GRAPHICAL MODELS AND IMAGE PROCESSING
Vol. 57, No. 4, July, pp. 303–314, 1995

Topology-Oriented Divide-and-Conquer Algorithm for Voronoi Diagrams*

YASUAKI OISHI AND KOKICHI SUGIHARA

Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Received November 23, 1993; revised January 4, 1995; accepted March 21, 1995

The conventional divide-and-conquer algorithm for constructing Voronoi diagrams is revised into a numerically robust one. The strategy for the revision is a topology-oriented approach. That is, at every step of the algorithm, consistency of the topological structure is considered more important than the result of numerical computation, and hence numerical values are used only when they do not contradict the topological structure. The resultant new algorithm is completely robust in the sense that, no matter how poor the precision may be, the algorithm always carries out its task, ending up with a topologically consistent output, and is correct in the sense that the output "converges" to the true Voronoi diagram as the precision becomes higher. Moreover, it is efficient in the sense that it achieves the same time complexity as the original divide-and-conquer algorithm unless the precision in computation is too poor. The performance of the algorithm is also verified by computational experiments. © 1995 Academic Press, Inc.

1. INTRODUCTION

Rapid progress of computational geometry in the past two decades has given tremendously many geometric algorithms. However, most of them are designed under the assumption that numerical computation can be done precisely. In actual computation, on the other hand, numerical computations are usually carried out only in finite precision, so that errors are inevitable. These errors often generate inconsistency in topological structures, which in turn causes the algorithm to fail. Thus, there is a great gap between theoretically correct geometric algorithms and practically valid computer programs [7].

Recently, the importance of this difficulty has gradually been recognized, and a number of methods for designing numerically robust algorithms have been proposed. These methods can be classified into four groups: a tolerance approach, an exact-arithmetic approach, an error-analysis approach, and a topology-oriented approach.

In the tolerance approach, a small positive number called a tolerance, say ε, is fixed, and whenever two geometric elements come closer than ε, they are regarded as being at the same location [6]. This is one of the most common methods used by actual programmers. However, this method does not overcome the difficulty completely, because this method sometimes generates inconsistent situations if three or more elements come close to each other.

The exact-arithmetic approach [1, 14, 18] employs precise arithmetic such as rational arithmetic. By this approach we can indeed avoid topological misjudgements, but must pay expensive cost for high-precision computation; high-precision computation requires a time-consuming procedure for every numerical computation on one hand, and decreases the portability of the algorithm on the other.

In the error-analysis approach, computational results are classified under reliable and unreliable according to the estimated error bounds, and only the reliable results are used in the algorithms [4, 10]. However, to find a good error bound is not easy in general. Moreover, the resultant algorithm is also complicated because the thresholding based on the error bounds complicates the algorithm implementation.

The topology-oriented approach starts with the assumption that every numerical computation contains errors and places higher priority on consistency of the topological structure than on numerical values, and the results of numerical computation are used only when they do not violate the topological consistency [16, 19, 20]. Unlike the exact-arithmetic approach, this approach does not require expensive cost for numerical computation. Unlike the error-analysis approach, this approach does not require complicated analysis of error bounds. In this sense this approach seems promising for designing numerically robust geometric algorithms.

In the present paper, we take the topology-oriented approach to the design of a new numerically robust divide-and-conquer algorithm for constructing Voronoi diagrams.

* This work is partly supported by the Grant in Aid for Scientific Research of the Ministry of Education, Science, and Culture of Japan.

1077-3169/95 $12.00

The topology-oriented approach was already applied successfully to the incremental algorithm for the Voronoi diagram [19, 20]. This algorithm is practical in the sense that the average time complexity is $O(n)$, but is not optimal because the worst-case time complexity is $O(n^2)$.

The divide-and-conquer algorithm, on the other hand, admits the worst-case optimal $O(n \log n)$ time complexity. For a long time it was believed that the average time complexity of this algorithm is also $O(n \log n)$ [5, 9, 11, 15]. Quite recently, however, Katajainen and Koppinen [8] showed that the divide-and-conquer algorithm runs in $O(n)$ time on the average if the supporting region is divided into square subregions. Hence, to make the divide-and-conquer algorithm numerically robust is of great importance from both theoretical and practical points of view.

The application of the topology-oriented approach to the divide-and-conquer algorithm is not trivial, because, unlike the incremental algorithm, the divide-and-conquer strategy is parallel in nature; how to maintain the topological consistency in parallel is indeed a new and challenging problem. In this paper, we introduce "topological barriers," by which we can keep the data consistent. This paper is based on the first author's Bachelor's thesis [12] and a preliminary version was published in Japanese [13].

After defining the Voronoi diagram in Section 2, we review the conventional divide-and-conquer algorithm in Section 3. In Section 4, we extract the combinatorial part of the algorithm and introduce the barriers in order to guarantee its correctness in the presence of numerical errors. In Section 5 we consider how to use numerical computation, and in Section 6 we consider the time complexity and accuracy aspects theoretically. In Section 7, we describe the performance of the proposed algorithm observed in computational experiments.

2. VORONOI DIAGRAMS AND DELAUNAY DIAGRAMS

Let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of n distinct points in the plane $\mathbf{R}^2$, and let $R(P, p_i)$ be the set of points p that are closer to p_i than to any other point in P, that is,

$$R(P, p_i) = \{p \in \mathbf{R}^2 \mid d(p, p_i) < d(p, p_j),\ j \neq i\}.$$

$R(P, p_i)$ is called the *Voronoi region* for p_i. The Voronoi regions $R(P, p_1), \cdots, R(P, p_n)$ give a partition of the plane; this partition is called the *Voronoi diagram* for P and is denoted by $\mathrm{Vor}(P)$. An example of the Voronoi diagram is shown in Fig. 1a. Points in P are called *generators* for $\mathrm{Vor}(P)$, and edges and vertices of $\mathrm{Vor}(P)$ are called *Voronoi edges* and *Voronoi points,* respectively.

From $\mathrm{Vor}(P)$, we get another diagram by joining generators p_i and p_j if and only if the boundaries of $R(P, p_i)$ and $R(P, p_j)$ share a common edge. The new diagram thus obtained is called the *Delaunay diagram* for P and is denoted by $\mathrm{Del}(P)$. The Delaunay diagram for the same set of points as in Fig. 1a is shown in Fig. 1b. Edges in $\mathrm{Del}(P)$ are called *Delaunay edges.* The Delaunay edges partition the region inside the convex hull of P into connected subregions, which are called *Delaunay polygons.*

From the definition of the Voronoi diagram, we see that a Voronoi point is in equal distance from at least three surrounding generators and no other generator is closer to the Voronoi point; in other words, a Voronoi point is the center of a circle that passes through at least three generators and that contains no generator in the interior. See the circle represented by the broken line in Fig. 1a.

According to the duality, a Voronoi point in $\mathrm{Vor}(P)$ corresponds to a Delaunay polygon in $\mathrm{Del}(P)$. Hence, a subset X of P constitutes the vertices of a Delaunay polygon if and only if all the points in X lie on a common circle and all the points in $P - X$ are outside of this circle; see a broken circle in Fig. 1b. This property is called the *empty circle property.*

$\mathrm{Vor}(P)$ and $\mathrm{Del}(P)$ are planar graphs embedded in the plane, so that their complexity is of $O(n)$. Hence, once we get $\mathrm{Del}(P)$, we can transform it into $\mathrm{Vor}(P)$ in $O(n)$ time. We hereafter consider the method for constructing the Delaunay diagram.

Throughout this paper we assume that no four generators lie on a common circle. Hence, every Voronoi point in $\mathrm{Vor}(P)$ as well as in $\mathrm{Vor}(P')$ for $P' \subseteq P$ is incident to exactly three Voronoi edges; equivalently, every Delaunay polygon in $\mathrm{Del}(P)$ as well as in $\mathrm{Del}(P')$ for $P' \subseteq P$ is triangular.

This is a kind of a nondegeneracy assumption. Such an assumption might seem a deception from a practical point of view. Indeed, in the conventional framework of designing algorithms, the nondegeneracy assumption is often used in order to escape from describing algorithmic complications induced by degenerate cases, which are usually tedious. However, the nondegeneracy assumption in this paper is not for such a simplification. Recall that we consider an algorithm in the presence of numerical errors. In this world, we cannot recognize degenerate situations; even if the result of numerical computation tells us that the degeneracy takes place, it merely means that the situation is near to degeneracy. Hence, we need not consider the degenerate cases and still can obtain the complete description of the algorithm. Though it may sound paradoxical, the algorithm design becomes simpler if we move from a world without error to a world with errors [19, 20].

3. CONVENTIONAL DIVIDE-AND-CONQUER ALGORITHM

In the divide-and-conquer method, P is partitioned into two subsets, say P_1 and P_2, in such a way that the difference

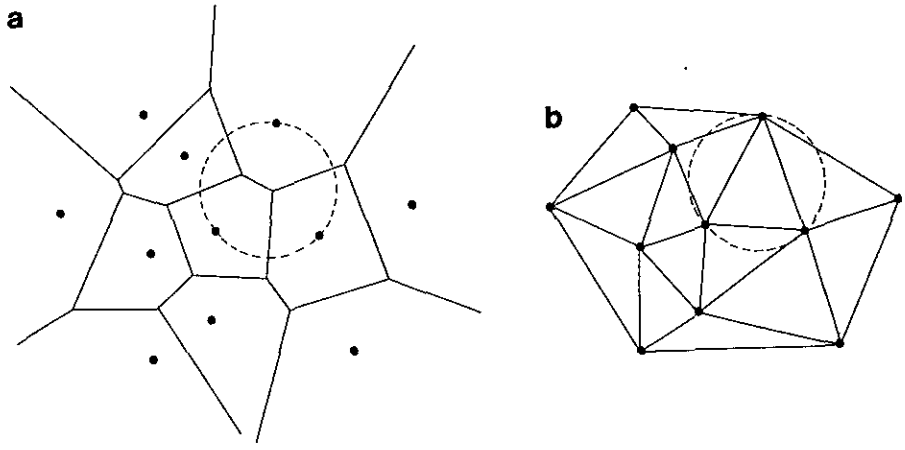

FIG. 1. (a) Voronoi diagram and (b) the associated Delaunay diagram.

of their sizes is at most one and they can be separated by a straight line [5, 9, 15]. Without loss of generality, we assume that points in P_1 have smaller x coordinates than those in P_2; we call P_1 and P_2 the *left generator set* and the *right generator set,* respectively. Each of P_1 and P_2 is recursively partitioned in the same manner until the size of a subset is small enough (e.g., 3, 4, or 5) to construct the Delaunay diagram trivially. Once we get the Delaunay diagrams for P_1 and P_2, we next merge them to get the Delaunay diagram for $P_1 \cup P_2$.

The merge process goes in the following way. Suppose that we obtained both Del(P_1) and Del(P_2) as shown in Fig. 2a. In this and subsequent figures, points in P_1 are represented by small filled circles whereas points in P_2 are represented by small empty circles. We first find $p_1 \in P_1$ and $p_2 \in P_2$ such that the upper half plane bounded by the line p_1p_2 contains $P_1 \cup P_2$, and generate the edge $\overline{p_1p_2}$ (for two points p and p', we denote by pp' the line passing through p and p', and by $\overline{pp'}$ the line segment connecting p and p'). We call the edge $\overline{p_1p_2}$ the *lower common tangent edge.* Similarly, we find $p_3 \in P_1$ and $p_4 \in P_2$ such that the lower half plane bounded by the line p_3p_4 contains $P_1 \cup P_2$, and generate the edge $\overline{p_3p_4}$; this edge is called the *upper common tangent edge.* These two new edges belong to the Delaunay edges of Del($P_1 \cup P_2$).

Next, starting with the lower common tangent edge, we remove superfluous edges from Del(P_1) and Del(P_2) and generate new Delaunay triangles for Del($P_1 \cup P_2$) one by one from the bottom upward until we reach the upper common tangent edge. A new Delaunay triangle is generated by adding an edge connecting a generator in P_1 and a generator in P_2; an edge connecting P_1 and P_2 is called a *traverse edge.*

Let e be the lower common tangent edge. To generate a Delaunay triangle incident to e, we do the following. As shown in Fig. 2a, let p_1 be the left terminal vertex of e. Let $e_1, e_2, \cdots, e_k$ be the edges in Del(P_1) that are incident to p_1 counterclockwise, and let $q_1, q_2, \cdots, q_k$ be the other terminal vertices of $e_1, e_2, \cdots, e_k$, respectively. For $i = 1, 2, \cdots, k-1$, we check whether the circle passing through p_1, q_i, q_{i+1} contains a generator in P_2, and if it does, we delete e_i from Del(P_1). For the case shown in Fig. 2a, we

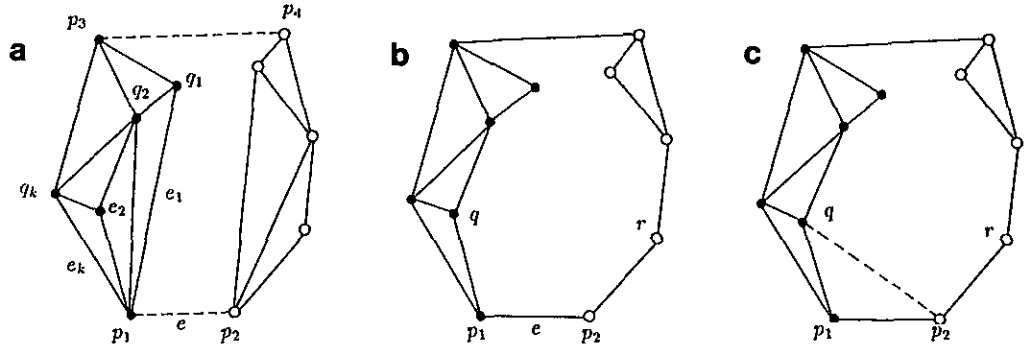

FIG. 2. Merging of the left and right Delaunay diagrams: (a) generation of the upper and lower common tangent edges; (b) deletion of superfluous edges; (c) generation of a new Delaunay edge.

delete e_1 and e_2, obtaining the diagram shown in Fig. 2b. Similarly, we check the edges incident to the right terminal vertex p_2 of e clockwise and delete edges whose associated circles contain generators in P_1; the resultant diagram is shown in Fig. 2b.

At this stage let $\overline{p_1q}$ be the counterclockwise next edge incident to p_1 and $\overline{p_2r}$ be the clockwise next edge incident to p_2; see Fig. 2b. The four points p_1, p_2, r, q are the vertices of a convex quadrilateral, and they lie on the boundary of the quadrilateral in this order. Hence, the circle passing through p_1, p_2, r contains q in the interior if and only if the circle passing through p_1, p_2, q does not contain r. We check whether the circle passing through p_1, p_2, r contains q. If it does (in this case, the circle passing through p_1, p_2, q does not contain r), we generate the traverse edge $\overline{p_2q}$, as shown in Fig. 2c. Otherwise, we generate the traverse edge $\overline{p_1r}$.

Thus, we generate a new traverse edge. We rename this new edge as e and repeat the same procedure until we get a Delaunay triangle containing the upper common tangent edge; see standard references [5, 9, 15] for details.

This algorithm works when the numerical computation is precise. In actual computation, however, the algorithm sometimes fails due to inconsistency caused by numerical errors. An example of such a failure is depicted in Fig. 3. This figure shows the stage where the lower common tangent edge $\overline{p_1p_2}$ was found, and the superfluous edges incident to p_1 or p_2 were deleted from $\mathrm{Del}(P_1)$ or $\mathrm{Del}(P_2)$. The next task is to choose $\overline{p_1r}$ or $\overline{p_2q}$ as a traverse edge. As shown in this figure, however, p_1 and p_2 come very close to each other, so that the computations of the circle passing through p_1, p_2, q and the circle passing through p_1, p_2, r are unstable. The result of these computations may tell that both of the two circles are nonempty, as shown by the dotted circles. Then we cannot choose the next traverse edge; the algorithm is stuck.

What will happen in the computer program depends on the implementation of the algorithm. If the empty circle test is done for both of the circles, the program will find inconsistency and will terminate with an error message. If the empty circle test is done for only one circle and it is found empty, then the program believes that the other circle is not empty and goes to the next step of the algorithm; this may destroy the data or generate another inconsistency in a later stage of the processing. As is easily understood by this example, the conventional divide-and-conquer algorithm is not necessarily valid in imprecise arithmetic.

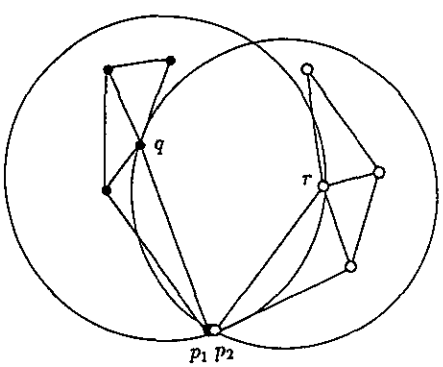

FIG. 3. Inconsistent situation due to numerical error.

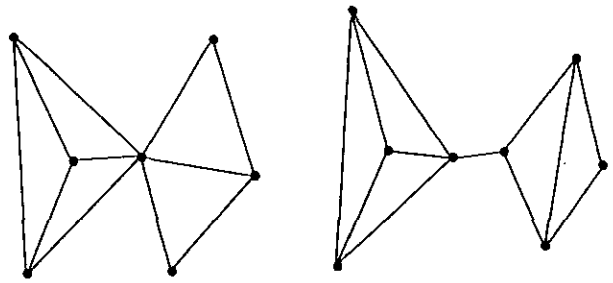

FIG. 4. Graphs which are not isomorphic to any Delaunay diagram.

4. COMBINATORIAL ABSTRACTION OF THE ALGORITHM

Here we abstract the divide-and-conquer algorithm in order to describe the fundamental part of the algorithm in terms of combinatorial computation only. From a combinatorial point of view, the Delaunay diagram is a planar graph G embedded in the plane. Let us assume that $|P| \geq 3$, where $|P|$ denotes the number of elements of the set P. Then, G satisfies the following topological properties:

(C1) G is simple (i.e., any cycle in G has at least three edges).

(C2) G is connected.

(C3) Each vertex of G has at least two edges.

(C4) The connected regions bounded by G, except for the outermost unbounded one, are bounded by exactly three edges.

(C5) The boundary of the outermost unbounded region is a simple cycle (i.e., when we move along the boundary, we visit every edge and every vertex on it exactly once until we reach the start vertex).

The property (C4) holds because of the nondegeneracy assumption. The property (C5) holds because the outermost boundary corresponds to the convex hull of the set of generators; hence the graphs shown in Fig. 4 are not allowed. In what follows, we regard the combinatorial structure of the Delaunay diagram as the embedded graph G satisfying (C1) through (C5). Vertices on the boundary of the outermost unbounded region are called *outermost vertices.*

Note that the properties (C1)–(C5) are purely combina-

torial; these properties are described only in combinatorial terms and do not refer to numerical values. Once we get such combinatorial properties, we can describe the basic part of the algorithm by combinatorial terms in such a way that at least these properties are preserved. For our particular problem, we can do this by introducing some barriers as will be shown in the following.

If $|P| = 3$ or 4 or 5, it is not difficult to generate embedded graph G that satisfies (C1) through (C5) and that "seems" closest to the correct Delaunay diagram for P, where "seems" means that numerical computation tells us so. Since numerical computation contains errors, G may not be the graph of the correct Delaunay diagram. However, we can guarantee at least (C1)–(C5); all of these properties are of combinatorial nature, and hence can be checked without numerical errors.

Now let us assume that G_1 and G_2 are the embedded graph that satisfy (C1)–(C5) and are supposed to be the graphs of $\mathrm{Del}(P_1)$ and $\mathrm{Del}(P_2)$, respectively. Then, we want to merge G_1 and G_2 to get new graph G that is supposed to be the graph of $\mathrm{Del}(P_1 \cup P_2)$. This task can be described in terms of combinatorial computation by the following algorithm; what we actually want to do is described as comments in parentheses (we can understand the algorithm more easily if we recall the example given in Fig. 2).

ALGORITHM 1 (COMBINATORIAL SKELETON OF THE MERGE PROCESS)

Input: Left and right generator sets P_1 and P_2, and two embedded graphs G_1 and G_2 having P_1 and P_2, respectively, as the vertex sets and satisfying (C1)–(C5).

Output: Embedded graph G that has $P_1 \cup P_2$ as the vertex set and that satisfies (C1)–(C5).

Procedure:

1. Generate two edges $\overline{p_1p_2}$ and $\overline{p_3p_4}$ where p_1 and p_3 are outermost vertices of G_1 and p_2 and p_4 are outermost vertices of G_2. (Comment: we want to generate the lower and upper common tangent edges as $\overline{p_1p_2}$ and $\overline{p_3p_4}$.)
2. $e \leftarrow \overline{p_1p_2}$.
3. Delete zero or more consecutive edges incident to the left terminal vertex of e counterclockwise starting with the edge next to e. (Comment: we want to delete from G_1 those edges that are not Delaunay edges for $P_1 \cup P_2$.)
4. Delete zero or more consecutive edges incident to the right terminal vertex of e clockwise starting with the edge next to e. (Comment: we want to delete from G_2 those edges that are not Delaunay edges for $P_1 \cup P_2$.)
5. If the region immediately above e is triangular and its boundary contains both e and $\overline{p_3p_4}$, report the current embedded graph as G and stop. (Comment: we are supposed to reach the upper common tangent edge.)
6. Generate a traverse edge e' adjacent to e in such a way that e and e' and another edge form a new triangle immediately above e. (Comment: we want to create a new Delaunay triangle immediately above e.)
7. $e \leftarrow e'$, and go to Step 3.

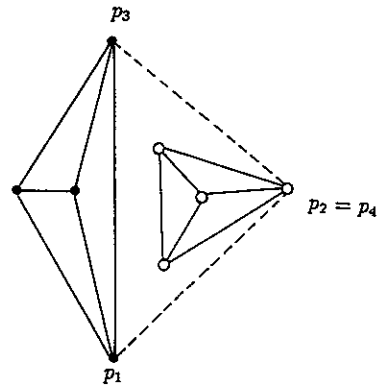

FIG. 5. Upper and lower common tangent edges with a common right terminal vertex.

Note that, if we ignore the comments in the parentheses, Algorithm 1 is purely combinatorial so that every step can be done without any worry about numerical errors. As for the cost of this, however, Steps 1, 3, 4, and 6 are ambiguous; there are two or more choices in doing the steps. In order to resolve the ambiguity we need to employ numerical computation. However, careless use of numerical computation will generate inconsistency. In order to prevent such inconsistency, we will introduce topological barriers. These barriers are described in combinatorial terms and can guarantee both that Algorithm 1 is carried out without inconsistency and that the output G satisfies (C1)–(C5).

We introduce a little more terminology. The edge e, which is initially set as the lower common tangent edge in Step 2 and later replaced by the newest traverse edge in Step 7, is called the *base edge*. Suppose that we are at the end of Step 4. The edge that is incident to the left [respectively, right] terminal vertex of the base edge and that is counterclockwise [respectively, clockwise] next to the base edge is called the *left neighbor edge* [respectively, *right neighbor edge*] of the base edge. In Fig. 2b, for example, the base edge is e, the left neighbor edge is $\overline{p_1q}$, and the right neighbor edge is $\overline{p_2r}$.

The first barrier is for Step 1.

Barrier 1. The vertices chosen in Step 1 should satisfy either $p_1 \neq p_3$ or $p_2 \neq p_4$.

Either $p_1 = p_3$ or $p_2 = p_4$ is allowed, as shown in Fig. 5, but not both because P_1 and P_2 have at least three vertices and they are nondegenerate.

It seems that Barrier 1 is the only barrier we can place for Step 1. This can be understood in the following way. Suppose that we chose p_1, p_2, p_3, p_4 arbitrarily, provided that Barrier 1 is not violated. Then we can perturb the

locations of points in P_1 and P_2 in such a way that $\overline{p_1p_2}$ and $\overline{p_3p_4}$ are the lower and upper common tangent edges for the new locations of the points. Such perturbation of the input might be considered equivalent to the effect of numerical errors. Hence, it is difficult to place a stronger condition than Barrier 1. Because of similar arguments, we can see that other barriers which we will place hereafter are also difficult to be replaced by stronger ones.

In Steps 3 and 4, we want to delete from G_1 and G_2 those edges that do not belong to $\mathrm{Del}(P_1 \cup P_2)$. For this purpose we can place the following barriers.

Barrier 2. An edge on the boundary of the convex hull of P_1 or P_2 should not be deleted in Step 3 or 4, respectively, if the resultant graph becomes disconnected.

If the graph becomes disconnected, we cannot connect it again by Algorithm 1 and hence we can never complete the Delaunay diagram. Suppose, for example, that, as shown in Fig. 2c, we generated traverse edges $\overline{p_1p_2}$, $\overline{p_2q}$ and now we are in Step 4 with the base edge $\overline{p_2q}$. Then Barrier 2 says that edge $\overline{p_2r}$ in the figure should not be deleted.

To test whether the resultant graph is disconnected is not easy in general. In our case, however, we can test it easily because an edge we are concerned about is on the boundary of the convex hull of P_1 or P_2. Without loss of generality, suppose that we concentrate our attention on an edge e on the boundary of the left generator set P_1. Let us define the direction of all the edges on the boundary of P_1, in such a way that they surround the convex hull counterclockwise. Then the deletion of e makes the resultant graph disconnected if and only if e comes in the left terminal vertex of the base edge.

Barrier 3. An edge should not be deleted in Step 3 or 4 if the edge is adjacent to traverse edges at both of the terminal vertices.

Suppose that the edges e_1, e_2, e_3 in Fig. 6a are traverse

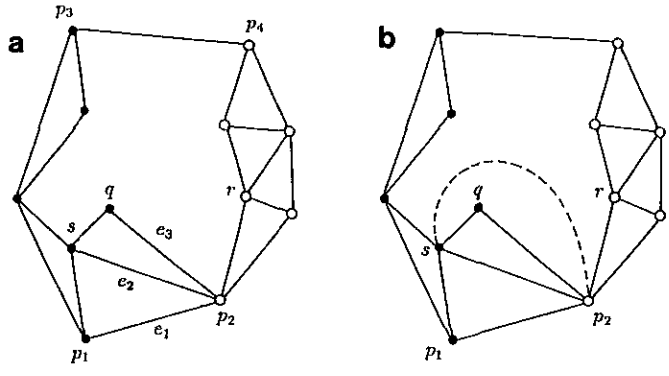

FIG. 6. Barriers 3, 4, and 5: (a) edge $\overline{sq}$ should not be deleted; (b) broken edge $\overline{sp_2}$ should not be generated.

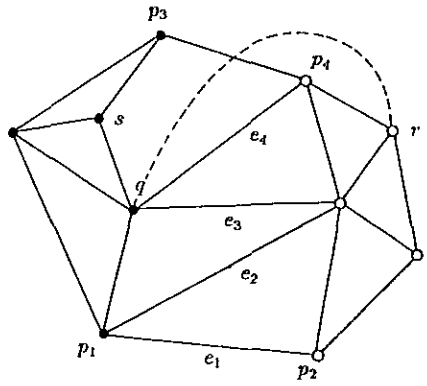

FIG. 7. Barrier 6.

edges generated in Algorithm 1, and that the present base edge is e_3. (Readers may note that this figure is far different from the correct merge procedure of the Delaunay diagram. Recall, however, that we consider the algorithm in imprecise arithmetic and that we concentrate our attention on the embedded graph structure; the locations of the points in this figure should be considered inexact.) Then Barrier 3 says that the edge $\overline{qs}$ in the figure should not be deleted; if $\overline{qs}$ is deleted, we eventually have to generate an edge connecting two vertices in P_2, which cannot happen in the correct Delaunay diagram.

Barrier 4. The edge $\overline{p_3p_4}$ created in Step 1 should not be deleted in Step 3 or 4.

This barrier is necessary, because the upper common tangent edge as well as the lower common tangent edge is a Delaunay edge of $\mathrm{Del}(P_1 \cup P_2)$.

For Step 6 we place two barriers.

Barrier 5. An edge should not be generated in Step 6 if the vertices are already connected by an edge.

Consider the situation in Fig. 6a again; suppose that the base edge is e_3 and we are at Step 6. Then there are two choices for a next traverse edge, $\overline{p_2s}$ or $\overline{qr}$. However, $\overline{p_2s}$ is impossible because, if $\overline{p_2s}$ is generated, the two vertices p_2 and s are connected by two traverse edges as shown in Fig. 6b. Therefore the only possible choice is the traverse edge $\overline{qr}$. This is what Barrier 5 implies.

Barrier 6. An edge should not be generated in Step 6 if it crosses the upper common tangent edge $\overline{p_3p_4}$.

This barrier is necessary because the Delaunay diagram gives a planar graph. As shown in Fig. 7, suppose that we have generated traverse edges e_1, e_2, e_3, e_4 and we are at Step 6 with the base edge e_4. Since the right neighbor edge is the upper common tangent edge $\overline{p_3p_4}$, we cannot generate the traverse edge $\overline{qr}$ represented by the broken line in this figure; it crosses the upper common tangent edge. The

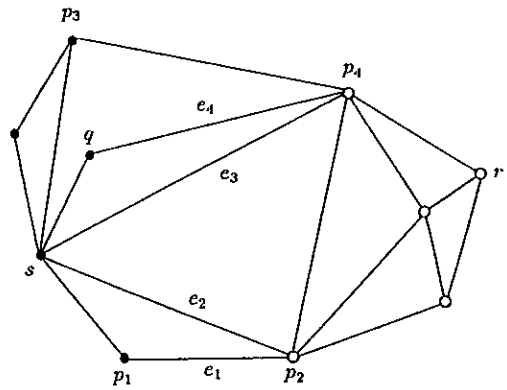

FIG. 8. Conflict between Barriers 5 and 6.

other traverse edge $\overline{p_4 s}$ is the only possible choice at this situation.

Barriers 5 and 6 can conflict with each other. An example of such a situation is shown in Fig. 8. Suppose that we have generated traverse edges e_1, e_2, e_3, e_4, and now we are at Step 6 with the base edge e_4. Since the vertices s and p_4 are already connected by e_3, we cannot generate a new traverse edge $\overline{sp_4}$ (Barrier 5). Since the right neighbor edge is the upper common tangent edge, we cannot generate the other traverse edge $\overline{qr}$ either (Barrier 6).

Such a conflict comes from the situation where numerical computation is far from reliable. Hence, in order to resolve the conflict, we arbitrarily triangulate the remaining region and terminate the merge procedure. Let us call this exceptional procedure the *conflict–resolution triangulation.* Note that any conflict–resolution triangulation does not violate conditions (C1)–(C5).

THEOREM 1. *Suppose that Algorithm* 1 *is done in such a way that Barriers* 1, 2, $\cdots$, 6 *are not violated and that the conflict–resolution triangulation is employed in the case where Barriers* 5 *and* 6 *conflict with each other. Then, Algorithm* 1 *is always carried out to the end and the output G satisfies* (C1)–(C5).

Proof. Step 1 can be done without violating Barrier 1, because both G_1 and G_2 have three or more vertices. Steps 3 and 4 can be done without violating Barriers 2 and 3, because "nothing is done" is one of plausible actions for these steps. Step 6 can be done because, even if any choice of the traverse edge violates the barriers, we can employ the conflict–resolution triangulation. Steps 2, 5, and 7 can be done obviously. Thus, all the steps in the algorithm can be done without any inconsistency.

The output G satisfies (C1), because G_1 and G_2 are simple and we generate no cycle of length two or less (Barrier 5). G satisfies (C2), because G_1 and G_2 are connected, they are changed into one connected graph in Step 1, and the resultant connected graph remains connected (Barrier 2). G satisfies (C3) and (C4), because Algorithm 1 terminates only when all the connected regions, except for the outermost unbounded one, become triangular. To prove that G satisfies (C5), let $(p_3, q_1, q_2, \cdots, q_k, p_1)$ be the counterclockwise sequence of outermost vertices from p_3 to p_1 in G_1, and let $(p_2, r_1, r_2, \cdots, r_l, p_4)$ be the counterclockwise sequence of outermost vertices from p_2 to p_4 in G_2. Then, the cyclic sequence $(p_3, q_1, q_2, \cdots, q_k, p_1, p_2, r_1, r_2, \cdots, r_l, p_4, p_3)$ forms the boundary of the outermost unbounded region throughout the algorithm because of Barriers 1, 2, 4, and 6. Hence, the output G satisfies (C5). Q.E.D.

5. HOW TO USE NUMERICAL COMPUTATION

Theorem 1 guarantees that Algorithm 1 is always carried out without inconsistency and the output is topologically consistent in the sense it satisfies (C1)–(C5). However, the algorithm still has ambiguity; Barriers 1–6 are not enough to specify the behavior of the algorithm uniquely. Indeed, there are many possible ways for choosing edges in Steps 1 and 6, and for deleting edges in Steps 3 and 4. Our primal goal is to construct the Delaunay diagram, and in order to choose the edges to be generated or to be deleted that seem most promising to lead to the Delaunay diagram, we use numerical computation.

The most important numerical computation is the empty circle test required in Steps 3, 4, and 6. Let the coordinates of vertex p_i be (x_i, y_i), and let us define

$$H(p_i, p_j, p_k, p_l) = \det \begin{bmatrix} 1 & x_i & y_i & x_i^2 + y_i^2 \\ 1 & x_j & y_j & x_j^2 + y_j^2 \\ 1 & x_k & y_k & x_k^2 + y_k^2 \\ 1 & x_l & y_l & x_l^2 + y_l^2 \end{bmatrix}.$$

Suppose that we take the right-hand coordinate system and that p_i, p_j, p_k form the counterclockwise sequences of the vertices of a triangle. Then the vertex p_l is inside the circumcircle of the triangle $p_i p_j p_k$ if and only if $H(p_i, p_j, p_k, p_l) < 0$ (this can be understood if it is noted that the equation $H(p_i, p_j, p_k, p) = 0$ with a variable point p represents the circle passing through p_i, p_j, and p_k) [19].

Suppose that $\overline{p_i p_j}$ is the base edge ($p_i \in P_1$ and $p_j \in P_2$) and the left neighbor edge is $\overline{p_i p_k}$. Then in Step 3 the edge $\overline{p_i p_k}$ is deleted if and only if (i) the deletion of $\overline{p_i p_k}$ does not violate Barriers 2 and 3 and (ii) $H(p_i, p_j, p_k, p_l) < 0$ for at least one vertex p_l in G_2. The deletion of an edge in Step 4 is done similarly.

Next, suppose that $\overline{p_i p_j}$ is the base edge ($p_i \in P_1$ and $p_j \in P_2$) and $\overline{p_i p_k}$ is the left neighbor edge and $\overline{p_j p_l}$ is the right neighbor edge. If one of the candidates $\overline{p_i p_l}$ and $\overline{p_j p_k}$ of the traverse edge is inhibited by Barrier 5 or 6, we

choose the other one. If both candidates are plausible, we choose $\overline{p_i p_l}$ if $H(p_i, p_j, p_k, p_l) < 0$, and we choose $\overline{p_j p_k}$ otherwise.

Employing numerical computation in this way, we can choose the edges to be generated or to be deleted that seem to lead to the construction of the correct Delaunay diagram. This is the topology-oriented algorithm we propose in this paper.

6. EFFICIENCY AND ACCURACY

In this section we consider time complexity of Algorithm 1. Note that we are interested in the behavior of the algorithm in the presence of numerical errors. Actually the time complexity depends on the amount of errors. The next theorem tells the complexity in the poorest precision.

THEOREM 2. *No matter how poor the numerical precision may be, Algorithm* 1 *with Barriers* 1, 2, $\cdots$, 6 *runs in* $O(n^2)$ *time, where* $n = |P_1 \cup P_2|$.

Proof. First, suppose that we do not employ Barriers 1, 2, $\cdots$, 6. Then a standard method [15] for finding the lower and upper common tangents for the convex hulls of P_1 and P_2 scans the boundaries of the two convex hulls at most twice, once clockwise and once counterclockwise. Hence Step 1 terminates in $O(n)$. Steps 2, 5, 6, 7 are done in $O(1)$. Steps 3 and 4 can be done in $O(n)$ because there are only $O(n)$ edges in any planar graph having the vertex set P_1 or P_2. Steps 3 through 7 are repeated at most $O(n)$ times, because one traverse edge is created at each repetition (which will never be deleted) and any planar graph with the vertex set $P_1 \cup P_2$ has at most $O(n)$ edges. Hence Algorithm 1 runs in $O(n^2)$.

Next, suppose that we employ Barriers 1, 2, $\cdots$, 6. Barrier 1 is guaranteed in $O(1)$ additional time, and hence the time complexity of Step 1 does not increase. In order to guarantee Barrier 2 in $O(1)$ time, we assign the counterclockwise direction to the outermost edges of G_1 and the clockwise direction to the outermost edges of G_2. Then, Barrier 2 can be paraphrased as "An edge on the boundary of the convex hull of P_1 or P_2 should not be deleted if it comes in the terminal vertex of the base line." Hence Barrier 2 is guaranteed in $O(1)$ time. Barrier 3 is guaranteed in $O(1)$ time, if we mark all the vertices incident to traverse edges. Barrier 4 is guaranteed in $O(1)$ time. Hence, the time complexity of Steps 3 and 4 does not increase when we employ the barriers. Barrier 5 is guaranteed in $O(n)$ time if we check all the traverse edges naively. Barrier 6 is guaranteed in $O(1)$ time; indeed we need to check the condition only when p_3 or p_4 (endpoints of the upper common tangent edge) coincides with one of the endpoints of the base edge. Hence, the additional cost for guaranteeing Barriers 5 and 6 in Step 6 is $O(n)$. Thus the employment of Barriers 1, 2, $\cdots$, 6, requires $O(n)$ additional preprocessing at the beginning of Algorithm 1 and $O(n)$ additional processing in Step 6; time complexity does not increase in other steps. Consequently, the total time complexity is still $O(n^2)$. Q.E.D.

THEOREM 3. *No matter how poor the numerical precision may be, the total time complexity of the divide-and-conquer method for* Del(P) *in which the merge process is done by Algorithm* 1 *together with Barriers* 1, 2, $\cdots$, 6 *is* $O(n^2)$, *where* $n = |P|$.

Proof. Let $T(n)$ be the total time of the divide-and-conquer method. Then we get

$$T(n) = 2T(n/2) + cn^2$$

for some constant c. Indeed, the first term on the right-hand side of this equation corresponds to the cost for recursively constructing Del(P_1) and Del(P_2), and the second term corresponds to the cost for Algorithm 1. From this equation we get $T(n) = O(n^2)$. Q.E.D.

Theorems 2 and 3 guarantee the $O(n^2)$ time complexity of the divide-and-conquer method in the presence of any large numerical errors. From the practical point of view, however, this time complexity is not very important, because the output may be far from the correct Delaunay triangulation if numerical errors are large. (Indeed we can show a better bound of the time complexity, but we skip it because of its practical unimportance.)

What we are interested in is the time complexity in the case where numerical errors are small. Roughly speaking, the additional cost for checking the barriers does not increase the $O(n \log n)$ time complexity of the conventional method (which is designed in precise arithmetic). To show this, we give the next computation model of small errors.

Let $\mathbf{x}$ be the input numerical data to an algorithm, and let $f(\mathbf{x})$ be any polynomial in $\mathbf{x}$. In the case of the Delaunay diagram, $\mathbf{x}$ is the vector consisting of all the coordinates of the input points: $\mathbf{x} = (x_1, y_1, x_2, y_2, \cdots, x_n, y_n)$, and an example of a polynomial is $H(p_i, p_j, p_k, p_l)$. Let, furthermore, $\tilde{f}(\mathbf{x})$ denote the computed value of $f(\mathbf{x})$ in a given arithmetic precision. Let sign(α) be the sign of real number α: sign(α) = 1 if $\alpha > 0$, and sign(α) = -1 if $\alpha < 0$. Now, let us place the next assumption.

Assumption 1 (*Small-Error Assumption*). If $f(\mathbf{x}) \neq 0$, $\text{sign}(\tilde{f}(\mathbf{x})) = \text{sign}(f(\mathbf{x}))$, whereas if $f(\mathbf{x}) = 0$, $\text{sign}(\tilde{f}(\mathbf{x}))$ is either 1 or -1 at random.

Assumption 1 intuitively means that the sign of a polynomial is judged correctly if the associated input is nondegenerate, whereas it may not be if the input is degenerate.

THEOREM 4. *If the input graphs* G_1 *and* G_2 *are the correct Delaunay diagrams* Del(P_1) *and* Del(P_2), *Algorithm*

1 *runs in* $O(n)$ *time under Assumption* 1, *where* $n = |P_1 \cup P_2|$.

Proof. Under Assumption 1, Algorithm 1 constructs the correct Delaunay diagram. Indeed, if the input is nondegenerate, all the topological judgements are done correctly, and hence the correct topological structure of the Delaunay diagram is obtained. If the input is degenerate, the output of Algorithm 1 is a triangulation with the empty circle property; that is, the circumcircle of any triangle contains no point in $P_1 \cup P_2$ in the interior. Hence, by definition the output is one of many possible degenerate Delaunay diagrams. Thus, if we ignore Barriers 1, 2, $\cdots$, 6, Algorithm 1 under Assumption 1 is nothing but the merge process of the conventional divide-and-conquer method and consequently runs in $O(n)$ time [5, 9, 15].

Hence, what we need to show is that the tests for Barriers, 1, 2, $\cdots$, 6 do not increase the time complexity. As we have already seen in the proof of Theorem 2, the tests for Barriers 1, 2, 3, 4, 6 require only $O(1)$ time. For Barrier 5, we generate a table of size $|P_1| \times |P_2|$ whose component $t[i, j]$ is defined in such a way that $t[i, j] = 1$ if v_i $(\in P_1)$ and v_j $(\in P_2)$ are connected by a traverse edge, and $t[i, j] = 0$ otherwise. Then we can check whether $p \in P_1$ and $q \in P_2$ are already connected by a traverse edge in $O(1)$ time. We need $O(n)$ preprocessing for Barriers 2 and 5, but the preprocessing is done only once at the beginning of Algorithm 1. Hence, the tests for Barriers 1, 2, $\cdots$, 6, do not increase the time complexity. Q.E.D.

THEOREM 5. *Under Assumption* 1, *the total time complexity of the divide-and-conquer method for* Del(P) *in which the merge process is done by Algorithm* 1 *together with Barriers* 1, 2, $\cdots$, 6 *is* $O(n \log n)$, *where* $n = |P|$.

Proof. Let the total time complexity be $T(n)$. From Theorem 4, we get $T(n) = 2T(n/2) + cn$ for some constant c and hence we obtain $T(n) = O(n \log n)$. Q.E.D.

Katajainen and Koppinen [8] showed that, for generators uniformly distributed in a square, the divide-and-conquer method runs in $O(n)$ time on the average if we recursively divide the generators to left and right halves and to upper and lower halves alternatingly. The same average time complexity is obtained by our method under Assumption 1.

We are interested in the behavior of our algorithm in the case where the arithmetic precision is neither very high nor very low. However, theoretical treatment is not easy. The main reason for this is that we do not know any necessary and sufficient condition for a planar embedded graph to be isomorphic to a Delaunay diagram; only some necessary conditions and some sufficient conditions are known [2, 3, 17]. Hence, at present we cannot define a "distance" between the output of our method and the correct Delaunay diagram.

Usually the distance between the correct answer and the output computed in inexact arithmetic is defined as the amount of perturbation of input data so as to make the output coincide with the correct answer of the perturbed version of the problem. For our problem, however, such perturbation does not necessarily work because sometimes the output is not isomorphic to any Delaunay diagram. Thus, it is a big open problem for the future to find a necessary and sufficient condition for a graph to be isomorphic to a Delaunay diagram, and to define the distance between the output obtained in finite precision and the correct answer.

Finally, let us mention the precision that is necessary and

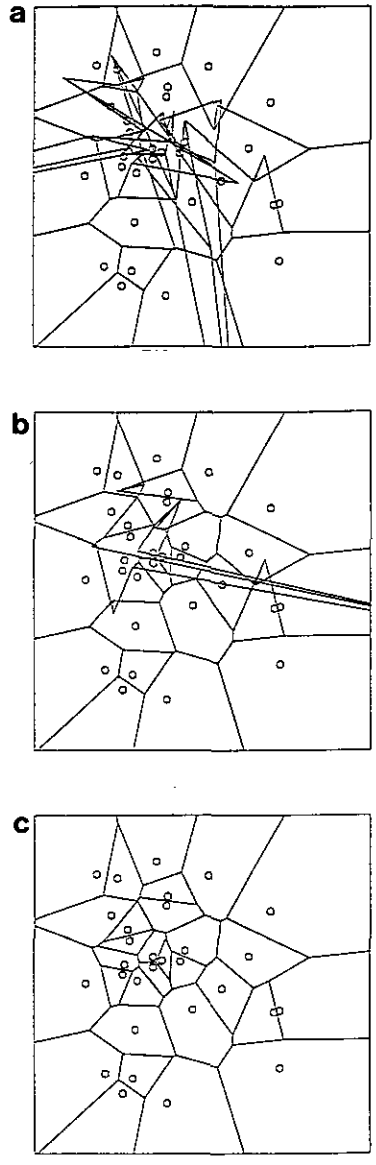

FIG. 9. Output of the algorithm executed in poor precision: (a) 2-bit precision; (b) 4-bit precision; (c) 6-bit precision.

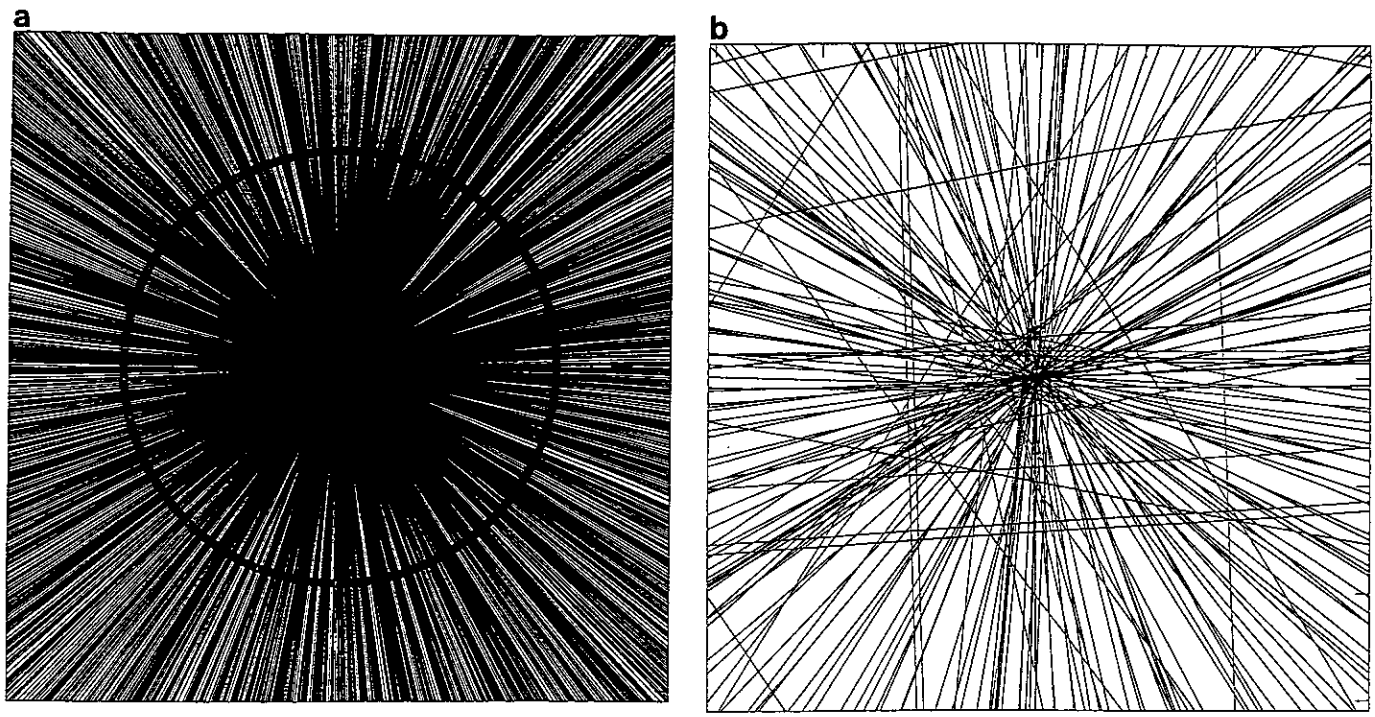

FIG. 10. Output for 2000 generators randomly placed on a common circle: (a) output in normal scale; (b) central portion magnified by 10^4.

sufficient to make Assumption 1 valid. For this purpose, suppose that each of the coordinates of the input generators is given in k-bit integers excluding the sign bit; that is, the input $\mathbf{x} = (x_1, y_1, \cdots, x_n, y_n)$ is an integer vector satisfying

$$-2^k < x_i, y_i < 2^k \quad \text{for } i = 1, 2, \cdots, n.$$

The precision necessary to make the computed sign $\text{sign}(\tilde{f}(\mathbf{x}))$ correct becomes higher as the degree of the polynomial $f(\mathbf{x})$ becomes larger. The highest degree polynomial required in Algorithm 1 is $H(p_i, p_j, p_k, p_l)$. From the Hadamard inequality (which says that the absolute value of the determinant of a matrix is not larger than the multiplication of the lengths of all column vectors), we get

$$\begin{aligned}|H(p_i,p_j,p_k,p_k)| &< \sqrt{1^2+1^2+1^2+1^2}\\ &\quad \sqrt{2^{2k}+2^{2k}+2^{2k}+2^{2k}}\\ &\quad \times \sqrt{2^{2k}+2^{2k}+2^{2k}+2^{2k}}\\ &\quad \sqrt{2^{2(2k+1)}+2^{2(2k+1)}+2^{2(2k+1)}+2^{2(2k+1)}}\\ &= 2 \times 2^{k+1} \times 2^{k+1} + 2^{2k+2} = 2^{4k+5}.\end{aligned}$$

Hence, the sign of $H(p_i, p_j, p_k, p_l)$ is judged correctly if it is computed in $4k + 5$ bits excluding the sign bit.

Thus we get the next theorem.

THEOREM 6. *Suppose that each coordinate of the generators is given by a $k + 1$-bit integer (1 bit for the sign and k bits for the absolute value). Then Assumption 1 holds if computation is done in $4k + 6$ bits (1 bit for the sign and $4k + 5$ bits for the absolute value).*

Assumption 1, Theorem 5, and Theorem 6 altogether imply that if the input coordinates are exact and nondegenerated and if sufficiently high precision arithmetic is used, the algorithm runs in $O(n \log n)$ time and gives the correct output. In other words, if we can use sufficiently high precision, the barriers introduced in our algorithm do not increase the time complexity; our algorithm has the same time complexity as the (nonrobust) conventional algorithms.

Note, however, that our algorithm runs in any finite precision though the output is an approximation of the correct answer if the precision is not sufficient. Hence, our algorithm is portable; we need not worry about the precision when we transfer the algorithm from one computer to another.

7. COMPUTATIONAL EXPERIMENTS

We constructed a computer program of the proposed algorithm and observed the performance of the algorithm. The program was written in C language and was about 2700 lines long. The experiments were done by the Fujitsu large computer FACOM 380 with UNIX System V at the Educational Computer Center of the University of Tokyo.

In order to see the behavior of the algorithm in imprecise arithmetic, we executed the program in many different

precisions in computation. An example of the behavior of the program for 20 generators is shown in Fig. 9, where Figs. 9a, 9b, and 9c are the outputs in the cases where numerical computation was done in two-bit precision, four-bit precision, and six-bit precision, respectively. From this figure, we can see at least empirically that (i) no matter how poor the precision may be, the algorithm carries its task, ending up with some output, (ii) the output is topologically consistent in the sense that it satisfies (C1)–(C5), and (iii) the output converges to the true Voronoi diagram as the precision becomes higher.

Figure 10a is the output in single-precision floating-point computation for 2000 generators randomly placed on a common circle. An input of this kind is highly degenerate and cannot be treated by conventional algorithms. Our algorithm gives the output shown here, which seems a correct Voronoi diagram as long as we see it in this scale. However, if we magnify the central portion by 10^4, we get the diagram shown in Fig. 10b; we can see great disturbance in this microstructure. Such disturbance seems inevitable because the computation was done only in single precision. What we want to emphasize is that the algorithm gave a globally correct output even though topological structures were midjudged due to numerical errors.

Figure 11 shows the CPU time required by the program. We implemented the algorithm in two different ways. One is a conventional divide-and-conquer, where the generator set is partitioned into left and right halves recursively. The theoretical time complexity of this type is $O(n \log n)$, both in the worst case and in the average case. The other is the Katajainen–Koppinen divide-and-conquer, where the generator set is partitioned into the left and right halves and the upper and lower halves alternatingly. The theoretical average time complexity of this type is $O(n)$ for randomly placed generators. In this experiment the generators were placed randomly in the unit square, and computation was done in single precision. The horizontal axis in Fig. 11 represents the number of generators in logarithmic scale, and the vertical axis represents the average CPU time per generator in linear scale. The small triangles correspond to the conventional type and the small circles correspond to the Katajainen–Koppinen type. From this figure, we can see that the linearity of the time complexity is almost preserved even though we introduced the topological barriers.

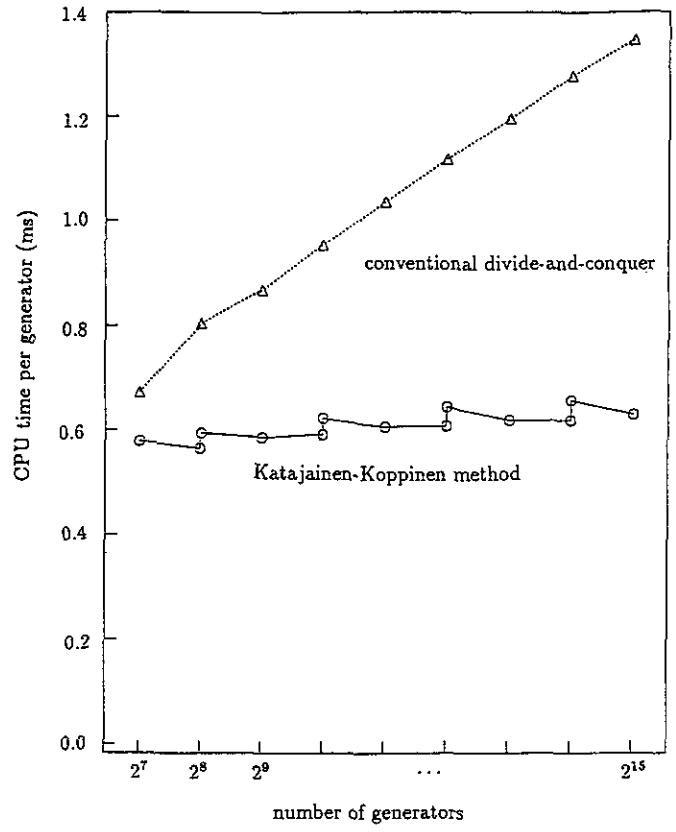

FIG. 11. Comparison of the CPU time between the conventional divide-and-conquer method and the Katajainen–Koppinen method; the horizontal axis represents the number of generators in logarithmic scale, and the vertical axis represents the CPU time required per generator.

8. CONCLUDING REMARKS

We have presented a numerically robust algorithm for constructing the Voronoi diagram in a divide-and-conquer manner. In this algorithm we place topological barriers and thus can prevent inconsistency due to numerical errors. Our algorithm (i) is robust in the sense that it always gives a topologically consistent output, (ii) is correct in the sense that the output converges to the true diagram as the precision becomes higher, (iii) is simple in the sense that exceptional branches for degeneracy are not required, and (iv) is optimal in the sense that the linear average time complexity is achieved.

ACKNOWLEDGMENT

The authors express their thanks to the anonymous referees for their valuable comments.

REFERENCES

1. M. Benouamer, D. Michelucci, and B. Peroche, Error-free boundary evaluation using lazy rational arithmetic—A detailed implementation, in *Proceedings of the 2nd Symposium on Solid Modeling and Applications, Montreal, 1993,* pp. 115–126.
2. M. B. Dillencourt, Toughness and Delaunay triangulations, *Discrete Comput. Geom.* **5,** 1990, 575–601.
3. M. B. Dillencourt, Reliazability of Delaunay triangulations, *Inform. Process. Lett.* **33,** 1990, 283–287.
4. L. Guibas, D. Salesin, and J. Stolfi, Epsilon geometry—Building robust algorithms from imprecise computations, in *Proceedings of*

the 5th ACM Annual Symposium on Computational Geometry, Saarbrücken, 1989, pp. 208–217.

5. L. Guibas and J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *ACM Trans. Graphics* **4,** 1985, 74–123.
6. C. M. Hoffmann, *Geometric and Solid Modeling,* Morgan Kaufmann, San Mateo, CA, 1989.
7. C. M. Hoffmann, The problems of accuracy and robustness in geometric computation, *Computer* **22**(3), 1989, 31–41.
8. J. Katajainen and M. Koppinen, Constructing Delaunay triangulations by merging buckets in quadtree order, *Fundament. Inform.* **11,** 1988, 275–288.
9. D. T. Lee and B. J. Schachter, Two algorithms for constructing a Delaunay triangulation, *Int. J. Comput. Inform. Sci.* **9,** 1980, 219–242.
10. V. Milenkovic, Verifiable implementations of geometric algorithms using finite-precision arithmetic, *Artif. Intell.* **37,** 1988, 377–401.
11. T. Ohya, M. Iri, and K. Murota, Improvements of the incremental method for the Voronoi diagram with computational comparisons of various algorithms, *J. Oper. Res. Soc. Jpn.* **27,** 1984, 306–336.
12. Y. Oishi, Numerical stabilization of the divide-and-conquer algorithm for computing Voronoi diagrams. Dissertation for the Bachelor's Degree, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 1990. [in Japanese]
13. Y. Oishi and K. Sugihara, Numerically robust divide-and-conquer algorithm for constructing Voronoi diagrams, *J. Inform. Process. Soc. Jpn.* **32,** 1991, 709–720. [in Japanese]
14. T. Ottmann, G. Thiemt, and C. Ullrich, Numerical stability of geometric algorithms, in *Proceedings of the 3rd ACM Annual Symposium on Computational Geometry, Waterloo, 1987,* pp. 119–125.
15. F. P. Preparata and M. I. Shamos, *Computational Geometry—An Introduction,* Springer-Verlag, New York, 1985.
16. K. Sugihara, Topologically consistent algorithms related to convex polyhedra, in *Proceedings of the 3rd International Symposium on Algorithms and Computation, ISAAC'92, Nagoya* (*Lecture Notes in Computer Science 650*), *1992,* pp. 209–218.
17. K. Sugihara, Simpler proof of a realizability theorem on Delaunay triangulations, *Inform. Process. Lett.* **50,** 1994, 173–176.
18. K. Sugihara and M. Iri, A solid modelling system free from topological inconsistency, *J. Inform. Process.* **12,** 1989, 380–393.
19. K. Sugihara and M. Iri, Construction of the Voronoi diagram for "one million" generators in single-precision arithmetic, *Proc. IEEE* **80,** 1992, 1471–1484.
20. K. Sugihara and M. Iri, A robust topology-oriented incremental algorithm for Voronoi diagrams, *Int. J. Comput. Geom. Appl.* **4,** 1994, 179–228.