Algorithms for Relative Neighbourhood Graphs and Voronoi Diagrams in Simple Polygons

(Extended Abstract)

V. Sarin & S. Kapoor[1]

1 Introduction

In this paper we present algorithms for two related problems. The first problem is that of finding the Voronoi diagram of the vertices of a simple polygon whereas the second problem is that of finding the relative neighbourhood graph of a simple polygon. These problems differ from the commonly known problems of finding these diagrams for a set of points in the plane in that there is an additional requirement of visibility in the definition. Finding the relative neighbourhood graph of simple polygons has been discussed by Toussaint [7]. This problem has applications in image processing and is also of theoretical interest because of its relationship to the Voronoi Diagram.

Previous work on these diagrams includes $O(nlogn)$ schemes for finding the constrained Delaunay triangulation amongst a set of line segments [1, 8]. In this paper a new algorithm has been presented for constructing the Voronoi diagram of a simple polygon. An extension of this algorithm to the situation when polygonal obstacles are present is also outlined. The algorithm assumes an arbitrary initial triangulation of the simple polygon (efficient methods for triangulation of simple polygons exist). The Voronoi diagrams are constructed using this triangulation. A divide and conquer approach is used in a fashion similar to the Voronoi diagram construction in [5]. The algorithm proposed in this paper provides a simple solution to a more general problem - that of finding the Voronoi diagram for a set of line segments. It may be noted that the dual of such a Voronoi diagram is the constrained Delaunay triangulation discussed in [1, 8].

The motivation for the Voronoi Diagram algorithm is in the construction of the Relative Neighbourhood graph for simple polygons. For the Relative Neighbourhood graph the previous best scheme is of $O(n^2)$ time complexity [2] (To the authors' knowledge). We present an efficient algorithm with $O(nlogn)$ time complexity for the relative neighbourhood graph problem (also referred to as the RND problem). The algorithm starts with a Delaunay Triangulation of the simple polygon (the dual of the Voronoi Diagram), and eliminates edges to obtain the RND. It uses a number of passes over the simple polygon and eliminates triangulation edges using the vertices of the polygon. This approach is also amenable to an extension of the problem, i.e. construction of the RND for simple polygons with obstacles.

The paper is organized as follows: Section 2 contains definitions, Section 3 deals with the Voronoi diagram and Section 4 with the RND. Section 5 makes concluding remarks.

2 Definitions

1. *Voronoi Diagram of a simple polygon(SVD)* : The Voronoi diagram of a simple polygon is the Voronoi diagram of the vertices of the polygon with the restriction that all points in the region associated with point p_i are visible to p_i. We let $Vor(P)$ denote the Voronoi diagram of polygon P and $Vor(p_i)$ denote the Voronoi region associated with the point p_i.
2. *Constrained Delaunay Triangulation (CDT)* : The straight line triangulation of the Voronoi diagram of a simple polygon.
3. *Lune* : $lune(a, b)$ ~~is the set of points x such that $d(a, x)$~~ $< d(a, b)$ and $d(a, x) < d(a, b)$ where $d(p, q)$ is the euclidean distance between points p and q.
4. *Relative Neighbourhood Graph (RNG)* : RNG of a set of points in a plane is a set of edges p_ip_j such that no point p_k lies inside $lune(p_i, p_j)$.
5. *Simple Polygon Relative Neighbour Decomposition (SRND)* : SRND of a simple polygon with vertices $p_0 \ldots\ldots p_{n-1}$ and edges $p_ip_{(i+1)mod n}$ is defined as a set of edges p_ip_j s.t. : (i) p_i & p_j are visible to each other, and (ii) No vertex p_k which is visible to either p_i or p_j lies inside $lune(p_i, p_j)$.

[1]Department of Computer Science & Engg., Indian Institute of Technology, Hauz Khas, New Delhi 110016, India.

6. *Simple Polygon with polygonal obstacles* : A simple polygon P which contains several other non overlapping simple polygons, say $P_1, P_2.....P_k$, within itself is called a simple polygon with polygonal obstacles. In this case the interior of the simple polygon is $P - \bigcup_{i=1}^{k} P_i$.

3 Voronoi Diagrams of a simple polygon

The problem of finding the Voronoi diagram of simple polygon P is an extension of the problem of finding the Voronoi diagram for a set of points in the plane. We now present an outline of an algorithm based on the divide and conquer approach to compute the Voronoi diagram of a simple polygon. This is followed by a detailed description of each step in the algorithm.

Procedure Voronoi_diagram_1

1. Partion P into two subpolygons of approximately equal sizes.
2. Construct $Vor(P_1)$ and $Vor(P_2)$ recursively.
3. Merge $Vor(P_1)$ and $Vor(P_2)$.

Step 1. Consider any arbitrary triangulation of P and call it P_t. Next consider the dual graph of P_t, say G. Note that G is a tree and the maximum degree of any vertex of G is three. G can be used to divide P into two parts containing approximately equal number of triangles. Suppose edge e divides G into two parts G_1 and G_2 . The triangulation edge t corresponding to e is used to partition P into P_1 and P_2 (corresponding to G_1 and G_2 resp.). After partition, t is included in both P_1 and P_2.
Step 2. This step is to recursively compute $Vor(P_1)$ and $Vor(P_2)$. To compute $Vor(P_1)$, we augment the subpolygon P_1 with two new vertices x_1 and x_2 as shown in Fig 1. This ensures that the part of $Vor(P_1)$ extending across the edge t is also computed at this step. This would later be merged with $Vor(P_2)$ to get $Vor(P)$.

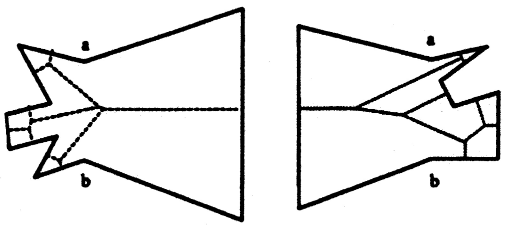

Figure 1: Voronoi diagrams of P_1 and P_2

Similar construction is done for P_2, and Voronoi diagrams for these augmented polygons are computed recursively. It may be noted that henceforth, the terms - subpolygon P_1 and augmented subpolygon P_1 would be used interchangeably.
Step 3. The last and final step is the merger of $Vor(P_1)$ and $Vor(P_2)$ into a single $Vor(P)$. To perform this merger we will first define a geometrical construct.

Definition 1. *Given a partition $\{P_1,P_2\}$ of P, let σ (P_1,P_2) denote the set of Voronoi edges that are shared by pairs of polygons $V(p_i)$ and $V(q_j)$ of $Vor(P)$, for $p_i \in P_1$ and $q_j \in P_2$.*

The following property is true when we are dealing with a set of points in the plane:

Property 1. *([5], pp. 207) If P_1 and P_2 are linearly separated then σ (P_1,P_2) consists of a single monotone chain w.r.t the line separating P_1 and P_2 .*

This property needs to be modified if it is to be useful in the case of a partition of simple polygons. The following modification incorporates visibility into the property.

Definition 2. *A line t linearly v-separates a polygon P into parts P_1 and P_2 if every path from a point in P_1 to a point in P_2 crosses t.*

If such a line exists then P_1 and P_2 are *linearly v-separable.*

Lemma 3.1 *If P_1 and P_2 are linearly v-separable by t then σ (P_1,P_2) consists of a single monotone chain w.r.t t.*

Proof. Similar to that in [5].

As P_1 and P_2 are linearly v-separable by the edge t, σ is a single monotone chain w.r.t. t. If we assume that σ (P_1, P_2) cuts P into a left portion π_L and a right portion π_R then it can be proved that

Lemma 3.2 *If P_1 is the left half and P_2 is the right half of P and σ (P_1, P_2) cuts the plane into a left portion π_L and a right portion π_R then $Vor(P) = (Vor(P_1) \cap \pi_L) \cup (Vor(P_2) \cap \pi_R)$.*
Proof. Follows along lines similar to [5].

Note that σ (P_1, P_2) is contained entirely within P.

So, **Step 3** consists of constructing the polygonal chain σ, separating P_1 and P_2, and discarding all edges of $Vor(P_2)$ to the left of σ and all edges of $Vor(P_1)$ to the right of σ. The result obtained is $Vor(P)$.

The most crucial step in the above algorithm is the construction of the dividing chain. We now present the algorithm to compute σ efficiently. The spirit is similar to the chain finding method for Voronoi diagrams of a set of points in the plane. The reader is referred to Fig. 2.

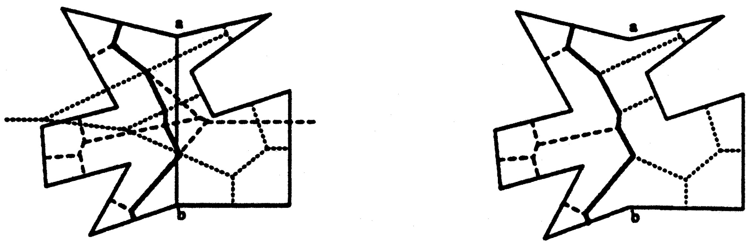

Figure 2: (a) Construction of σ (b) Voronoi diagram of Simple Polygon P

Procedure compute_chain

1. Suppose that P has been divided across the edge t. Let x_0 be the point of intersection of P and $V_1(a)$. Further, let x_0 be on the boundary of Voronoi region of p_i in P_1. Let $q_j = a$.
2. Let $l_{i,j}$ be the perpendicular bisector of $p_i q_j$.
3. Let x_r be the first point of intersection of $l_{i,j}$ and some edge e_i in $Vor(P_1) \cup Vor(P_2)$ in a direction monotone w.r.t. t.
    - If $e_r \in P$ then $l_{i,j} = e_r$. Go to 3.
    - If $e_r \in V_1(b) \cup V_2(b)$ then exit.
    - If $e_r \in Vor(P_1)$ and e_i is the bisector of line joining p_i and p_k, then $p_i = p_k$. Go to 2.
    - If $e_r \in Vor(P_2)$ and e_i is the bisector of line joining q_j and q_k, then $q_j = q_k$. Go to 2.

The key step in forming the dividing chain is to determine the edge intersected first by the walk at each stage. For this we have a property of the walk similar to that in [5].

Lemma 3.3 *If $V_1(p_i) \in Vor(P_1)$ then as long as σ is within $V_1(p_i)$ (it can change direction many times within $V_1(p_i)$ if it intersects a number of edges of $Vor(P_2)$) it always keeps turning in a monotone direction (either clockwise or anti-clockwise).*
Proof. Similar to that in [5].

Since P_1 and P_2 are planar graphs, σ can be computed in $O(n)$ time.

Lemma 3.4 *The time complexity of Voronoi_diagram_2 is $O(nlogn)$.*
Proof. Step 1 of Voronoi_diagram_2 can be performed in $O(n)$ by using DFS. Time for constructing the polygonal chain is $O(n)$. Therefore Step 2 can be performed recursively in time $T(cn) + T((1-c)n)$ where $1/2 < c < 2/3$. Step 4 can be performed simultaneously while computing σ. Therefore, $T(n) = O(nlogn)$.

Theorem 3.1 *SVD(P), the Voronoi diagram of a simple polygon can be computed in $O(nlogn)$ time.*
Proof. Correctness of the algorithm follows from Lemmas 3.1 and 3.2 and the fact that the chain construction is done correctly by procedure compute_chain. The time complexity has been proved using in Lemma 3.4.

3.1 Voronoi diagram of a simple polygon with polygonal obstacles

Procedure Voronoi_diagram_2 (modified for simple polygon with obstacles)

1. Partition P into approximately equal subpolygons P_1 and P_2. This may be done by a straight line which divides the vertices of P into two halves. The partial polygon on the left of this line would be P_1 and the one on the right would be P_2. As P_1 and P_2 may have more than one common edge, we take all such edges $t_i's$ which arise out of the partition instead of just one edge as in Voronoi_diagram_1.
2. Construct $Vor(P_1)$ and $Vor(P_2)$ recursively.
3. Construct the polygonal chain σ, separating P_1 and P_2 . In this step we need to merge $Vor(P_1)$ and $Vor(P_2)$ along all the edges $t_i's$ one by one. Lemma 3.5 presented later allows easy construction of σ.
4. Discard all edges of $Vor(P_2)$ to the left of σ and all edges of $Vor(P_1)$ to the right of σ .

Lemma 3.5 *The polygonal dividing chain σ consists of edge-disjoint chains.*

Proof. Let l be the straight line dividing P into P_1 and P_2. Suppose that σ consists of chains $\sigma_i's$, where each σ_i corresponds to an edge t_i. Since each σ_i is monotonic w.r.t. l, it does not have a common edge with σ_j $(i \neq j)$.

We can use this lemma to actually construct the merger in linear time. We note that now the dividing line may not be a single monotone chain but a number of non-interacting chains which are monotone w.r.t. the same line l. The construction of each such chain - σ_i can proceed independent of any other chain, and follows exactly the same method as described earlier (in procedure Voronoi_diagram_1). Since σ consists of $O(n)$ distinct edges, the time complexity of the construction of σ is $O(n)$.

The time complexity of the above procedure is given by the following lemma.

Lemma 3.6 *The time complexity of procedure Voronoi_diagram_2 is $O(nlogn)$.*

Proof. As discussed above, Step 3 takes $O(n)$ time, and according to Step 1, Step 3 is performed $O(logn)$ number of times. Therefore, procedure Voronoi_diagram_2 has a time complexity of $O(nlogn)$.

Theorem 3.2 *Voronoi diagram of a simple polygon with polygonal obstacles can be computed in $O(nlogn)$ time.*

Proof. Correctness follows from Lemma 3.5 and the correctness of the proceDure for computing the dividing chain σ. The time complexity follows from Lemma 3.6.

In conclusion we note that the problem of finding the Voronoi diagram of a set of line segments can be viewed as a special case of the problem of finding the Voronoi diagram of a simple polygon with polygonal obstacles. In this case we can consider the polygon to be an *infinite* rectangle encompassing all the line segments. Each of the line segments can be treated as a polygonal obstacle. The Voronoi diagram obtained can be suitably modified to remove the effects of the four vertices of the infinite rectangle. This results in the Voronoi diagram of the set of line segments (which were viewed as obstacles). A dual of this Voronoi diagram is the constrained Delaunay triangulation (also discussed in [1, 8]).

4 Relative Neighbour Decomposition of a simple polygon

The problem of finding the relative neighbour decomposition of a simple polygon has been defined in Sec. 2.

An $O(n^2)$ algorithm for finding the RND of a simple polygon is presented in [2]. This algorithm is based on finding the visibility graph (VG) of the simple polygon. Since this may require finding $O(n^2)$ edges, the algorithm offers no scope for further improvement.

Instead of the VG, we use the fact that RNG is a subset of the Delaunay Triangulation (DT) to construct the RND of the simple polygon. This technique was used by Supowit [6] to construct the RND of a planar point set. The CDT is the dual of the Voronoi diagrams constructed in Sec. 3. This can be proved by techniques similar to that used in [8].

4.1 Algorithm for RND of a simple polygon.

A new algorithm for finding the RND of a simple polygon is presented here.

Algorithm-RND

1. Find CDT of the simple polygon P.
2. Remove the edges from DT which do not belong to RND.

The algorithm assumes that for a simple polygon, RND is a subset of CDT (Theorem 2 presented in [2] can be applied in a similar way to prove this assumption). An $O(nlogn)$ algorithm for finding the CDT of a simple polygon is presented in Sec. 3. Therefore , we will only describe step 2 of the algorithm-RND. This step will run in $O(nlogn)$ time yielding a total time complexity of $O(nlogn)$. We first establish the following lemma.

Lemma 4.1 *If a vertex v eliminates an edge ab from being in RND, i.e. if $v \in lune(a,b)$ then $\angle avb = 60^o$.*
Proof. From the definition of $lune(a,b)$.

In order to delete those edges from CDT which are not there in the RND, we can look at each point $v \in P$ and delete each edge $e \in CDT$ whose lune contains v. This is simplified by Lemma 4.1. According to this, we need to send just six rays from v and check if $v \in lune(e)$ for all the edges e which are intersected by any of these six rays.

We perform the above exercise in the following way. We will scan the polygon along six directions separated by 60^o one after the other. Here we discuss one such scan (all the others are similar). Without loss of generality we can assume that the ray moves from positive infinity towards zero along the x-axis. So we will consider such a ray for each point and determine the edges of CDT eliminated by the point.

The algorithm consists of the following steps.

Step 2.1. We use G (dual of P) to sort the edges in a topological order along the scan direction. We obtain a directed tree after this step with some source nodes (indegree = 0) and some destination nodes (outdegree = 0). Another property of this tree is that if we consider edge e, then its dual edge - $t \in CDT$ can be eliminated only by points which occur in the triangles which are the ancestors of e. In other words, if there is no path from a node v in the tree to an edge e, then the projection of points of v (v is a triangle in P) on the y-axis falls outside the projection of t (dual of e) on the y-axis.

Step 2.2. As mentioned earlier, the property of the tree is that points which may eliminate an edge t are found only along the paths leading from the sources to the edge e (dual of t). So we will devise a scheme to determine if the edge t is to be eliminated or not.

Assume, for simplicity, that the tree above is just one path from a single source vertex s to e. Start from the source triangle and form a list of vertices (initially, this list contains only one vertex of the source triangle). These vertices will be useful in eliminating edges from CDT. As we keep moving along the path from s to e, we cross CDT edges one by one. At each edge ab we do the following :

Assume that the list has a top and a bottom end (the vertices in the list are ordered along the list from top to bottom as they occur clockwise in P). Let a occur on the top and b at the bottom side of the edge ab. Now for inserting a into the top end of the list we keep popping vertices till we get a vertex v whose projection on the y-axis is less than that of a. In other words, a will eclipse the vertices popped. Now push a onto the list. Insert b at the bottom end of the list in a similar way. Clearly, the vertices popped will never be able to eliminate edge e because their rays have been obstructed by the polygon P.

When we reach e, we have a list of vertices whose rays should intersect t (after we have popped off vertices and before pushing the end points of t onto the list).

Now we determine if a vertex v in the list eliminates t, i.e if $v \in lune(t)$. This can be done sequentially from any one of the ends of the list. If $v \notin lune(t)$, then pop v (It is easy to show that v will not be able to eliminate any edge after e along this path) If on the other hand $v \in lune(t)$, then eliminate t. Also in this case do not pop v but maintain the current list.

Lemma 4.2 *The above procedure correctly eliminates an edge which belongs to CDT - RND.*

Proof. When we reach an edge e, we have a list of vertices which is a superset of the vertices which may eliminate e (along that scan direction). Therefore, if e is to be eliminated, it is eliminated by some vertex in the list. If there is a vertex in the list, say $v \in lune(e)$, then it is not necessary to check if v is visible from e because $v \in lune(t)$, the ray from v strikes t and all vertices eliminated before v (say from the top of the

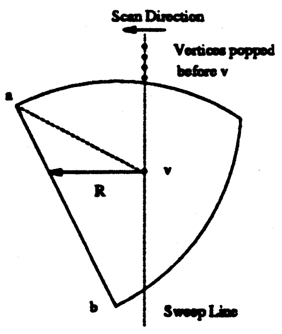

Figure 3: Elimination of an edge

list) do not lie in the lune. If there had been an obstruction of visibility between the end points of the edge e and the point v, then there would have been a vertex in the lune considered before v (Ref. Fig 3).

To establish the efficiency of the scheme we note that the vertices in the list are pushed and popped once each only, and the top and bottom elements of the list are used to eliminate CDT edges.

Actually, the tree is made up of a number of sources and destinations and paths among them. Therefore, instead of a single sweep line (list of nodes) travelling from the source to the destination, we now have one sweepline starting from each source vertex and moving down a path. Complications arise in two cases :

- When a path gets split into two (remember that G is a tree with maxdegree equal to 3, so a path can split into at most two paths).
- Two paths merge into a single path (reverse of splitting).

In the case of merger of two paths, the two lists associated with each of the merging paths are concatenated into a single list to be moved down along the single merged path. In case of splitting, the approach is different. One single list has to be split at a certain point and each of the two resulting lists is taken along different paths. The point of splitting of the list is determined by the point of P at which the path splits (See Fig. 4).

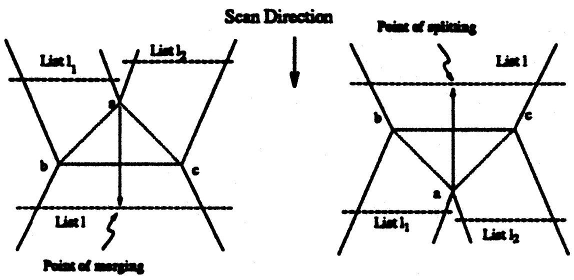

Figure 4: (a) Merging of two lists and (b) Splitting of a list

Now the demands on the list can be summarized as : (i) insertions and deletions from both ends only, (ii) concatenating two lists and (iii) splitting a list into two at a specific point.

The following lemma can be used to perform these operations efficiently.

Lemma 4.3 *Concatenation of two lists and splitting of a list at a specific point can be done in O(logn) steps. Insertions and deletions at the ends of the list can be done in O(1) steps.*

Proof. Note that the elements in the list are ordered according to the values of their projections on the y-axis. Using the data structure of concatenable queue we can ensure, in $O(logn)$ time the following operations on a dynamic ordered sequence - Concatenation of two lists and Splitting of a list at a specified point. Maintaining pointers to the minimum and maximum values in the queue allows insertions and deletions at the ends of the lists in $O(1)$ steps.

The time complexity of the preceding algorithm is given by the following lemma.

Lemma 4.4 *The time complexity of the algorithm for computing RND is O(nlogn).*

Proof. As each point is inserted and deleted exactly once, $O(n)$ work is done on all insertions and deletions. Splitting and merging can be charged to the points causing them. From Lemma 4.3 it follows that the net time complexity of the algorithm is $O(nlogn)$.

Theorem 4.1 *The RND of a simple polygon can be obtained in O(nlogn) operations.*
Proof. If an edge is to be eliminated, it gets eliminated by a vertex in some list (in one of the scans along the six directions). Moreover, no edge gets eliminated incorrectly (lemma 4.2). Hence the algorithm correctly computes the RND of a simple polygon. Lemma 4.4 establishes that RND of a simple polygon can be computed in $O(nlogn)$ operations.

4.2 RND of a simple polygon with polygonal obstacles

The preceding scheme is also applicable to finding the RND of a simple polygon with obstacles. In this case the scan direction gives us a DAG instead of a tree. The major difference now is that the after a path has split it may merge again forming a cycle. However note that the DAG still has nodes of degree atmost 3 and therefore the merging process may be carried out as before. The time complexity of the scheme in this case is also $O(nlogn)$.

5 Conclusions

The algorithm for computing the Voronoi diagram of a simple polygon leads to an O(nlogn) scheme for this problem. This time complexity may not be optimal but the technique used was conceptually simple and intuitively appealing. The existing algorithms are for the more general case of n points in a plane with n line segments, which introduced extra complexity. The algorithm presented uses the information about a (triangulated) simple polygon to simplify the task of finding the Voronoi diagram. Its extension to the case of a simple polygon with obstacles retains the simplicity of this method.

The algorithm for computing Relative neighbour decomposition presented in Section 4 has the time complexity of $O(nlogn)$ which is an improvement over the previous best algorithm of $O(n^2)$ time complexity.

An interesting problem would be the extension of these algorithms to dynamic simple polygons where we may consider vertices being added to the polygon dynamically. This, however is an open problem.

References

[1] "Constrained Delaunay Triangulation" by L.P.Chew, Proc. of third ACM Symposium on Computational Geometry, June 1987.

[2] "Computing Relative neighbourhood decomposition of a simple polygon" by H.A. ElGindy and G.T.Toussaint, pp 53-76, Computational Morphology , Elsevier Science Publishers.

[3] "Linear time algorithms for visibility and shortest path problems inside simple polygons" by L.Guibas, J.Hershberger, D.Leven, M.Sharir and R.Tarjan, Proc. of third ACM Symposium on Computational Geometry, June 1986.

[4] "Efficient Algorithms for Euclidean shortest paths and visibility problems amongst polygonal obstacles" Kapoor Sanjiv and S.N. Maheshwari, Proc. ACM Symposium on Computational Geometry, 1988.

[5] Computational Geometry : An Introduction by F.P.Preparata and M.I.Shamos, Springer Verlag.

[6] "Relative neighbourhood graph" by K.J.Supowit, JACM Vol.30, No.3, 1983.

[7] "The Relative neighbourhood graph of a finite planar set" by G.T.Toussaint, Image Recognition, Vol. 12, pp. 261-268.

[8] "An optimal algorithm for constructing the Delaunay triangulation of a set of line segments" by C.Wang and L.Schubert, Proc. of third ACM Symposium on Computational Geometry , June 1987.