RA-004

Constant-space Data Structure for Farthest-point Voronoi Diagram

Tetsuo Asano Matsuo Konagaya *

Abstract

This paper presents a constant-space data structure for the farthest-point Voronoi diagram for a set of n points in the plane, which supports various operations using only a constant number of words of $O(\log n)$ bits and a read-only array to store the given point set. We show that the supported operations can be executed in $O(n)$ time.This is an extension of our previous results [1, 2, 3, 4].

1 Introduction

Recent progress in computer systems has provided programmers with unlimited amount of work storage for their programs. Nowadays there are plenty of space-inefficient programs which use too much storage and become too slow if sufficiently large memory is not available. We believe that there is high demand for space-efficient algorithms.

In this paper we assume that a point set is given on a read-only array. Thus, no permutation or rearrangement on an input array is allowed. Why do we insist on the read-only property? Given a point set, we may want to have several different data structures. If we reorder input points for a data structure, then we have to reorder them for another one. For example, a good ordering for closest-point Voronoi diagram may be different from one for farthest-point Voronoi diagram. In fact, a problem of finding the minimum-width annulus for a set of points in the plane can be solved using both of the Voronoi diagrams.

In this paper we introduce a new idea called a *constant-space data structure.* We just compute and maintain a constant number of words of $O(\log n)$ bits for a set of n points, and thus it takes work space of $O(1)$ words (of $O(\log n)$ bits). We prepare a collection of algorithms for supporting the imaginary data structure. All the operations on the target data structure are supported, but they may be slow. In this paper we propose a constant-space data structure for a farthest-point Voronoi diagram $FV(S)$ for a point set S in the plane. It is usually described using a doubly-connected edge list, which can be computed in $O(n \log n)$ time for n points. It supports the following operations

(1) to enumerate all Voronoi vertices,
(2) to enumerate all directed Voronoi edges,
(3) to determine whether a specified point is on the convex hull, and
(4) to follow the boundary of the Voronoi region for a point on the convex hull if we specify the point.

Once the doubly-connected edge list is constructed for a given set S of n points in $O(n \log n)$ time using $O(n)$ work space, we can enumerate all vertices in $O(1)$ time per vertex. In the constant-space data structure, with no preprocessing time we can enumerate all Voronoi vertices in $O(n)$ time per each vertex. It is just the same for Voronoi edges. Following the boundary of a Voronoi region is also done in $O(n)$ time per step.

2 Constant-space Data Structure

We propose a constant-space data structure for supporting a farthest-point Voronoi diagram $FV(S)$ for a set S of n points in the plane. For simplicity we assume that given points are in general positions, that is no four points of S are cocircular and thus every vertex of $FV(S)$ is incident to exactly three Voronoi edges. This restriction will be removed later. A diagram is defined by Voronoi regions and Voronoi edges. A Voronoi region $FVR(p_i)$ for a point $p_i \in S$ is the region such that the point p_i is farthest among the point set S from any point in the region. Each Voronoi region is known to be an infinite polygonal region, whose boundary consists of two infinite edges and (possibly no) finite edges with two endpoints. In this paper we orient Voronoi edges on the boundary of a Voronoi region $FVR(p_i)$ so that the Voronoi region for the point p_i lies to their left. Each Voronoi edge lies between two Voronoi regions. So, by $E(p_i, p_j)$ we denote a Voronoi edge between two Voronoi regions $FVR(p_i)$ and $FVR(p_j)$ with $FVR(p_i)$ to its left (and $FVR(p_j)$ to its right). Thus, the oppositely directed edge, called the twin

*School of Information Science, JAIST, Japan, {t-asano, matsu.cona}@jaist.ac.jp

edge, is denoted by $E(p_j, p_i)$. By our assumption exactly three Voronoi edges meet at each endpoint of Voronoi edges, which defines a Voronoi vertex. Thus, we can assume that each Voronoi vertex is characterized by three points from an input points such as $V(p_i, p_j, p_k)$.

It is well known that only those points of an input point set on its convex hull have their Voronoi regions [5, 7].

An example of a farthest-point Voronoi diagram is shown in Figure 1. In the figure, the leftmost point among the input points is denoted by p_1 and other input points on the convex hull are denoted by $p_2, \ldots, p_5$ in the counter-clockwise order. The Voronoi region $FVR(p_1)$ for the point p_1 is shadowed in the figure.

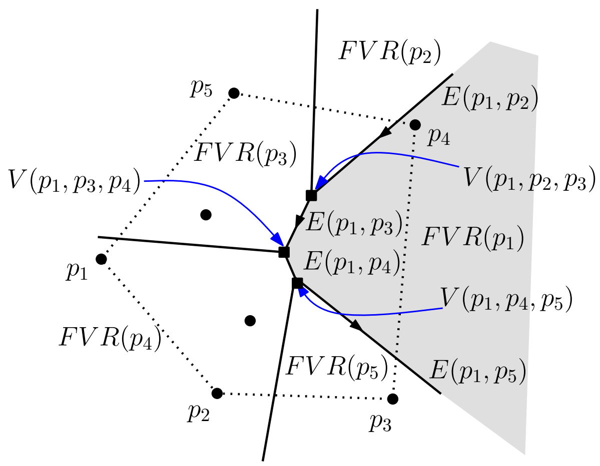

Figure 1: Farthest-point Voronoi diagram. Vertices on the convex hull are $\{p_1, \ldots, p_5\}$. $FVR(p_i)$ and $E(p_i, p_j)$ are the Voronoi region for point p_i and Voronoi edge for two points p_i and p_j, respectively.

A farthest-point Voronoi diagram is defined by Voronoi vertices, Voronoi edges which are either directed rays or directed line segments, and Voronoi regions which are infinite regions. It is common to use a doubly-connected edge list (DCEL in short) to represent a farthest-point Voronoi diagram. The DCEL consists of three collections of records [5].

Vertex record: A vertex record of a vertex v stores the coordinates of v and a pointer *IncidentEdge(v)* to a directed edge outgoing of v.

Face record: A face record of a face f stores a pointer *FirstVoronoiEdge(f)* to the first Voronoi edge on the boundary of the face f, which is a ray from the infinity.

Edge record: An edge record of a Voronoi edge e stores a pointer *NextVoronoiEdge(e)* to the next Voronoi edge on the same boundary and a pointer *IncidentFace(e)* to the face to its left.

We support these functions, *IncidentEdge(v)*, *FirstVoronoiEdge(f)*, *NextVoronoiEdge(e)*, and *IncidentFace(e)* by providing the following functions.

FirstExtremePoint(S**)** returns the leftmost extreme point (more exactly, the index of the point) in a set S of points.

CounterClockwiseNextExtremePoint(p_i**)** returns the index of the extreme point next to p_i in a counter-clockwise order on the convex hull.

FrontEndpointOfVoronoiEdge($E(p_i, p_j)$**)** returns the index k of the point p_k of S that determines the front (terminating) endpoint $V(p_i, p_j, p_k)$ of a directed Voronoi edge $E(p_i, p_j)$.

BackEndpointOfVoronoiEdge($E(p_i, p_j)$**)** returns the index k of the point p_k of S that determines the back (starting) endpoint $V(p_i, p_j, p_k)$ of a directed Voronoi edge $E(p_i, p_j)$.

NextVoronoiEdge($E(p_i, p_j), V(p_i, p_j, p_k)$**)** returns the next Voronoi edge $E(p_i, p_k)$ of $E(p_i, p_j)$ on the Voronoi region $FVR(p_i)$ which starts at the Voronoi vertex $V(p_i, p_j, p_k)$, more exactly the two indices i and k.

ExtremePoint(p_i**)** returns TRUE if and only if the point p_i is on the convex hull.

3 Algorithms for Supporting the Operations

Our constant-space data structure first computes the centroid c of the input point set in advance by computing the average x and y coordinates of all given points, and keeps it in the data structure. It is well known that the centroid always lies in the interior of the convex hull for the point set.

The operations listed above can be implemented in linear time using only $O(1)$ work space as follows:

FirstExtremePoint(S**):** The leftmost point in a point set S must be on the convex hull of S since the left half plane defined by the vertical line through the leftmost point is empty (i.e., no point of S is contained there). It is easy to find the leftmost point in S in $O(n)$ time using $O(1)$ work space.

CounterClockwiseNextExtremePoint(p_i)**:** Let $p_i \in S$ be an extreme point on the convex hull of S. We define a ray emanating from the point p_i in the opposite direction to the centroid c (refer to Figure 2). Then, we rotate the ray in the counter-clockwise order until it encounters a point of S, which is the point required. This is an intuitive description of an algorithm. Formally, we find the next extreme point p_j as follows. The point p_j must satisfy the following two properties:
(1) (c, p_i, p_j) is clockwisely oriented since p_j must be to the left of the directed line $\overrightarrow{cp_i}$, and (2) for any other point $p_k \in S \backslash \{p_i, p_j\}$ with the property (1) the three points p_k, p_i, p_j are ordered clockwisely since p_j lies to the left of $\overrightarrow{p_k p_i}$ to minimize the angle with the ray from p_i (see Figure 2). Thus, it can be computed in $O(n)$ time using $O(1)$ work space.

FrontEndpointOfVoronoiEdge$(E(p_i, p_j))$**:** Each Voronoi edge $E(p_i, p_j)$ is associated with one or two enclosing circles, whose centers are the endpoints of the edge. Due to our orientation, the front endpoint of a Voronoi edge $E(p_i, p_j)$ is determined by a point of S lying to the left of $\overrightarrow{p_i p_j}$. For each such point p_k (such that (p_i, p_j, p_k) is counter-clockwisely ordered) we compute the center of the circle through p_i, p_j, and p_k. This is a kind of mapping of a point of S into one on the perpendicular bisector of p_i and p_j. The center point giving the front endpoint must give an enclosing circle as stated above. Thus, the center point must be farthest from the center point of p_i and p_j. Thus, it can be computed in $O(n)$ time using $O(1)$ work space. See Figure 3. It shows how to find such a point. Given a Voronoi edge $E(p_1, p_3)$, extreme points of S lying to the left of the directed line $\overrightarrow{p_1 p_3}$ are p_4 and p_5. Since p_4 corresponds to a larger circle, the front endpoint of the edge is determined by p_4 together with p_1 and p_3 in this example. It should be noted that the last Voronoi edge on the boundary of a Voronoi region when we traverse it counterclockwisely has its front endpoint at infinity and thus its front endpoint is undefined.

BackEndpointOfVoronoiEdge$(E(p_i, p_j))$**:** This is just symmetric to the case of the front endpoint. Thus, the first Voronoi edge on the boundary of a Voronoi region has no back endpoint.

NextVoronoiEdge$(E(p_i, p_j), V(p_i, p_j, p_k))$**:** Once Voronoi edge $E(p_i, p_j)$ and its front endpoint $V(p_i, p_j, p_k)$ are known (more exactly, three indices i, j, and k are known), the next Voronoi edge is $E(p_i, p_k)$. Thus, it is done in $O(1)$ time.

ExtremePoint(p_i) We can easily compute the line L_i that is perpendicular to the ray from the centroid c toward p_i. If one of the half plane contains no point of S except p_i on the boundary, then the point p_i is on the convex hull by the definition of the convex hull. Otherwise, it is an interior point. See Figure 4. This is done in $O(n)$ time.

In addition, given a Voronoi edge $E(p_i, p_j)$ and its front endpoint $V(p_i, p_j, p_k)$, we know the three Voronoi edges $E(p_j, p_i), E(p_i, p_k)$ and $E(p_k, p_j)$ are outgoing edges from the Voronoi vertex $V(p_i, p_j, p_k)$ ordered in a clockwise way around the vertex. Thus, the data structure above behaves like a doubly-connected edge list.

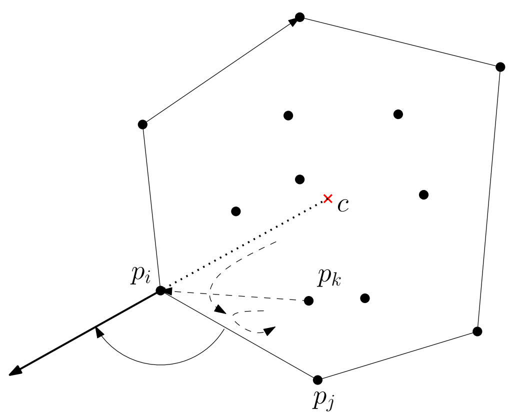

Figure 2: Finding the counter-clockwise next extreme point using a ray from p_i and the centroid c.

4 How to Cope with Degeneracies

We have assumed that given points are in general positions, that is, (1) no three points are on a line or (2) no four points are on a circle. In this section we will show how to cope with degeneracies on given points.

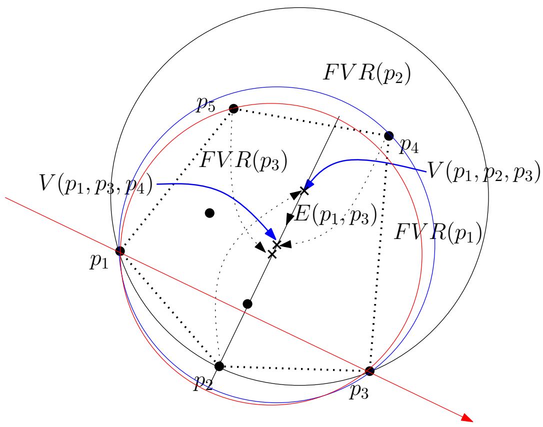

Figure 3: Finding the front endpoint of a Voronoi edge which is determined by a point of S lying to the left of the directed line $\overline{p_1p_3}$. Points lying to the directed line $\overline{p_1p_3}$ are p_4 and p_5. The center point of the circle defined by (p_1, p_3, p_4) is farther than that defined by (p_1, p_3, p_5), and thus the front endpoint of the directed Voronoi edge $E(p_1, p_3)$ is the Voronoi vertex $V(p_1, p_3, p_4)$. On the other hand, only one point p_2 lies to the directed line $\overline{p_3p_1}$, and thus that of $E(p_3, p_1)$ is $V(p_3, p_1, p_2) = V(p_1, p_2, p_3)$.

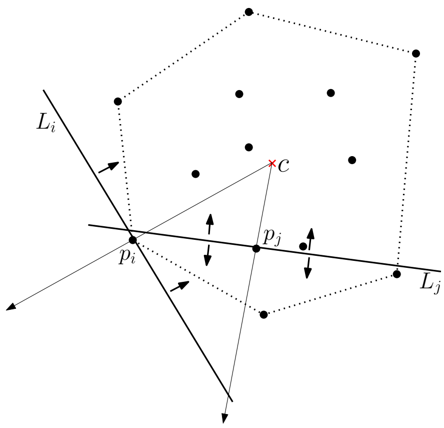

Figure 4: Deciding whether a given point is on the convex hull. The point p_i is on the convex hull shown by dotted lines since one of the half plane defined by the line L_i is empty. The point p_j is not so since none of the half planes is empty.

4.1 Degeneracy Caused by Collinear Points

Figure 5 shows an example of a degeneracy caused by collinear points in which four points lie on the convex hull of a given point set.

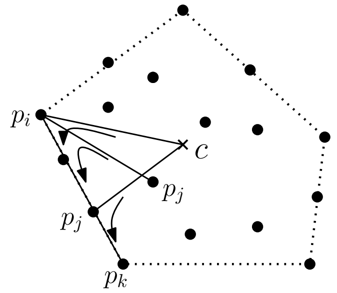

Figure 5: Degeneracy caused by collinear points.

Suppose three points from an input point set S lie on a line and one of the half plane defined by the line is empty, that is, it contains no point of S. If three points p_a, p_b, and p_c are arranged in this order on the line, the middle point p_b never contributes to the farthest-point Voronoi diagram for S, in other words, p_b has no its own Voronoi region, for any circle touching p_b never includes both of p_a and p_c, and the point p_a (resp., p_c) lying outside the circle is farther from the center of the circle than the other point p_c (resp., p_a). This means that we can neglect those intermediate points on the convex hull edges, which are not convex hull vertices. All these observations lead to the following algorithm for *CounterClockwiseNextExtremePoint(p_i)*:

CounterClockwiseNextExtremePoint(p_i)
  for each point $p_k \in S \backslash \{p_i\}$ **do**
      if (c, p_i, p_k) is counter-clockwise **then**
          break;
  for each point $p_j, j = k + 1, \ldots, n$ **do**
  if (c, p_i, p_j) is counter-clockwise **then**
      if (p_k, p_i, p_j) is counter-clockwise
          then $p_k = p_j$;
      else if (p_k, p_i, p_j) is collinear **and** (c, p_k, p_j)
      is counter-clockwise **then** $p_k = p_j$;
  return p_k.

4.2 Degeneracy Caused by Cocircular Points

Figure 6 shows another type of degeneracy, which is caused by cocircular points. In the figure five points $p_1, \ldots, p_5$ on the convex hull lie on a circle.

Suppose we are about to examine a convex hull edge (p_1, p_2). We first find a Voronoi edge $E(p_1, p_2)$,

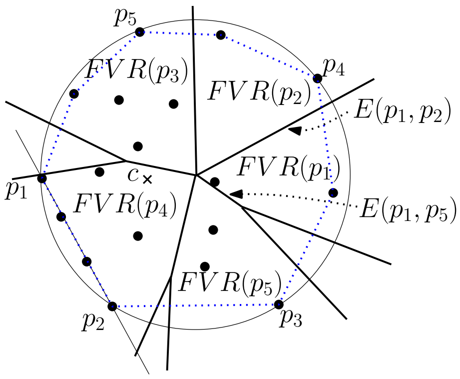

Figure 6: Degeneracy caused by cocircular points.

which is a ray from the infinity, as shown in the figure. To compute its front endpoint we examine all the points lying to the left of the directed line from p_1 to p_2 to find one whose corresponding circle center is farthest from the middle point of p_1 and p_2. In this case the three points p_3, p_4 and p_5 all give the same circle center since they are cocircular. Note that all those points must be extreme points. What we want is the point closest to p_2 in the clockwise order on the convex hull. Thus, if we find two candidate extreme points p_a and p_b to define the front endpoint of a Voronoi edge $E(p_i, p_j)$ and the four points p_a, p_b, p_i and p_j are cocircular, then we check the orientation of (p_i, p_a, p_b). We choose p_a if it is counter-clockwise, and choose p_b otherwise. This extra condition leads to a correct ordering of those cocircular points. In the example of Figure 6, the front endpoint of $E(p_1, p_2)$ is defined by p_3, and thus the next Voronoi edge should be $E(p_1, p_3)$. Its front endpoint is defined by p_4 and thus the following edge should be $E(p_1, p_4)$. In the same manner the Voronoi edge $E(p_1, p_4)$ is followed by $E(p_1, p_5)$. So, we have an edge sequence $E(p_1, p_2), E(p_1, p_3), E(p_1, p_4), E(p_1, p_5)$. Here note that the Voronoi edges $E(p_1, p_3)$ and $E(p_1, p_4)$ are degenerated edges, that is, their two endpoints coincide.

5 Applications of the Data Structure

Using the Constant-Space Data Structure for Farthest-Point Voronoi Diagram, we can of course draw the diagram for any given set of n points in $O(n^2)$ time using only $O(1)$ work space given as Algorithm 1 below.

A constant-work-space algorithm for drawing the farthest-point Voronoi diagram
Input: A set $S = \{p_1, \ldots, p_n\}$ of n points.
Output: Voronoi edges and Voronoi vertices of the farthest-point Voronoi diagram of the set S.
Algorithm{
  p_i = FirstExtremePoint(S).
  $i_0 = i$.
  repeat{
    p_j = CounterClockwiseNextExtremePoint(p_i).
    p_k = FrontEndpointOfVoronoiEdge(p_i, p_j).
    Report the first Voronoi edge $E(p_i, p_j)$ emanating from the Voronoi vertex $V(p_i, p_j, p_k)$.
    repeat{
      $p_j = p_k$.
      p_k = FrontEndpointOfVoronoiEdge(p_i, p_j).
      if(p_k is undefined) **then** exit the loop.
      Report the Voronoi edge (segment) $E(p_i, p_j)$ (pair of indices i and j in practice) and the Voronoi vertex $V(p_i, p_j, p_k)$ together with its coordinates and three indices.
    }(forever)
  } **until**($i = i_0$)
  Report the last Voronoi edge (ray) $E(p_i, p_j)$ emanating from the last Voronoi vertex.
  p_i = CounterClockwiseNextExtremePoint(p_i).
}

We can also compute the smallest enclosing circle of the points set by enumerating all the Voronoi vertices and Voronoi edges in $O(n^2)$ time. The smallest enclosing circle for a point set S is defined either by three points associated with a Voronoi vertex or by a diametral pair of extreme points. In the former case the point must appear as a Voronoi diagram of the farthest-point Voronoi diagram. In the latter case, the diametral pair of points must appear as one associated with a Voronoi edge. Thus, if we enumerate all Voronoi vertices and Voronoi edges, we can find the center of the smallest enclosing circle. Since there are $O(n)$ Voronoi vertices and edges, the algorithm runs in $O(n^2)$ time.

Another application is to the smallest annulus of a point set. Given a set S of n points in the plane, two co-centric circles are called an annulus of S if all the points of S lie between the two circles. See Figure 7. The width of an annulus is the difference of the two radii.

There are two cases to determine the center of the smallest-width annulus. In one case one of the circles is determined by three points and the other by a single point. In the other case both of them are determined by two points. The center in the latter case is given as an intersection of two Voronoi edges, one from the closest-point Voronoi diagram and the other from the farthest-point Voronoi diagram of S [6]. An algorithm for enumerating all the edges of the closest-point Voronoi diagram in $O(n^2)$ time using $O(1)$ work space is available [4]. Thus, a straightforward algorithm is to enumerate all edges of the farthest-point Voronoi diagram for each edge in the closest-point Voronoi diagram and to check intersection of those edges from different Voronoi diagrams. This algorithm runs in $O(n^4)$ time and $O(1)$ work space.

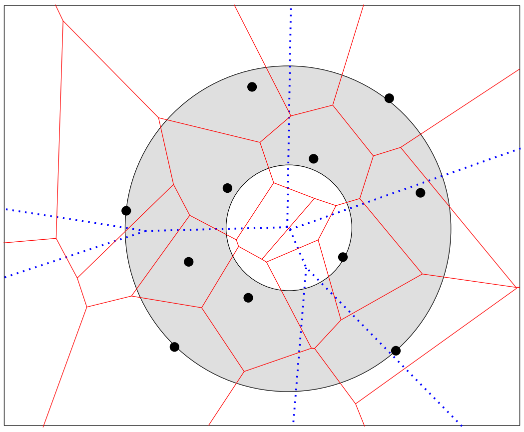

Figure 7: The minimum-width annulus for a set of points. The closest-point and farthest point Voronoi diagrams are drawn in solid and dotted lines, respectively, in the figure.

6 Concluding Remarks

We have presented a constant-space data structure for the farthest-point Voronoi diagram, which is a collection of algorithms to execute all of operations associated with the diagram as efficiently as possible using only constant work space. A number of problems are left open. One of them is to establish some trade-off between running time and amount of work space. Given work space of $O(s)$, how fast can we compute a farthest-point Voronoi diagram? It is not known whether we can establish time complexity such as $O(n^2/s)$ or $O(n^2/s \log n)$. To answer this question we need to devise a data structure using $O(s)$ space with $s = o(n)$. One typical question is how fast can we draw a farthest-point Voronoi diagram for a set of n point in the plane using $O(\sqrt{n})$ work space.

Acknowledgments

The work of T.A. was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas and Scientific Research (B). The authors would like to thank Yoshio Okamoto for his valuable comments.

References

[1] T. Asano. Constant-work-space algorithms: how fast can we solve problems without using any extra array? In *Proc. 19th Annu. Internat. Sympos. Algorithms Comput. (ISAAC), invited talk*, volume 5369 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag, 2008.

[2] T. Asano and M. Konagata. Zero-space Data Structure for Farthest-point Voronoi Diagram, Abstract of the 4th Annual Meeting of Asian Association for Algorithms and Computation, p.44, HsinChu, Taiwan, April, 2011.

[3] T. Asano, W. Mulzer, and Y. Wang. Constant-Work-Space Algorithm for a Shortest Path in a Simple Polygon. In *Proc. 4thWorkshop on Algorithms and Computation (WALCOM)*, pages 9–20, 2010.

[4] T. Asano and G. Rote. Constant-Working-Space Algorithms for Geometric Problems. In *Proc. 21st Canadian Conference on Computational Geometry. (CCCG)*, pages 87–90, Vancouver, 2009.

[5] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, third edition, 2008.

[6] H. Ebara, N. Fukuyama, H. Nakano, and Y. Nakanishi. Roundness algorithms using the Voronoi diagrams *Abstract of the First Canadian Conference on Computational Geometry.* p.41, 1989.

[7] F. P. Preparata and M. I. Shamos. *Computational Geometry. an Introduction.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1985.