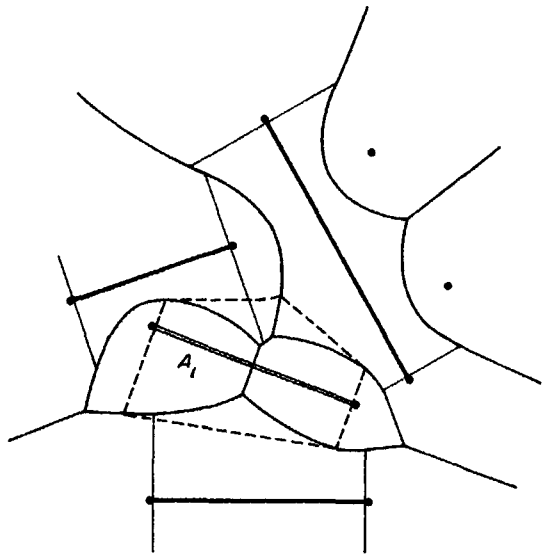

Figure 4.8.2 Incremental method for a Voronoi diagram for line segments.

and call it the *dominance region* of Γ' over Γ''. $b(\{A_i\}, \Gamma')$ and $\mathrm{Dom}(\{A_i\}, \Gamma'')$ are abbreviated to $b(A_i, \Gamma'')$ and $\mathrm{Dom}(A_i, \Gamma'')$, respectively.

In the incremental method, the Voronoi diagram $\mathcal{V}(\{A_1, \ldots, A_l\})$ is constructed from the Voronoi diagram $\mathcal{V}(\{A_1, \ldots, A_{l-1}\})$ for each l, where the main task is to find the bisector $b(A_l, \{A_1, \ldots, A_{l-1}\})$. Let us assume that the Voronoi region $V(A_i)$ is simply connected. Then, for any $\Gamma' \subset \Gamma \setminus \{A_i\}$, $b(A_i, \Gamma')$ consists of a single connected curve, a closed curve or a curve extended infinitely in both directions. This is the case for an additively weighted Voronoi diagram, a power diagram, a farthest-point Voronoi diagram, a Voronoi diagram of lines or areas, and a Voronoi diagram with an L_p-metric for $1 < p < \infty$. For those types of Voronoi diagrams the following incremental method can be used.

Algorithm 4.8.1 (Incremental method for a generalized Voronoi diagram)
Input: Set $\Gamma = \{A_1, A_2, \ldots, A_n\}$ of n generators and distance d.
Output: Voronoi diagram $\mathcal{V}(\Gamma) = \{V(A_1), V(A_2), \ldots, V(A_n)\}$ for Γ with respect to the distance d.
Procedure:
Comment. Let $\Gamma_l = \{A_1, A_2, \ldots, A_l\}$ for $1 \le l \le n$.
Step 1. Construct $\mathcal{V}(\Gamma_2)$.
Step 2. For $l = 3, 4, \ldots, n$, do 2.1, 2.2 and 2.3.
2.1. Find a point on $b(A_l, \Gamma_{l-1})$.
2.2. Starting with the point found in 2.2, trace the whole bisector $b(A_l, \Gamma_{l-1})$.

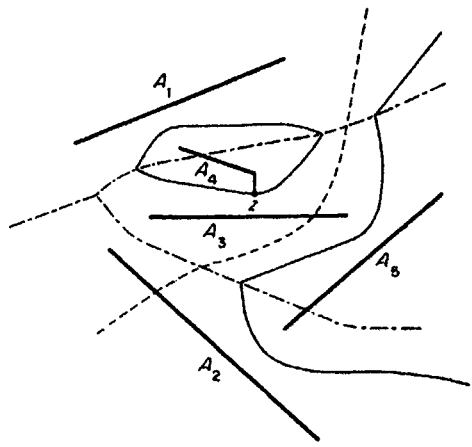

Figure 4.8.3 Merger of two Voronoi diagrams $\mathcal{V}(\{A_1, A_2, A_3\})$ and $\mathcal{V}(\{A_4, A_5\})$.

2.3. Construct $\mathcal{V}(\Gamma_l)$ by inserting $b(A_l, \Gamma_{l-1})$ to $\mathcal{V}(\Gamma_{l-1})$ and then deleting the substructure inside $\mathrm{Dom}(A_l, \Gamma_{l-1})$.

Step 3. Return $\mathcal{V}(\Gamma) = \mathcal{V}(\Gamma_n)$.

An example of the addition of a generator is shown in Figure 4.8.2 for the case of a line Voronoi diagram. In this example each generator is partitioned into two terminal points and an open line segment, and they are treated as distinct generators. The Voronoi diagram for all the terminal points is generated first, and then open line segments are added one after another. The double line segment represents the new generator g_l and the broken lines represent $b(A_i, \Gamma_{l-1})$.

If the Voronoi region $V(A_i)$ contains the associated generator A_i, just as in the cases for a Voronoi diagram with a non-Euclidean metric and a Voronoi diagram for line segments or areas, the starter point in Step 2.2 can be found easily. All we have to do is first find the Voronoi region $V(A_i)$ containing at least one point in the new generator A_l and then find the point of the intersection of $b(A_i, A_l)$ with the boundary of $V(A_i)$. If $V(A_i)$ does not necessarily contain A_i, as is the case for a power diagram and a farthest-point Voronoi diagram, some non-trivial method should be constructed to find the starter point.

A randomized version of the incremental algorithm was studied by Mehlhorn *et al.* (1991) and Klein *et al.* (1990, 1993).

Next, let us consider the divide-and-conquer method. In this method two Voronoi diagrams $\mathcal{V}(\Gamma')$ and $\mathcal{V}(\Gamma'')$ are merged to $\mathcal{V}(\Gamma' \cup \Gamma'')$, where the main task is to find the bisector $b(\Gamma', \Gamma'')$.

For an ordinary Voronoi diagram the bisector $b(\Gamma', \Gamma'')$ is always a single connected curve (consisting of line segments). For a generalized Voronoi diagram, on the other hand, $b(\Gamma', \Gamma'')$ is not necessarily connected, but is a collection of a finite number of connected curves, some closed and the others are extended infinitely in both directions.

In Figure 4.8.3 an example of the situation where $b(\Gamma', \Gamma'')$ is disconnected is shown for the case of the Voronoi diagram for line segments, where $\Gamma' = \{A_1, A_2, A_3\}$ and $\Gamma'' = \{A_4, A_5\}$. Dot-and-dash lines represent $\mathcal{V}(\Gamma')$, broken lines represent $\mathcal{V}(\Gamma'')$ and solid lines represent $b(\Gamma', \Gamma'')$. Note that the partition of Γ into Γ' and Γ'' is not very sophisticated; we obtain this partition if we choose the leftmost points of the line segments as the representatives and divide them into a left set Γ' and a right set Γ''.

To enumerate all the connected curves, we need a set of 'starters' from which we start tracing the curves. The set S of points is called a *starter set* of $b(\Gamma', \Gamma'')$ if any connected component of $b(\Gamma', \Gamma'')$ has at least one point in S. To trace the curves, S need contain only one point for each connected component. However, it is not always easy to tell the number of components of $b(\Gamma', \Gamma'')$ before we construct $b(\Gamma', \Gamma'')$. Hence, we put in S candidates of starter points that are enough to trace all components. Thus, a divide-and-conquer algorithm can be described in the following way.

Algorithm 4.8.2 (Divide-and-conquer method for a generalized Voronoi diagram)

Input: Set $\Gamma = \{A_1, A_2, \ldots, A_n\}$ of n generators and distance d.

Output: Voronoi diagram $\mathcal{V}(\Gamma) = \{V(A_1), V(A_2), \ldots, V(A_n)\}$ for Γ with respect to the distance d.

Procedure:

Step 1. If $n \leq 2$, construct $\mathcal{V}(\Gamma)$ directly and return it. Otherwise, do Steps 2, 3, ..., 8.

Step 2. Divide Γ into two disjoint subsets Γ' and Γ'' of almost the same size.

Step 3. Construct $\mathcal{V}(\Gamma')$ by Algorithm 4.8.2.

Step 4. Construct $\mathcal{V}(\Gamma'')$ by Algorithm 4.8.2.

Step 5. Find a starter set S of $b(\Gamma', \Gamma'')$, and put $B \leftarrow \emptyset$.

Step 6. While S is not empty, choose and delete an element s from S and if s is not on any curve in B, trace the connected component of $b(\Gamma', \Gamma'')$ containing s and put it in B.

Step 7. Merge $\mathcal{V}(\Gamma')$ and $\mathcal{V}(\Gamma'')$ into $\mathcal{V}(\Gamma)$ by inserting all the curves in B and deleting the substructure inside $\mathrm{Dom}(\Gamma', \Gamma'')$ from $\mathcal{V}(\Gamma'')$ and the substructure inside $\mathrm{Dom}(\Gamma'', \Gamma')$ from $\mathcal{V}(\Gamma')$.

Step 8. Return $\mathcal{V}(\Gamma)$.

The most difficult step in this algorithm is Step 5. We prefer a smaller starter set, but in general it is not easy to tell the number of components in $b(\Gamma, \Gamma'')$ unless we construct $b(\Gamma', \Gamma'')$ itself. For any $A_i \in \Gamma'$ and $A_j \in \Gamma''$, if point z with $d(z, A_i) = d(z, A_j)$ lies both in $V(A_i)$ of $\mathcal{V}(\Gamma')$ and in $V(A_j)$ of $V(\Gamma'')$, then z is on $b(\Gamma', \Gamma'')$, and hence can be a starter point (Lee and Drysdale, 1981). For instance, the point z in Figure 4.8.3 satisfies the above condition; this point is the midpoint of the perpendicular line segment dropped from the rightmost point of A_4 to A_3.

If the generators are points, we can partition Γ into a left generator set Γ' and a right generator set Γ'', and in this case we can sometimes find the starter set easily. For a Voronoi diagram with the Laguerre metric, $b(\Gamma', \Gamma'')$ consists of a single curve extended infinitely, and the starter point can be found from the common support of Γ' and Γ'' just as in the case of an ordinary Voronoi diagram (Imai *et al.*, 1985). For a Voronoi diagram with an L_p-metric for $1 < p < \infty$, $b(\Gamma', \Gamma'')$ is not necessarily connected, but every connected component in $b(\Gamma', \Gamma'')$ extends infinitely and the starter set can be found in a similar way from the convex hulls of Γ' and Γ'' (Chew and Drysdale, 1985). Unifying studies of divide-and-conquer methods can be found in Chew and Drysdale (1985), Klein (1988, 1989), Klein and Wood (1988), and Klein *et al.* (1990).

There are a variety of other algorithms for individual types of generalized Voronoi diagrams. We briefly summarize them.

For the power diagram, Imai *et al.* (1985) proposed an incremental algorithm, which runs in $O(n^2)$ time in the worst case. On the other hand, Aurenhammer (1987a) pointed out that the power diagram in the plane can be obtained as the orthographic projection of the intersection of three-dimensional half spaces, and constructed a divide-and-conquer algorithm which runs in $O(n \log n)$ time. Gavrilova and Ronke (1996) also studied the construction of the power diagram. For the sectional Voronoi diagram, see Ash and Bolker (1986), Aurenhammer (1988b), Imai *et al.* (1985) and Sibson (1980a) (also see Section 3.1.5).

The construction of the multiplicatively weighted Voronoi diagram is studied by Aurenhammer and Edelsbrunner (1984), Ash and Bolker (1986), and Aurenhammer (1988b), and the additively weighted Voronoi diagram is studied by Ash and Bolker (1986), Fortune (1986, 1987) and Hanjoul *et al.* (1989).

Typical algorithms for the order-k Voronoi diagram and the ordered order-k Voronoi diagram in the plane are based on the arrangement of planes in the three-dimensional space. With each generator $p_i = (x_i, y_i)$, we associate the plane $z = 2x_i x + 2y_i y - (x_i^2 + y_i^2)$. Let A be the arrangement composed of such planes associated with all the generators. Property V17 can be restated that the ordinary Voronoi diagram is obtained as the orthographic projection of the uppermost layer of the arrangement A (note that the surface structure of the intersection of the upper half spaces bounded by these planes is nothing but the uppermost layer of A). The projection of the second layer of A corresponds to the order-2 Voronoi diagram, and the projection of the first and second layers together corresponds to the ordered order-2 Voronoi diagram. Similarly, the kth layer corresponds to the order-k Voronoi diagram, and the collection of the upper k layers corresponds to the ordered order-k Voronoi diagram (Aurenhammer, 1988b; Chazelle *et al.*, 1986; Mulmuley, 1991). In particular, the nth layer (the lowest layer) corresponds to the farthest-point Voronoi diagram (Brown, 1979; Buckley, 1988). Hence, the (ordered) order-k Voronoi diagram can be constructed through the arrangement A (Seidel, 1982; Chazelle and Edelsbrunner, 1985, 1987; Edelsbrunner

and Seidel, 1986; Aurenhammer, 1988b, 1990b; Boissonnat *et al.*, 1990, 1993). Aurenhammer and Schwarzkopf (1991) proposed a randomized incremental algorithm. A similar strategy using a slightly modified arrangement can be used for the order-k power diagram (Aurenhammer, 1987) and for the multiplicatively weighted Voronoi diagram (Aurenhammer and Edelsbrunner, 1984). Lee (1982b) studied a divide-and-conquer strategy. For the farthest-point Voronoi diagram, an incremental algorithm was also studied (Suzuki, 1989).

For the shortest-path Voronoi diagram, see El Gindy and Avis (1981), Lee (1983), Lee and Chen (1985), Imai *et al.* (1985), Chazelle and Guibas (1985), Asano and Asano (1987) and Seoung and Asano (1987). Algorithms for the Voronoi diagram for a simple polygon were proposed by Asano and Asano (1987) and Aronov (1987, 1989). An algorithm for the visibility Voronoi diagram was presented by Aurenhammer (1988b).

The Voronoi diagram for line segments can be constructed by the plane sweep method (Fortune, 1986, 1987), by the incremental methods (Kokubo, 1985; Imai *et al.*, 1985; Imai and Sugihara, 1994), by the divide-and-conquer method (Lee and Drysdale, 1981; Yap, 1987), and by other methods (Lee and Lin, 1986; Burnikel *et al.*, 1994; Gold *et al.*, 1995).

For the Voronoi diagram for polygons, see Preparata (1977), Lee (1982a), Aronov (1989), McAllister *et al.* (1996), Srinivasan and Nackman (1987) and Meshkat and Sakkas (1987), and for the Voronoi diagram for areas, see O'Dúnlaing *et al.* (1986), Leven and Sharir (1987) and Canny and Donald (1988). Chou (1995) studied an algorithm for the Voronoi diagram inside an arbitrary closed curve.

Variations in the distance are also studied from an algorithmic point of view. The Voronoi diagram for the Manhattan distance was studied by Carter *et al.* (1972), Hwang (1979), Lee (1980), Lee and Wong (1980), Chang *et al.* (1990a), Shute *et al.* (1991) and Sakakibara *et al.* (1996); for the Karlsruhe distance by Koshizuka and Kurita (1986) and Klein (1988); for the Hausdorff distance by Aurenhammer (1988b); for the elliptic distance by Nielson (1993); and for the general convex distance by Drysdale (1990) and Kao and Mount (1991). The shortest path on the surface of the three-dimensional object defines another type of distance. Distances of this type include the Voronoi diagram on a sphere (Miles, 1971; Brown, 1980; Paschinger, 1982; Ash and Bolker, 1985; Augenbaum and Peskin, 1985), on a cone (Dehne and Klein, 1987; Klein, 1988; Klein and Wood, 1988), and on a polyhedral surface (Aronov and O'Rourke, 1992).

The constrained Voronoi and Delaunay diagrams are another important direction of the generalizations. They are studied mainly from the mesh-generation point of view by Chew (1987, 1989b), Cline and Renka (1990), Du (1996), Fang and Piegl (1994), Borgers (1990), Lo (1989), Sloan (1993) and Zhou *et al.* (1990). Wang and Schubert (1987) constructed an optimal $O(n \log n)$ algorithm in which the Voronoi diagram for the end points was constructed first and then revised. Wang and Tsin (1990) showed that the complexity of the multiplicatively weighted Voronoi diagram is of $O(n^4)$. For

the purpose of finite element meshes, approximations of the constrained Delaunay diagrams are studied; see Section 6.5 for details.

Another important class of generalization is the dynamic Voronoi diagrams. There are two meanings to 'dynamic'. The first is that the generators move from time to time (Tokuyama, 1988; Roos, 1993), and the other is that the generators appear or disappear occasionally (Devillers *et al.*, 1992). Other generalizations include the oriented Voronoi diagram (Chang *et al.*, 1990a) and the Voronoi diagram in a river (Sugihara, 1992a).

4.9 APPROXIMATION ALGORITHMS

Sometimes we want to construct Voronoi diagrams for only a few different sets of generators. In such a situation we cannot ignore the time required in developing a computer program; the time for writing a program is often more serious than the running time itself. This problem might be solved by approximation schemes.

If generators are mutually disjoint sets of points (such as curves and areas) and if we want to construct a Voronoi diagram for them with respect to distance d, the problem can be reduced to the construction of a Voronoi diagram for points with respect to d in the following way. Given a set of generators, we first replace each generator with a finite number of points that approximate the original generator, then construct the Voronoi diagram for these points, and finally remove superfluous Voronoi edges and superfluous Voronoi vertices. Thus, we get the next algorithm.

Algorithm 4.9.1 (Approximation by points)

Input: Set $\Gamma = \{A_1, A_2, \ldots, A_n\}$ of n disjoint figures in the plane.

Output: Approximation of the Voronoi diagram for Γ.

Procedure:

Step 1. For each $i = 1, 2, \ldots, n$, create a finite set P_i of points that approximates the boundary of A_i.

Step 2. Construct the Voronoi diagram $\mathcal{V}$ for the point set $P_1 \cup P_2 \cup \ldots \cup P_n$.

Step 3. Delete from $\mathcal{V}$ those Voronoi edges whose generator points belong to the same original figure. Delete isolated Voronoi vertices, if any, from $\mathcal{V}$.

Step 4. Return $\mathcal{V}$.

Figure 4.9.1 shows an example of this scheme applied to a set of generators containing different types of figures such as line segments, circular arcs and polygons; (a) represents the Voronoi diagram for 882 points located on the boundaries of the original figures, and (b) shows the approximation of the Voronoi diagram for the original figures obtained from (a).

To obtain a good approximation, we have to use many points to approximate the original generators. Hence, in general, the number of generators (= points)

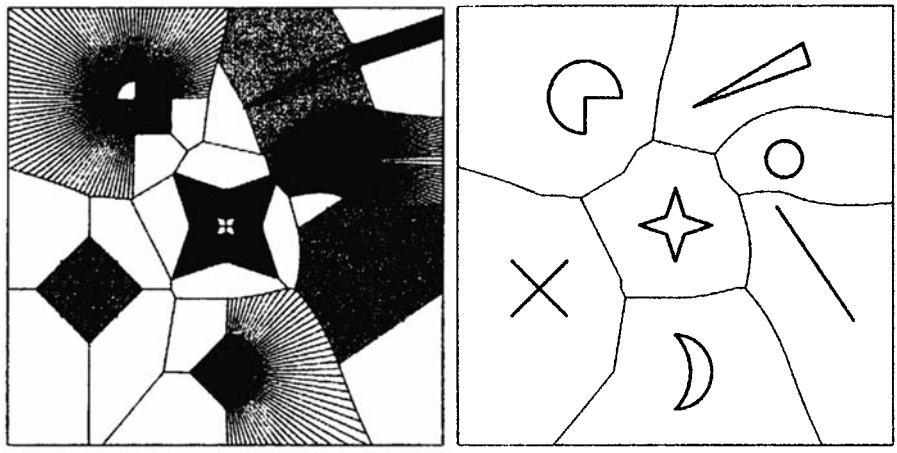

Figure 4.9.1 Approximation of generators by sets of points: (a) Voronoi diagram for 882 points that approximate seven generator figures; (b) approximation of the generalized Voronoi diagram obtained by deleting superfluous edges and superfluous vertices from (a).

becomes very large. Moreover, these points often give highly degenerate configurations. Therefore the approximation scheme is valid only when we can use a fast and robust algorithm, such as the one developed in Section 4.6. This approximation scheme can be extended to an m-dimensional generalized Voronoi diagrams directly, if an algorithm for Voronoi diagrams for points in the m-dimensional space is available.

Another, more brute-force approximation scheme is the use of digital image techniques. A digital image is a two-dimensional array, say $I(i,j)$, $1 \leq i, j \leq N$, where the entry $I(i,j)$ represents some property of the image at the point with coordinate (i,j). Each point (i,j) is called a *pixel*, and $I(i,j)$ is called a *pixel value*. A pixel with a positive pixel value is called a *positive pixel*. The four-neighbour set of pixel (i,j) is defined by

$$\mathrm{NB}_4(i,j) = \{(i,j-1),(i,j+1),(i-1,j),(i+1,j)\},$$

and the eight-neighbour set of pixel (i,j) is defined by

$$\mathrm{NB}_8(i,j) = \mathrm{NB}_4(i,j) \cup \{(i-1,j-1),(i-1,j+1),(i+1,j-1),(i+1,j+1)\},$$

where pixels outside the array are ignored. The four-neighbour set or the eight-neighbour set of (i,j) is simply called a neighbour set of (i,j) and is denoted by $\mathrm{NB}(i,j)$.

In the next scheme, an approximation of a Voronoi diagram is constructed in the form of a digital image. First, each generator is replaced with a set of pixels, and then from these pixels the 'territories' of the generators are expanded simultaneously at the same speed until the territories collide with

one another. This can be done by repeating local parallel operations over the neighbours of pixels in the following way (an example of the behaviour of the algorithm immediately follows the algorithm).

Algorithm 4.9.2 (Approximation by a digital image)

Input: Integer N and set $\Gamma = \{A_1, A_2, \ldots, A_n\}$ of n disjoint figures in the plane.

Output: Digital image $I(i,j)\,(1 < i,j < N)$ approximating the Voronoi diagram for Γ, where $I(i,j) = k > 0$ if pixel (i,j) belongs to the Voronoi region of A_k, and $I(i,j) = -1$ if pixel (i,j) is incident to two or more different Voronoi regions.

Procedure:

Step 1. For all $I < i,j < N$,
$I(i,j) \leftarrow k$ if there exists k such that $(i,j) \in A_k$,
$I(i,j) \leftarrow 0$ otherwise.

Step 2. Repeat 2.1 until pixel values are not changed any more.

2.1. For all (i,j) such that $I(i,j) = 0$, do the following simultaneously.
$I(i,j) \leftarrow k$ if all the positive pixels in $\mathrm{NB}(i,j)$ have the same value k,
$I(i,j) \leftarrow -1$ if the positive pixels in $\mathrm{NB}(i,j)$ admit two or more different values.

Step 3. Return the array $I(i,j)$.

An example of the behaviour of the algorithm is shown in Figure 4.9.2. Panel (a) shows the initial value of the array, where the value i in a circle represents the ordinal number of the generator A_i to which the pixel belongs, and blank pixels represent those with value 0. The generators A_2, A_3 and A_4 are represented by one pixel, whereas the generators A_1 and A_5 occupy more than one pixel. The array after the first, the second and the third execution of Step 2.1 are shown in (b), (c) and (d), respectively, where the eight-neighbour set $\mathrm{NB}_8(i,j)$ is used as the neighbour set in the algorithm. After the third execution of Step 2.1 all the pixels have non-zero values, and hence the algorithm terminates. In (d), the boundaries of the Voronoi regions are represented by bold lines.

In Step 2.1 the pixel values are changed simultaneously; this is a kind of parallel procedure. To implement this procedure in the usual sequential computer, we use one more array, say $J(i,j)$, of the same size; we write the resulting pixel values in $J(i,j)$ in the first execution of Step 2.1, and switch the roles of these two arrays $I(i,j)$ and $J(i,j)$ every time we execute Step 2.1.

This approximation scheme is conceptually simple and does not require any exact algorithm for Voronoi diagrams for points, but is very time and space consuming. Moreover, the output of this algorithm is not a winged-edge data structure; the output merely gives us approximations of Voronoi regions and we need further processing to extract the incidence relations

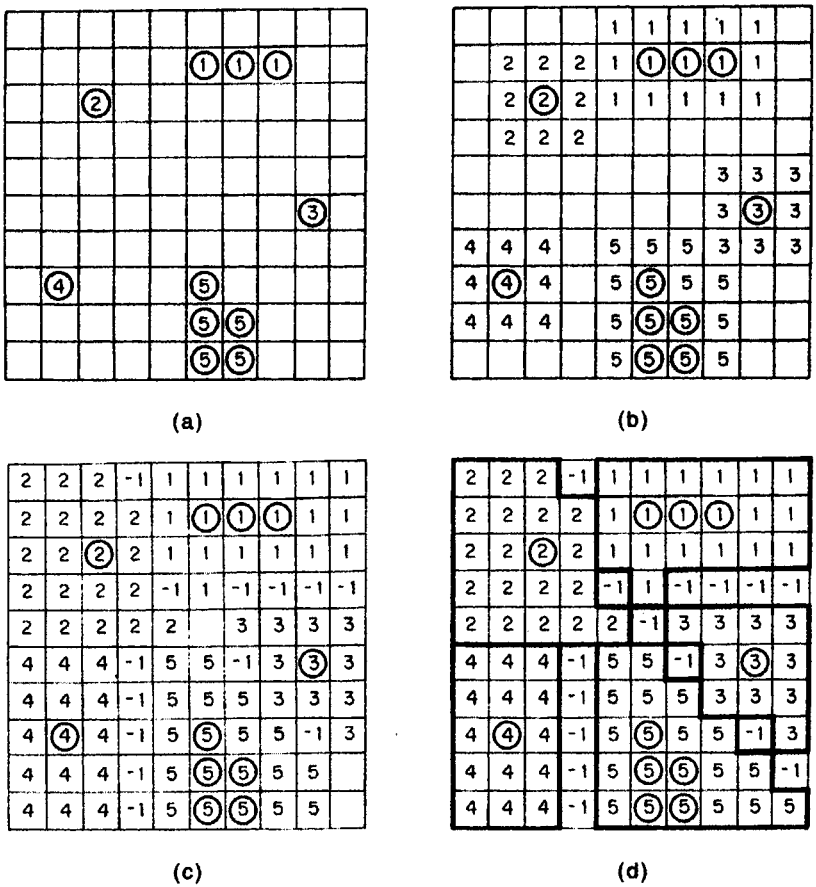

Figure 4.9.2 Approximation of the Voronoi diagram by a digital image: (a) digital image of five generators; (b) first execution of Step 2.1; (c) second execution of Step 2.1; (d) final result.

among the regions, the edges and the points. Furthermore, the distance is not Euclidean. If the four-neighbour set is used in Step 2.1, the 'territories' expand according to the L_1-metric, while if the eight-neighbour set is used, they expand according to the L_∞-metric. If these two kinds of the neighbour sets are used alternately in the repetition of Step 2.1, the output gives a slightly better approximation of a Euclidean-metric Voronoi diagram. Therefore this approximation scheme should be used only for special purposes such as for Voronoi diagrams with the L_1- or L_∞-metric or for very rough approximations of Voronoi diagrams.

On the other hand, if the original generators themselves are given in the form of a digital image, as is the case in digital pattern analysis, Algorithm 4.9.2 can be applied directly to various kinds of proximity analysis (Toriwaki *et al.*, 1982; Mark, 1987; Yoshitake *et al.*, 1987; Toriwaki and Yokoi, 1988).