8-1-2014

Approaches for Generating 2D Shapes

Pratik Shankar Hada
University of Nevada, Las Vegas, pratik.hada@gmail.com

APPROACHES FOR GENERATING 2D SHAPES

by

Pratik Shankar Hada

Bachelor of Computer Engineering
Tribhuvan University
Institute of Engineering, Pulchowk Campus
2007

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science Degree in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2014

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Pratik Shankar Hada

entitled

Approaches for Generating 2D Shapes

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Department of Computer Science

Laxmi P. Gewali, Ph.D.,, Committee Chair

Ajoy Datta, Ph.D.,, Committee Member

John Minor, Ph.D.,, Committee Member

Rama Venkat, Ph.D.,, Graduate College Representative

Kate Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

August 2014

Abstract

Constructing a two dimensional shape from given a set of point sites is a well known problem in computation geometry. We present a critical review of the existing algorithms for constructing polygonal shapes. We present a new approach called *inward denting* for constructing simple polygons. We then extend the proposed approach for modeling polygons with holes. This is the first known algorithm for modeling holes in the interior of 2d shapes. We also present experimental investigations of the quality of the solutions generated by the proposed algorithms. For this we implemented the proposed algorithms in Java programming language. The prototype program can be executed by users to enter point sites interactively. The experimental results show that the perimeter of the generated shape is at most 33% more than the length of the minimum spanning tree.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, *Dr. Laxmi Gewali* for guiding me on each and every aspect of the thesis. I would also like to thank *Dr. Ajoy Datta* for helping me with the difficulties and the confusion in official matters. I would also like to thank *Dr. John Minor* and *Dr. Rama Venkat* for being part of the committee.

My sincere gratitude goes to my parents, my brother *Sushant Shankar Hada* and my sister *Moonmoon Hada* as they have always inspired me to work hard and supported me.

At last, I would like to thank all my friends, seniors and juniors for their love and support.

Pratik Shankar Hada

University of Nevada, Las Vegas
August 2014

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Connecting dots to make a shape is a fascinating problem well known from the dawn of civilization to the present time. Star clusters in the night sky were intuitively connected to give shapes such as ursa major and ursa minor by ancient astronomers [4]. After the advent of computational geometry in early 1970, a formal way of connecting dots to make shapes was pursued [6]. One of the main motivations for investigating this problem is its wide range of applications in image processing, edge detection and object recognition [3]. Constructing shapes by connecting dots has also been applied in optical character recognition and face recognition [3].

In computational geometry, one of the widely used models of shape is a simple polygon [5]. A simple polygon is a closed chain consisting of line segments such that the plane is partitioned by the polygon into three connected parts: bounded interior, unbounded exterior, and the polygon itself. For experimental investigation of the properties of polygonal shapes, it is very important to construct random polygons for a given set of point sites. However, the problem of constructing random polygons from a given point sites (vertices) is still open [1]. For restricted classes of polygons, a few algorithms for randomly generating such shapes have been reported [1,8]. For example, polynomial time algorithms for random generation of monotone polygons are described in [8]. For generating simple polygons from a given set of point sites only heuristics have been considered [1]. In a heuristic approach, the boundary is constructed by following certain rules. If the resulting shape is a simple polygon then it is accepted as a solution. If the boundary edges are intersecting in their

interior then such edges are replaced by non-intersecting ones. This approach can generate polygonal shapes for practical purpose but can not be used for random generation.

In this thesis, we present a new approach for generating a simple polygon for given set of point sites (i.e. vertices). The technique, which we call *inward denting*, starts from the convex-hull boundary as a partial solution S_i and connects interior nodes iteratively by deforming a selected edge from S_i. We also consider the performance of the proposed algorithm by actual implementation.

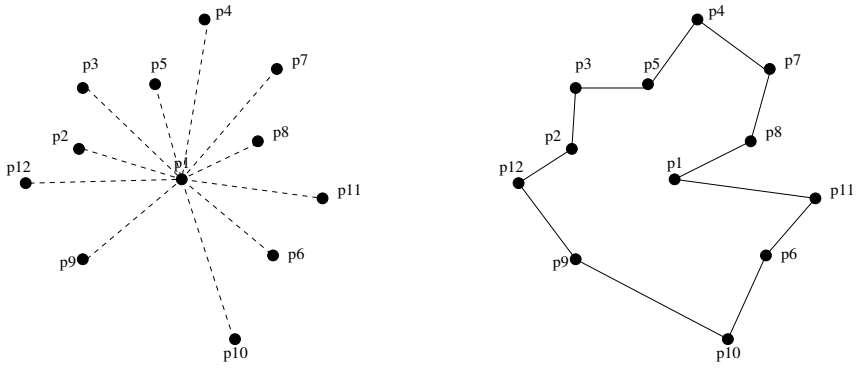
The thesis is organized as follows. In Chapter Two, we present a brief review of existing algorithms for generating polygonal shapes. In Chapter Three, we present a detailed development and description of the proposed inward denting algorithm. We show how Voronoi diagrams can be used to make the algorithm more efficient. In addition, we show that Voronoi diagrams can be further used to model holes inside the polygon. In Chapter Four, we present an implementation of the inward denting algorithm in the java programming language. To measure the performance of the generated solution, we compare the length of the perimeter of the generated shape with the length of the minimum spanning tree induced by the input vertices of the polygon. Finally, in Chapter Five, we discuss the implication of the experimental results and possible improvements and generalizations of the proposed inward denting approach.

Chapter 2

Review of Polygon Generation Algorithms

2.1 Preliminaries

In this chapter we present a critical review of algorithms reported in the literature, for connecting a given set of points (nodes) to make a simple polygon. One of the simple ways to connect a set of nodes S in two dimensions (2D) is to first obtain an angularly sorted list L of nodes about some point in the interior of the convex hull. The points are connected in the order they appear in L. This is illustrated in Figure 2.1.



(a) Given Set of Nodes S. (b) Generated Polygon.

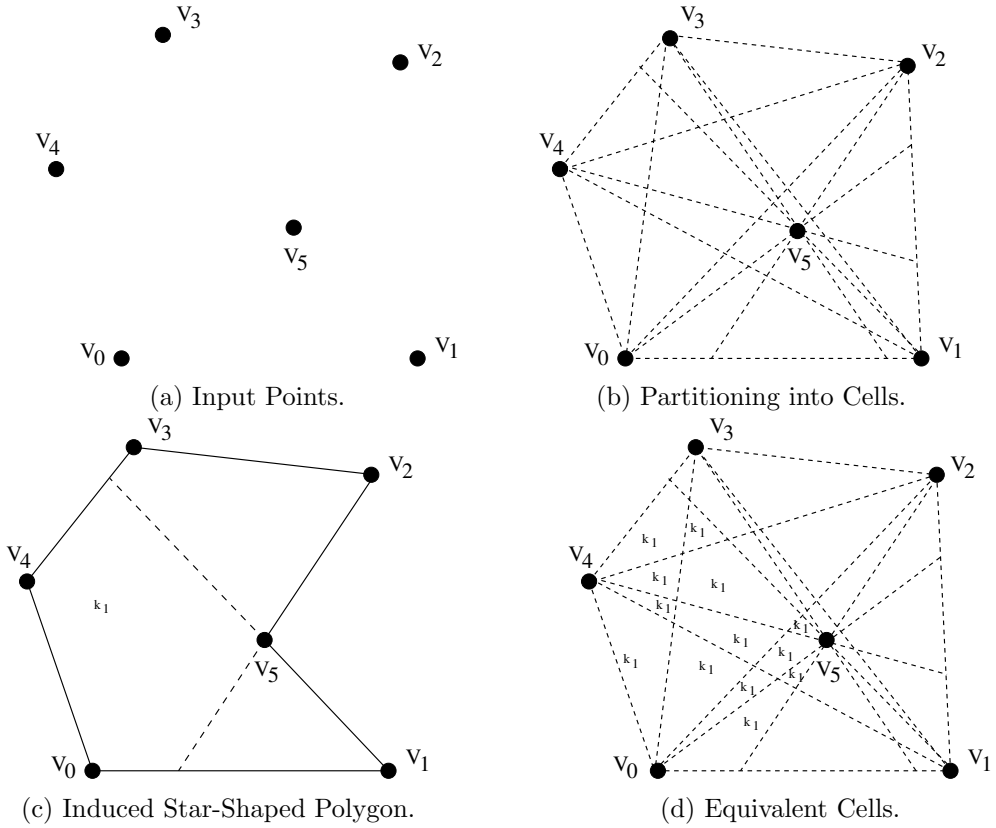
Figure 2.1: Illustrating Polygonization by Sorting.

In Figure 2.1a, twelve point sites are drawn as small dots. The set S of these points are

denoted as $S = \{p_1, p_2,...,p_{12}\}$. We pick a point say p_1 in the interior of convex hull CH of S. The points are angularly sorted about p_1. The sorted list is $L=\{p_3, p_2,p_12,...,p_5\}$. When consecutive point sites in L are joined by edges we get a simple polygon as shown in Figure 2.1b. We call this method angular-sorting method. The shape of the resulting polygon clearly depends on the choice of the point about which the sorting is performed. The sorting method is intuitive and easy to implement. The time complexity of the algorithm is clearly $O(nlogn)$, where n is the number of vertices in the polygon.

2.2 Generating Star-Shaped Polygons

Some researchers have considered the problem of generating polygons satisfying certain properties. Auer and Held [1] have proposed an algorithm for constructing a star-shaped polygon from a given set of point vertices $V= v_0, v_1, ..., v_{n-1}$ in the plane. This algorithm's time complexity is $O(n^2)$. It is noted that a polygon is called star-shaped if there is an internal point from which all other points inside the polygon are visible. The algorithm starts with the convex hull CH of the given set V and partitions the CH region into several cells. The cells are produced by considering partitioning line segments formed by extending pairs of vertices. The end points of partitioning segments are on the boundary of the convex hull CH. The arrangement of partitioning segments produce cells. There could be potentially $O(n^4)$ cells. Figure 2.2b shows the partitioning cells for the point distribution shown in Figure 2.2a.



(a) Input Points. (b) Partitioning into Cells.

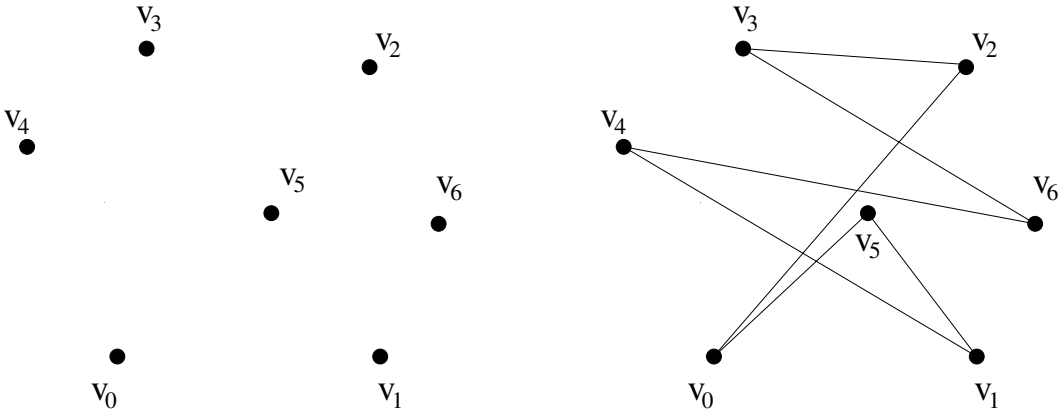
(c) Induced Star-Shaped Polygon. (d) Equivalent Cells.

Figure 2.2: Illustrating Partitioning into Cells.

The star polygon induced by a point q inside a given cell is obtained by connecting input vertices angularly sorted about q. Figure 2.2c shows a star-shaped polygon corresponding to cell k_1. It is easily observed that star-polygons induced by all points inside a particular cell are identical. It is also noted that in some cases more than one cell could have identical star-shaped polygons. All cells equivalent to k_1 are labeled as k_1 in Figure 2.2d.

2.3 Heuristics Approaches

A technique of generating a polygon from given nodes is to try a permutation of the input point set. This approach was suggested by Auer and Held [1] in which a permutation of n given vertices is picked randomly and the vertices are connected implied by the picked permutation order. If the implied permutation results in a simple polygon then it is accepted. Otherwise, the permutation is rejected. We can illustrate with an example as follows.



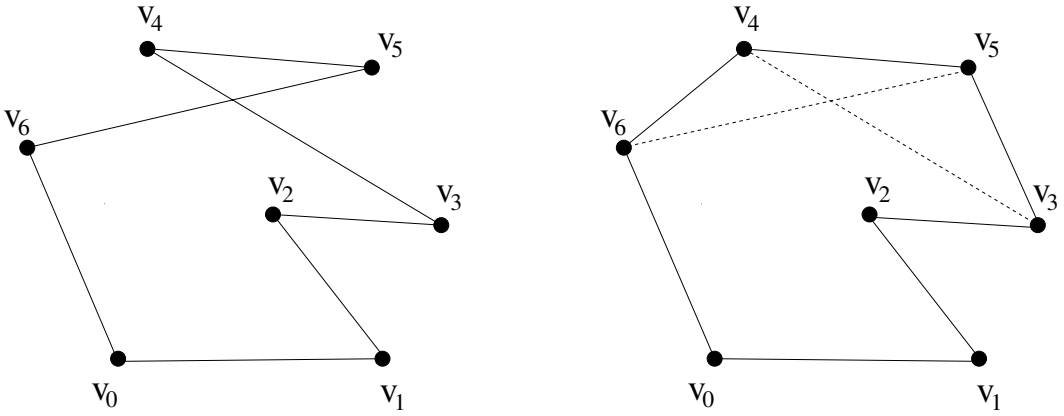
(a) Given Set of Nodes S. (b) Generated Polygon.

Figure 2.3: Illustrating Permute and Reject.

Suppose we have seven input points as shown in Figure 2.3a. If we pick a permutation $v_0, v_5, v_1, v_4, v_6, v_3, v_2$, then the implied polygon is as shown in Figure 2.3b. Clearly this permutation has to be rejected. If we pick permutation $v_0, v_5, v_1, v_6, v_2, v_3, v_4$ then no two edges intersect and the permutation is taken as accepted. This method uses an exhaustive search and check approach and is not feasible even for number of nodes around 20. The time complexity of the algorithm is exponential in the number of nodes and is not at all useful for practical implementation.

A method for generating polygons that attempts to correct intersecting edges was proposed by Auer and Held [1]. In this method, a permutation of n given vertices is picked randomly and the vertices are connected implied by the picked permutation order resulting in a polygon P. Any self intersections of P are removed by applying the so-called '2-opt moves'. The 2-opt move replaces a pair of intersecting edges (v_i, v_{i+1}) and (v_j, v_{j+1}) with the edges (v_{j+1}, v_{i+1}) and (v_j, v_i). The pair of intersecting edges for correction are picked at random.

We can illustrate this method by a specific example shown in Figure 2.4. The boundary implied by connecting permutation $v_0, v_1, v_2, v_3, v_4, v_5, v_6$ results in the intersection of one-pair of edges with edges (v_3, v_4) and (v_6, v_5). The correction is applied to replace these two intersecting edges with edges (v_3, v_5) and (v_6, v_4) which is shown in Figure 2.4b. The paper [1] does not explain the steps to be taken if the 2-opt move results in further intersections. This method also takes exponential time to check all cases and is not feasible for practical



(a) Polygon P from Random Permutation.

(b) Polygon contained after 2-opt Moves.

Figure 2.4: Illustrating 2-opt Moves.

implementation.

2.4 Generating Random Polygons

In order to perform experimental investigation of algorithms on polygonal shapes it is necessary to use randomly generated polygons. In this context it is first necessary to clarify the very notion of random polygon. As reported in [8] a polygon P is generated randomly for a given set S of n points in the plane if the probability of generating P is $1/k$, where k is the number of simple polygons that can be generated from S. The problem here is that no method is known yet to determine the value of k, hence the problem of generating a random polygon from given vertices is still an open problem. For restricted classes of polygons, a few algorithms have been reported.

2.4.1 Generating Random Monotone Polygons

The problem of generating random monotone polygons was first reported in [1]. It is noted that a simple polygon P is called monotone [5] with respect to a given direction d if the boundary of P can be partitioned into two chains such that both chains are monotone with respect to d. Figure 2.5 shows a monotone polygon which is monotone with respect to the x-axis.

In the figure the vertices are labeled in the increasing order of x-coordinates from left

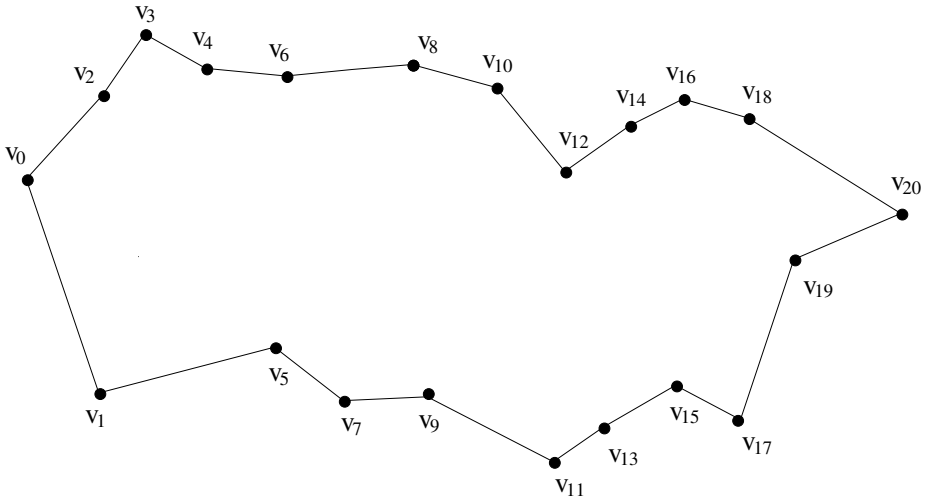


Figure 2.5: Illustrating Monotone Polygon with respect to X axis.

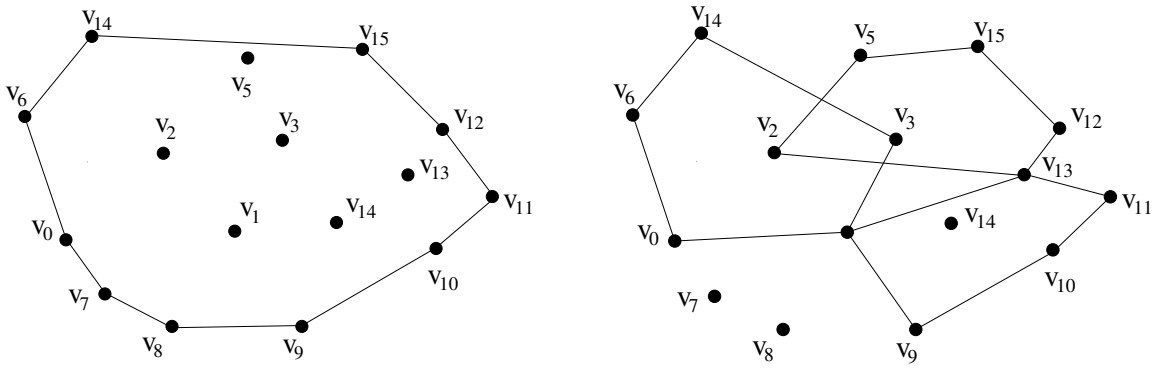
to right. The algorithm presented in [8] establishes that the number of permutations of vertices that admit monotone polygons can be counted. The counting is done by using a recursive formulation as follows.

Let S_i denote the ordered list of vertices from leftmost vertex v_0 to vertex v_i, i.e. $S_i = < v_0, v_1, ..., v_i >$. As a direct consequence of monotonicity of the polygon along the x-axis, any prefix of S_i is also a monotone polygon. The segment (v_{i-1}, v_i) could be either in the upper-chain or on the lower chain. The number of monotone polygons for S_i can be counted in terms of the number of monotone polygons for S_{i-1} and the ones formed by adding the ast segment (v_{i-1}, v_i) . Let $T(i)$ be the number of monotone polygons on S_i that have segment (v_{i-1}, v_i) in the upper chain. Furthermore, let $B(i)$ be the number of monotone polygons on S_i with (v_{i-1}, v_i) in the lower chain. It is established in [8] that the total number of monotone polygon on S_i is related to B_i and T_i. To complete B_i and T_i efficiently, one can use the visibility graph [8] of the monotone polygons. It is necessary to pre-compute B_i and T_i. Zhu et al [8] have shown that given pre-computed values of B_i and T_i, $1 < i < n$, random generation of polygons can be performed in $O(n)$ time.

2.4.2 Random Generation of Convex Polygons

Given a set of points in 2D, there is only one way to generate a convex polygon from them, and it is their convex hull. So, researchers have suggested a variation of this problem [8] in

which it is required to randomly generate a convex polygon using subsets of a given point set. If there are n given points and we need to generate a convex polygon of size $k \leq n$ then several such convex polygons can be generated. This is shown in Figure 2.6.



(a) Convex Hull from Point Sets.

(b) Convex Polygons from Subset of Point Sets.

Figure 2.6: Illustrating Random Generation of Convex Polygon.

Figure 2.6b shows only three convex polygons out of many possible ones. Here again the critical issue is counting the number of convex polygons.

Chapter 3

Inward Denting and Polygon Generation Algorithms

In this chapter we present a new approach for developing algorithms for extracting polygonal shapes for a given set of n point $S{=}v_0, v_1, ..., v_{n-1}$. Our approach called 'inward denting' constructs the shape iteratively starting from the convex hull boundary of the polygon. We then present algorithms for constructing holes inside the extracted polygon. We next propose a method for measuring the quality of the generated solution by using the induced minimum spanning tree.

3.1 Inward Denting Algorithm

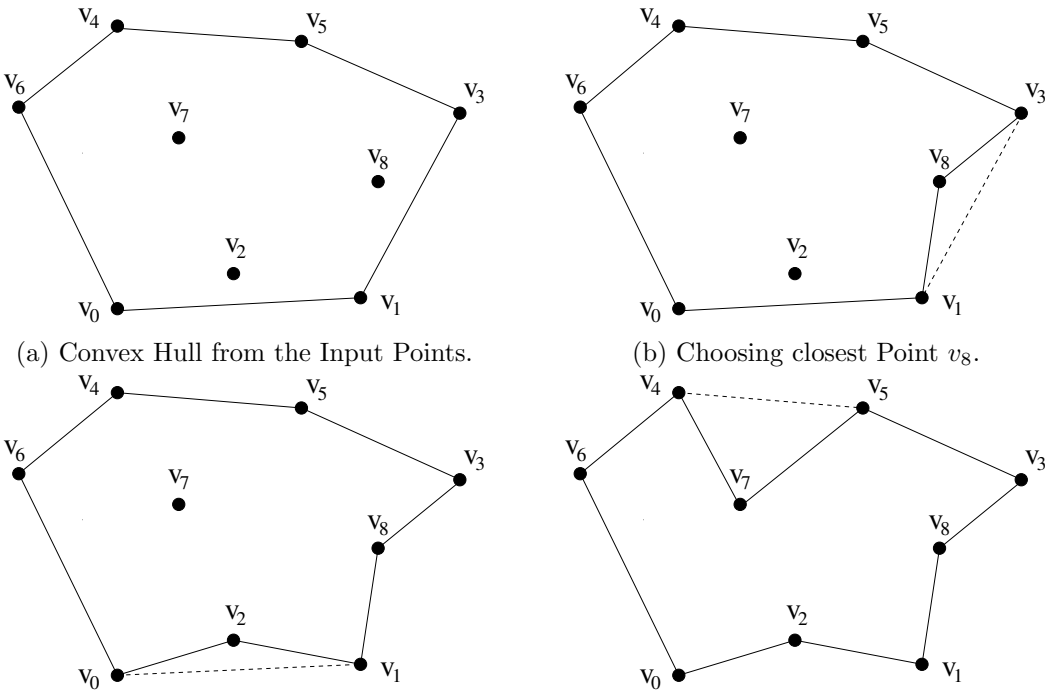
We first clarify the objectives for generating polygonal shapes. The first objective is that the nodes that are very close to each other should also appear within a small 'hop distance' along the constructed polygonal boundary. Here the hop-distance between two nodes u and w in the constructed polygonal boundary is the number of edge links between u and w in the polygonal path. The hop-distance between two nodes u and w on the polygonal boundary can be measured clockwise or counterclockwise. When we simply use the term 'hop-distance' it is taken as the smaller of the clockwise and counterclockwise hop distance. The inward denting starts from the convex hull boundary. The algorithm examines each edge in this approximate boundary and determines the *splitable edge* which is defined as

follows.

Definition 3.1 : If there are unconnected nodes inside the approximate boundary then each edge has a closest node. The distance of an edge to its closest node is called the **breaking distance**.

Definition 3.2 : The boundary edge with the smallest breaking distance is called the **splitable edge** . In Figure 3.1a, edge (v_1, v_3) is the splitable edge.

Once the splitable edge is identified, it is refined by splitng it into two edges by connecting its endpoints to its closest node as shown in Figure 3.1b, where edge (v_1, v_3) is replaced by two edges (v_1, v_8) and (v_8, v_3). The process of identification of the splitable edge and edge refinement is repeated until there is no unconnected node in the interior of the constructed boundary.



(a) Convex Hull from the Input Points.

(b) Choosing closest Point v_8.

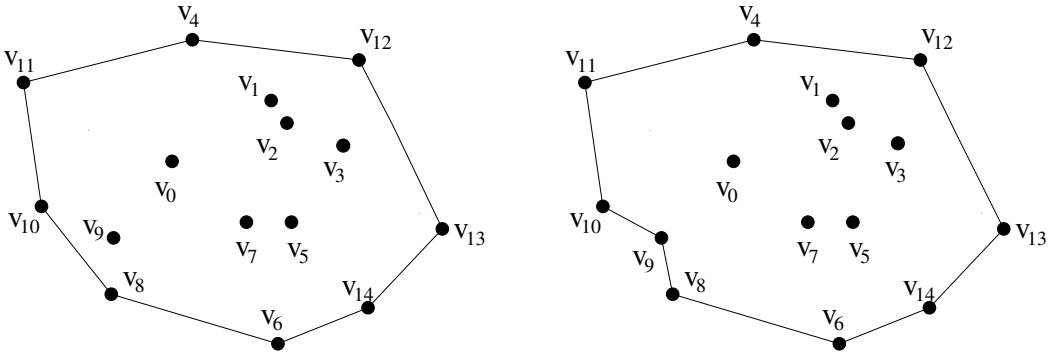
(c) Choosing closest Point v_2.

(d) Choosing closest Point v_7.

Figure 3.1: Illustrating Denting.

If we perform the inward denting operation repeatedly then we should be able to connect

all nodes in a polygonal chain. However there is a catch in this simple iteration. When the intermediately constructed polygon is complex, then the denting operation can lead to self intersecting polygonal edges as shown in Figure 3.2. The denting of an edge e that results in a self intersecting polygon is called an *invalid edge*. In Figure 3.2, edge (v_1, v_3) is an invalid edge. Such edges should be discarded and marked invalid and the next splitable edge is searched.

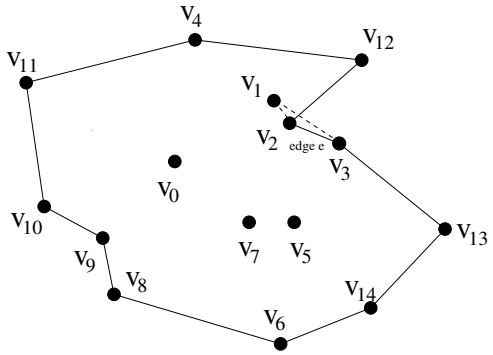


(a) Convex Hull from the Input Points.

(b) First Denting.

(c) Second Denting.

(d) Third Denting.



(e) Invalid Edge e.

Figure 3.2: Illustrating Invalid Edge.

In order to develop a formal sketch of the algorithm in a convenient and clear way, we store the boundary of polygon P in a data structure consisting of a list of nodes and a list of edges. The i^{th} node and j^{th} edge of polygon are referred as p[i].node and p[j].edge, respectively. In addition, the data structure for storing polygon P has necessary methods such as intersection of nodes, etc. The set of internal nodes Q is stored in a simple list data structure. Furthermore, each edge of the polygon boundary has methods to access start

node and end node with obvious meaning. We distinguish each edge of the polygon either as 'valid' or 'invalid'. An edge e of polygon P is valid if there is an internal node nd_1 in Q such that possible replacement edges $(e.startNode(), nd_1)$ and $(e.endNode(), nd_1)$ do not intersect with edges of P. A function to determine whether a given edge e of P containing the set of internal nodes Q is valid is listed as follows.

Function 3.1

```
bool isValid (Edge e, Polygon P, Node [ ] Q) {
    bool valid = false;
    i = 0;
    Edge e1, e2;
    while (not valid) {
        e1 = (e.start, Q[i]);
        e2 = (Q[i], e.end);
        if(notIntersecting(e1, P) and nonIntersecting(e2, P))
          valid = true;
        i++;
    }
    return valid;
}
```

This function checks for intersection of replacement edges with edges of polygon P by considering all nodes of Q as candidates for nd_1. The function $getNearestNode(..)$ returns the node of Q that is closest to edge e_1 of polygon P such that its replacement edges do not intersect the boundary of P. This function is listed as Function 3.2 .

Function 3.2

```
Node getNearestNode (Edge e1, Polygon P, Node [ ] Q){
    double d1, d2;
    Node nearestNode = null;
```

```
    d1 = largeNum;
    for (int i=0; i< Q.size; i++) {
        if(e1.valid){
            d2 = dist(e1, Q[i]);
            if(d2 < d1){
                d1 = d2;
                nearestNode = Q[i]
            }
        }
    }
    return nearestNode;
}
```

Function 3.3

```
void markValidEdges(P, Q){
    for (int i=0; i< P.size; i++) {
        if(isValid(P[i].edge,P, Q)){
            p[i].edge.valid=true;
        }
    }
```

The function $getBreakingEdge(...)$ returns the best edges via an internal node. The function essentially examines each valid edge of polygon P to determine the best one to break. The valid edge with smallest total length after replacement is considered as the best edge. This function is listed as Function 3.4 .

Function 3.4

```
Node getBreakingEdge(P, Q) {
    double d1 = largeNum;
    Edge eBreaking = null;
    for (int i=0; i < P.size; i++) {
        e1 = P[i].edge;
        if isValid( e1.valid){
            d2=getNearestNode(e1, P, Q)
            if(d1 < d2){
                d1 = d2;
                eBreaking = e1;
            }
        }
    }
    return eBreaking;
}
```

Now we are equipped with the necessary functions to present a formal description of the inward denting algorithm which is listed as Algorithm 3.1. The algorithm takes a given set S of points in 2D as input and outputs the polygonal boundary constructed from them. It increases the number of boundary edges by one by replacing the selected boundary edge with a replacement edge pair. It repeats this replacement process until there are no internal nodes left. The functions to implement most of the steps in the algorithm are listed as Function 3.1-3.4. Step 4 to compute the nearest node of a valid edge can be implemented in a straightforward manner by checking the distance to internal node set Q from edge e.

This takes $O(n)$ time.

Algorithm 3.1 Inward Denting Algorithm

Input: A set of point nodes $S = v_0, v_1, ..., v_{n-1}$ in 2D.

Output: A polygonal shape with nodes is S.

Step 1: **a.** Let P be the convex hull boundary of points in S.

b. $Q = S - P$;

Step 2: while (Q is not empty) {

Step 3: $e = getBreakingEdge(P, Q)$;

Step 4: $w = getNearestNode(Q, e)$;

Step 5: **a.** $insertNode(P, e, w)$;

b. $removeNode(Q, w)$;

Step 6: $markValidEdges(P, Q)$;

}

Step 7: Output P.

The time complexity of Inward Denting Algorithm (Algorithm 3.1) can be analyzed as follows. Convex hulls can be computed in $O(nlogn)$ time [5]. Hence Step 1 takes $O(nlogn)$ time. Step 4 and Step 5 each take $O(n)$ time. Step 3 is one of the most expensive steps in the while loop which takes $O(n^2)$ time. Step 6 takes $O(n^2)$ time. One execution of the while loop eliminates one internal node and hence the while loop iterates in $O(n)$ time. The total time for the whole algorithm adds up to $O(n^3)$.

3.2 Voronoi Based Inward Denting Algorithm

The time complexity of Algorithm 3.1 is $O(n^3)$ which is rather high. If we closely examine the functions invoked by Algorithm 3.1 we find that we could reduce its time complexity if we could find a faster way of computing the nearest neighbor of a node. It is remarked that the time complexity of function $getNearestNode(Q, e)$ is $O(n)$. The nearest neighbor of each node of the partially constructed polygon is repeatedly computed to determine the edge to split into two edges. This overhead for obtaining nearest neighbors can be

reduced by pre-computing the nearest neighbor of each edge and storing them for future use. A straight-forward approach for pre-computing all nearest neighbors is to determine nearest neighbors for each node separately, which leads to $O(n^2)$ time. The Voronoi diagram induced by input points can be used to determine the nearest neighbors of all nodes in S. This is illustrated in Figure 3.3, where the overlay of Voronoi Diagram and the partial polygon connecting input nodes is shown. The nearest node of a polygonal node could be one of the polygonal or internal nodes. In order to find candidate internal nodes we should avoid polygonal nodes. To illustrate this we can inspect Figure 3.3, where the nearest node for edge (v_2, v_9) is v_7 which is a polygonal node. In such situations we need to examine all neighbors of the end points of a candidate edge in the Voronoi Diagram. What happens if all Voronoi neighbor of a candidate polygonal node are polygonal nodes? In such situations we can examine all $k - hop$ neighbors (say $k = 3$) of the candidate polygonal edge to the determine nearest internal node. A function based on this technique, which we call *getNearestNodeViaVD(...)*, is listed as Function 3.5. The function takes candidate edge e_1, partially constructed polygon P, list of internal nodes Q, and Voronoi diagram VD of input points as arguments and returns the nearest internal node. This function is essentially an improved version of Function 3.2.

Function 3.5

```
Node getNearestNodeViaVD(Edge e1, Polygon P, Node [ ] Q, Voro VD){
    double d1, d2;
    Node nearestNode = null;
    Let R be the k − hop neighbors of e1 in VD
    nearestNode = nearestValid(e1, R, P, VD)
    return nearestNode;
}
```

Similarly, functions *getBreakingEdgeViaVD(...)* and *getNearestNodeViaVD(...)* can be obtained as improved versions of getBreakingEdge(...) and *getNearestNode(...)*. When a node is inserted into the partially constructed polygon P, the two new edges could be valid or invalid. This can be checked in $O(n)$ time by using the Voronoi diagram. An improved

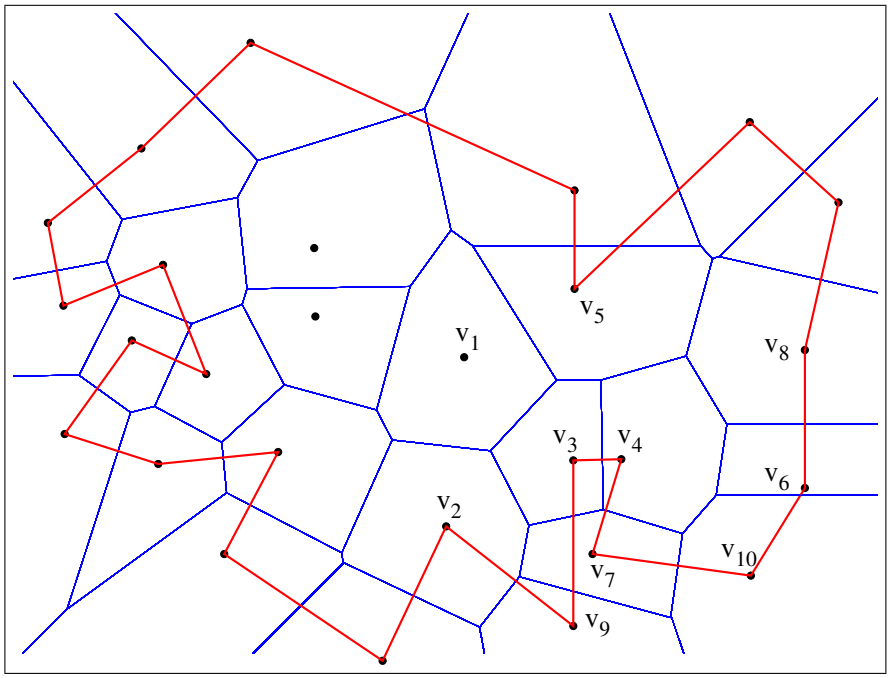


Figure 3.3: Overlay of Voronoi Diagram and Partially Constructed Polygon.

version of Algorithm 3.1 is sketched as Algorithm 3.2. The time complexity of this algorithm can be analyzed as follows. Step 1 and Step 2 can be implemented in $O(nlogn)$ time [5] . Since only $k-hop$ neighbors of a node are examined by using the pre-computed Voronoi diagram, Step 4 and Step 5 can be done in $O(n)$ time and $O(1)$ time respectively. Step 6 takes $O(1)$ time. Since each partially constructed edge needs to be checked for possible intersection, each execution of Step 7 takes $O(n^2)$ time. Hence the total time of Algorithm 3.2 is $O(n^2)$.

Algorithm 3.2 Improved Inward Denting Algorithm

Input: A set of point nodes $S = v_0, v_1, ..., v_{n-1}$ in 2D.

Output: A polygonal shape with nodes is S.

Step 1: a. Let P be the convex hull boundary of points in S.

b. $Q = S - P$;

Step 2: Let VD be the Voronoi Diagram of points in S.

Step 3: while (Q is not empty) {

Step 4: $e = getBreakingEdgeViaVD(P, Q, VD)$;

Step 5: $w = getNearestNodeViaVD(e, P, Q, VD)$;

Step 6: **a.** $insertNode(P, e, w)$;

b. $removeNode(Q, w)$;

Step 7: $markAdjacentValidEdges(P, Q)$;

}

Step 8: Output P.

3.3 Modeling Holes

Most of the algorithms for constructing polygons, reported in computational geometry literature, do not consider polygons with holes. In this subsection we present an approach that can be used to construct holes inside a polygonal boundary. Intuitively, holes are the regions which have no node in their interior. Nodes around empty regions can be connected (some how) to model a hole. This is illustrated in Figure 3.4.

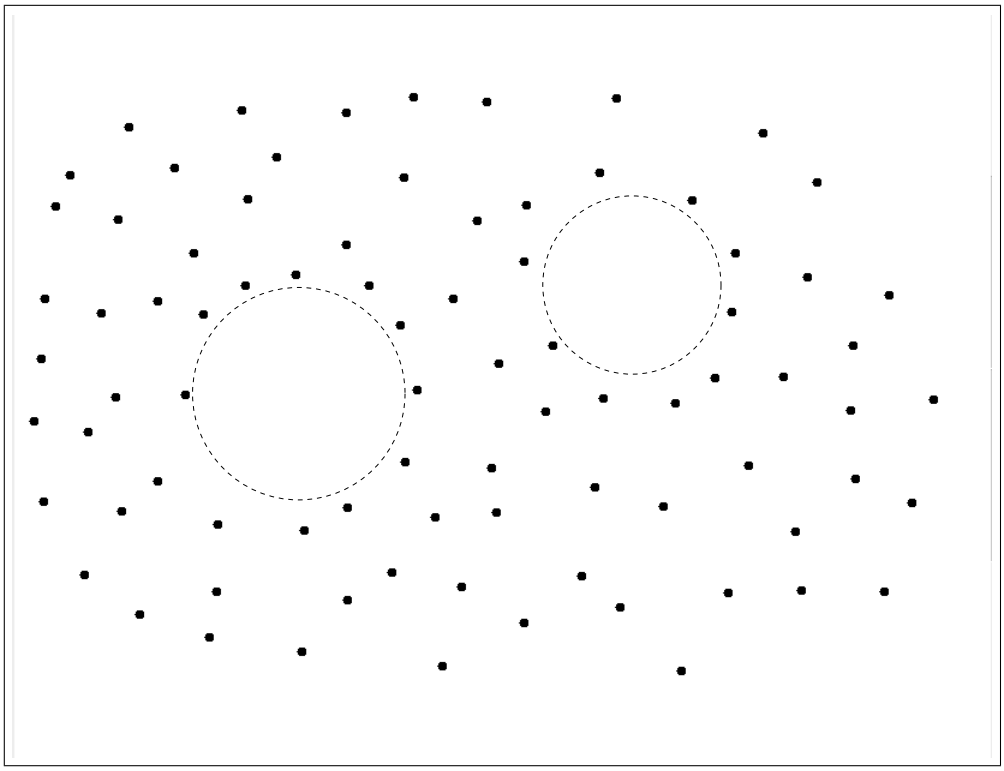


Figure 3.4: Showing Empty Spots in a Node Distribution.

In the figure, there are two spots indicated by dashed circles, which are distinctively empty regions. This gives us a hint that nodes in the vicinity of the empty spots can be connected in some effective way to model holes.

Consider the empty largest circle that is contained completely inside the convex hull of input nodes. The left empty circle is such a circle for node distribution in Figure 3.4. To construct the largest empty circle inside the convex hull we can use the Voronoi diagram of input points.

Figure 3.5 shows the Voronoi diagram of input points of Figure 3.4. From the properties of Voronoi diagram it is known [5] that the vertex of a Voronoi diagram is the center of an empty circle defined by three nodes. In Figure 3.5, the Voronoi node corresponding to the largest empty circle is shown as an unfilled dot. So, the approach is to identify the center of the largest empty circle corresponding to each Voronoi vertex and select the one that has the largest area. Only those Voronoi vertices are examined whose corresponding empty circles lie completely inside the convex hull. Once the largest empty circle C_L is

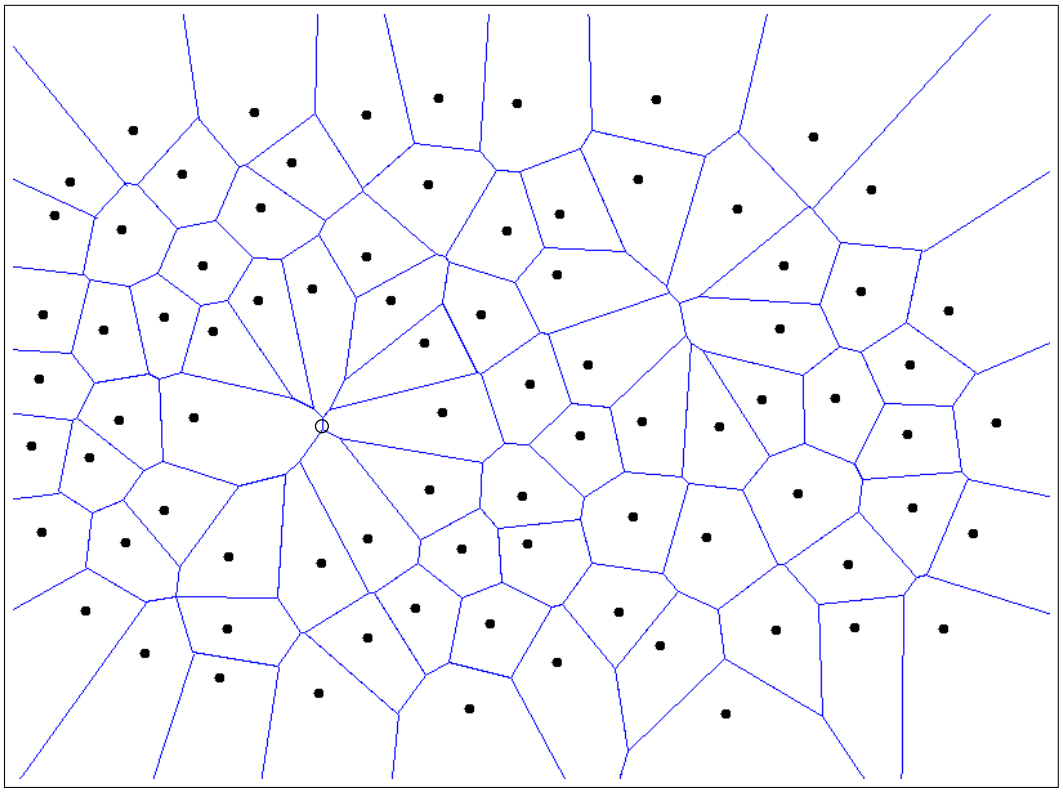


Figure 3.5: Voronoi Diagram of the Empty Spots.

identified, we can start constructing holes by connecting three nodes corresponding to C_L. The empty triangle T_L corresponding to C_L can now be dented outward to construct a hole. A procedure similar to the inward denting algorithm described in subsection 3.1 can be used to achieve outward denting. Some threshold factor can be predetermined to stop the outward denting process. Figure 3.7 shows the progress of the outward denting procedure and the construction of the final hole.

A formal sketch of the hole construction algorithm can be listed as follows.

Algorithm 3.3 Hole Construction Algorithm

Input: A set of point nodes $S = v_0, v_1, ..., v_{n-1}$ in 2D.

Output: A polygonal H modeling hole.

Step 1: Compute the Voronoi diagram VD of S.

Step 2: Compute the convex hull CH of S.

Step 3: For each vertex w_i in VD do

C_i = Empty Circle for w_i.

Step 4: Find the largest empty circle C_L among C_i's that lies inside CH.

Step 5: Let C_T be the triangle corresponding to C_L.

Step 6: Perform $k(k = 3$,say) outward denting on C_T to construct H.

Step 7: Output H as hole.

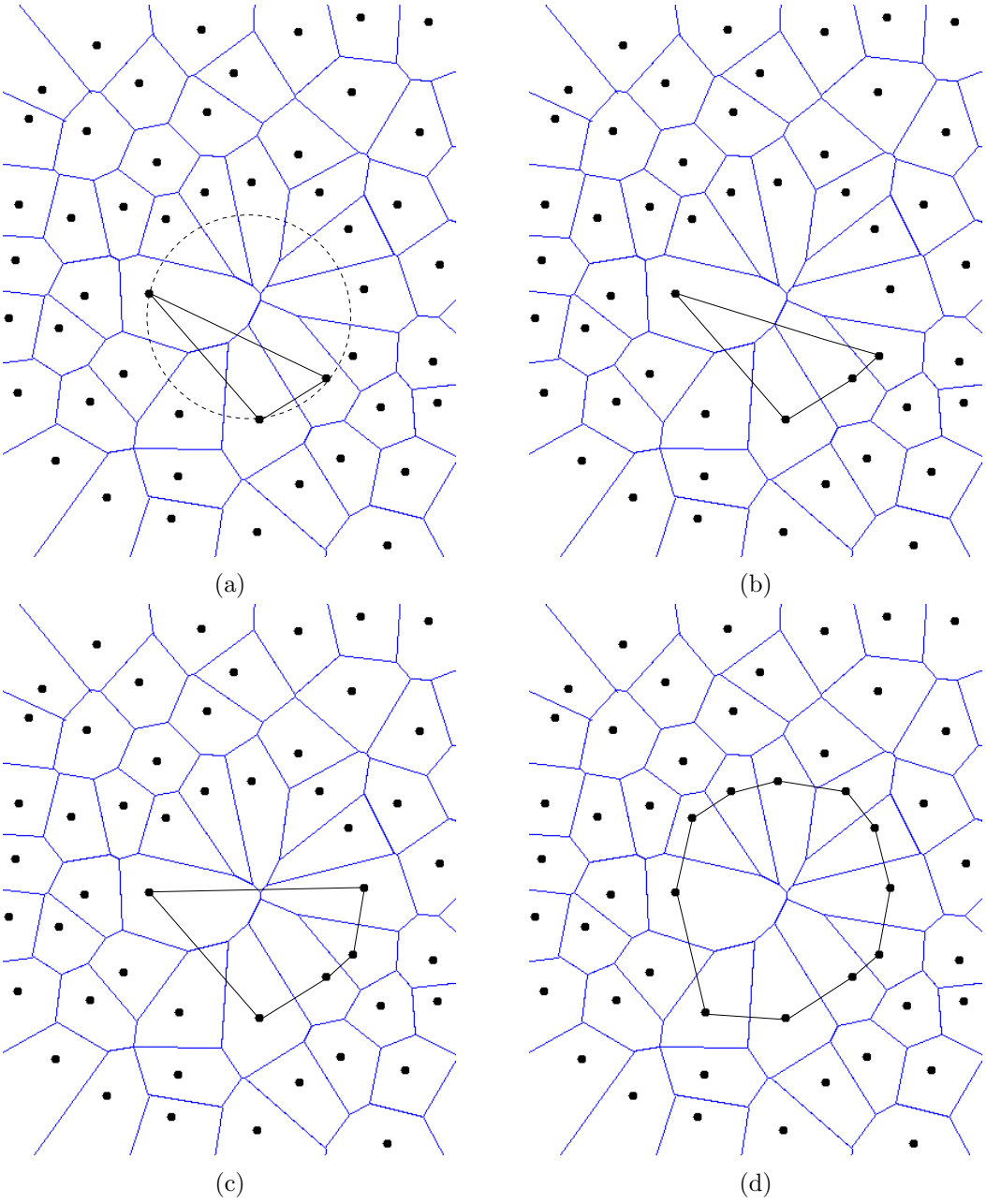
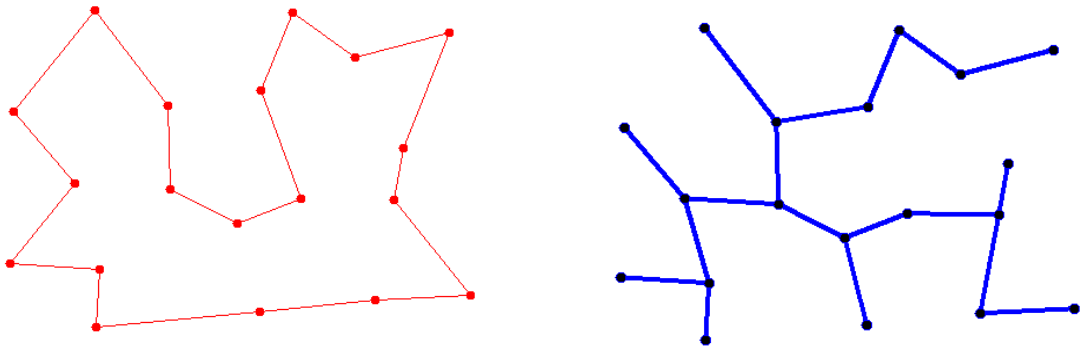


Figure 3.6: Construction of Hole using Largest Empty Circle.

3.4 Measuring Solution Quality

It is very tempting to measure the quality of the solution obtained by inward denting algorithm. For this purpose we need to first set the objective criteria. We set the length of the boundary of the generated polygon as the quality to minimize. It is obvious that

the boundary of the polygon can be taken as the route that visits all the nodes, one node exactly one time. So, we could compare the boundary length of the generated polygon with the length of the Euclidean minimum spanning tree. Let us illustrate with an example in Figure 3.7 where the left part shows the boundary of the generated polygon and the right part shows the Euclidean minimum spanning tree.



(a) (b)

Figure 3.7: Comparing Polygon boundary with the Minimum Spanning Tree.

Let T denote the length of the minimum spanning tree induced by input point sites. Let $L1$ denote the length of the boundary of the generated polygon. If we remove one edge from the boundary of the generated polygon then we get a new spanning tree. Let $L2$ be the length of the chain obtained by removing the largest edge from the boundary of the polygon. It is straightforward to observe that $L2 \geq T$. Hence we can get an estimate of the quality of the generated solution by comparing the values of $T, L1$ and $L2$.

Chapter 4

Implementation

In this chapter we present an implementation of algorithms proposed in Chapter Three. The implementations include (i) inward-denting algorithm for generating 2-d shapes, (ii) assessment of the generated shape, and (iii) methods for modeling holes. All implementations are done in Java programming language. The top layer of the program is a user friendly graphical interface. Users can interact with the program intuitively by mouse clicks and drop-down menus. The prototype programs are implemented both in desktop environment and for Android tablet devices.

4.1 GUI Description

The main graphical user interface is formed by importing the JFrame object from javax.swing. The main frame is partitioned into four panels: top, left, middle, and right as shown in the layout of Figure 4.1. The top panel contains drop down menus for file and other functionalities. The central panel is used to display the graphics of input data and generated output. The left panel contains various check boxes so that users can specify the state of the program and other related properties. The right panel contains various buttons and text boxes.

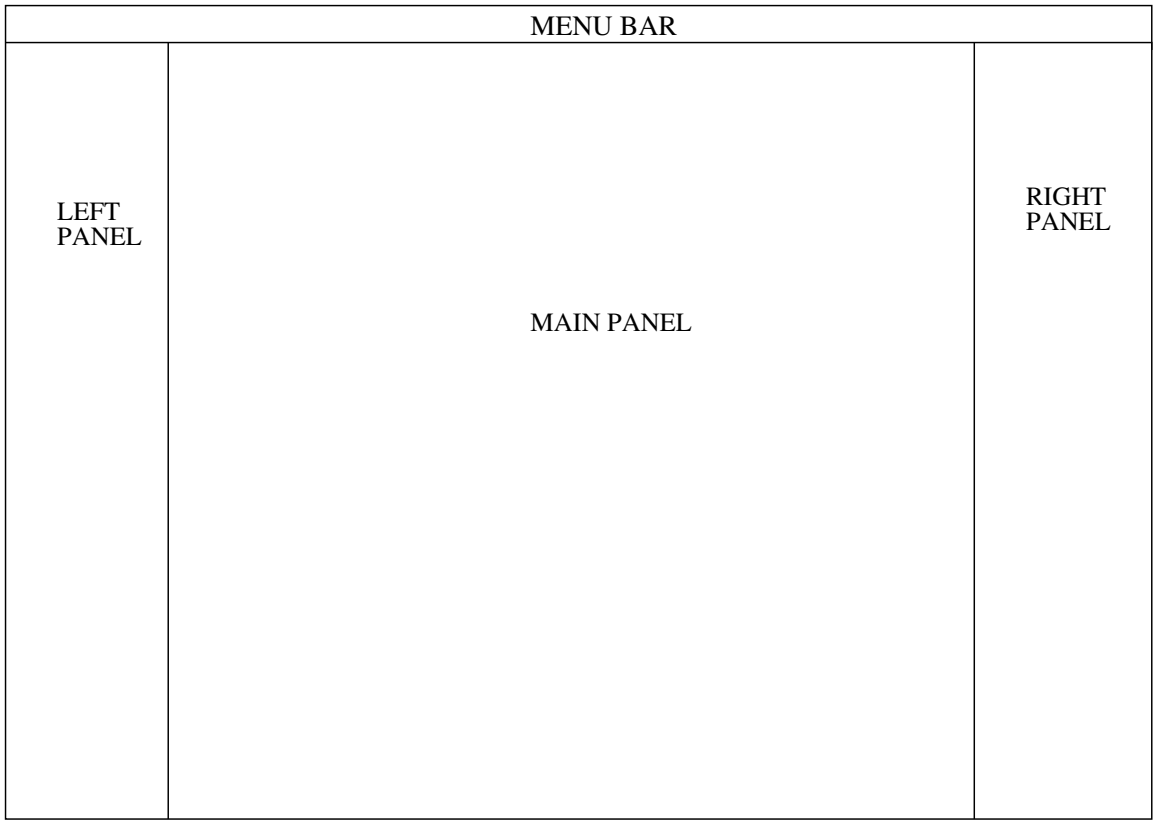


Figure 4.1: Layout of Main User Interface.

4.2 Interface Description

Figure 4.2 shows a snap-shot of the actual top level user interface of the program. The file menu on the top panel allows users to (i) read object data from an existing file, (ii) save the object data to a file system, (iii) save the object data in xfig format, and (iv) exit the application. A brief description of the functionalities of the file menu items is listed in Table 4.1 . Users can plot nodes by enabling the *Draw node* checkbox. The nodes can be edited and a 2D shape can be drawn with it. To actually draw a node in the main panel, the user can use the mouse. The mouse position cursor arrow is displayed on the draw canvas and suggests where the user can draw a node. The coordinates of the position of the mouse cursor are displayed in the upper left corner of the draw canvas. When the user clicks the left button of the mouse a small black-filled circular dot is drawn there. The corresponding co-ordinates in ASCII characters are displayed on the textbox which is contained on the

right panel. Figure 4.2 is a snap-shot of the actual interface which shows 10 vertices entered by a user via mouse clicks. If the user wants to change the position of one or more nodes, this can be done by checking the *Edit vertex* box in the left panel. When *Edit Vertex* box is checked and the left button of the mouse is pressed and dragged, the vertex nearest to the cursor changes its position following the position of the mouse cursor. A brief description of the functionalities of the check box items in the left panel are listed in Table 4.2. Similarly, the functionalities of the buttons contained in the right panel are listed in Table 4.3.

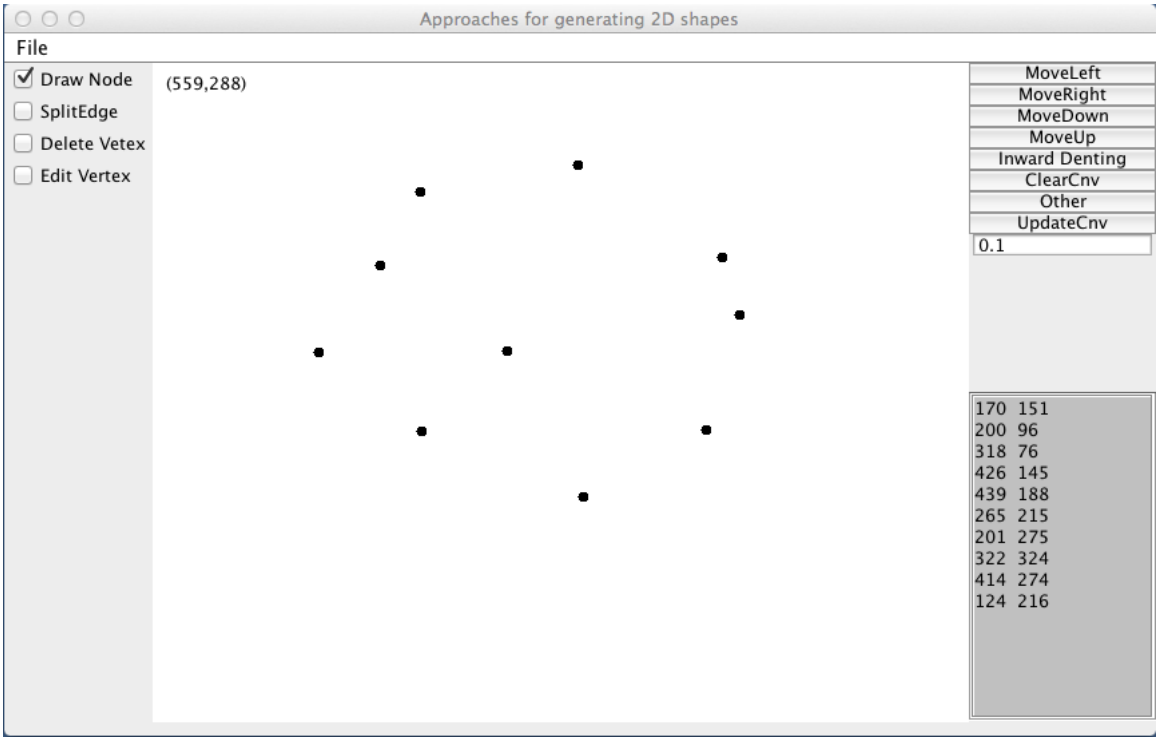


Figure 4.2: Graphical User Interface.

Table 4.1: File Menu Items Description.

S.N.	File Menu Items	Functionalities
1	Read File	Brings up a dialogue box to allow the user to select a pre-saved file.
2	Save File	Brings up a dialogue box up to allow the user to save the diagram.
3	Save Xfig File	Brings up a dialogue box to allow the user to save the diagram in fig format.
4	Exit	Exits the application

Table 4.2: Checkbox Items Description.

S.N.	Menu Item	Functionalities
1	Draw Vertex	Allows users to draw vertices on the mainPanel.
2	Edit Vertex	Allows users to edit drawn vertices.
3	Minimum Spanning Tree	Displays the Minimum Spanning Tree of the points.
4	Voronoi Diagram	Displays the Voronoi Diagram of the points.
5	Inward Denting	Display the Polygon including all the points by using Inward Denting Approach.
6	Circum Circles	Draws circumcircles around the triangles in the mesh.

Table 4.3: Button Description.

S.N.	Menu Item	Functionalities
1	Clear Canvas	Clears the main panel
2	Random	Draws random set of points on the main panel

4.3 Execution of Denting Algorithm

After the nodes are displayed in the draw canvas, the user can execute the inward denting algorithm. It is noted that the nodes on the canvas can be entered either by mouse click or read from a previously saved node coordinate file. When the inward denting checkbox is checked and the mouse movement is detected on the draw canvas, the inward denting algorithm is invoked and the resulting polygonal shape is displayed by connecting the nodes. Figure 4.3 shows a snap-shot of the polygonal shape generated by the inward denting algorithm. When the denting algorithm proceeds by connecting nodes, some intersecting edges can be formed in some rare cases. The algorithm can be executed in two versions. In the normal version, the final polygon is displayed as in Figure 4.3. In the tracing version, the intersecting edges are displayed for verification as shown in Figure 4.4. Of course the correction is made to the intersecting edges by following the replacement method described in Chapter Three. Figure 4.5 shows the Class Interface Diagram of the implemented program.

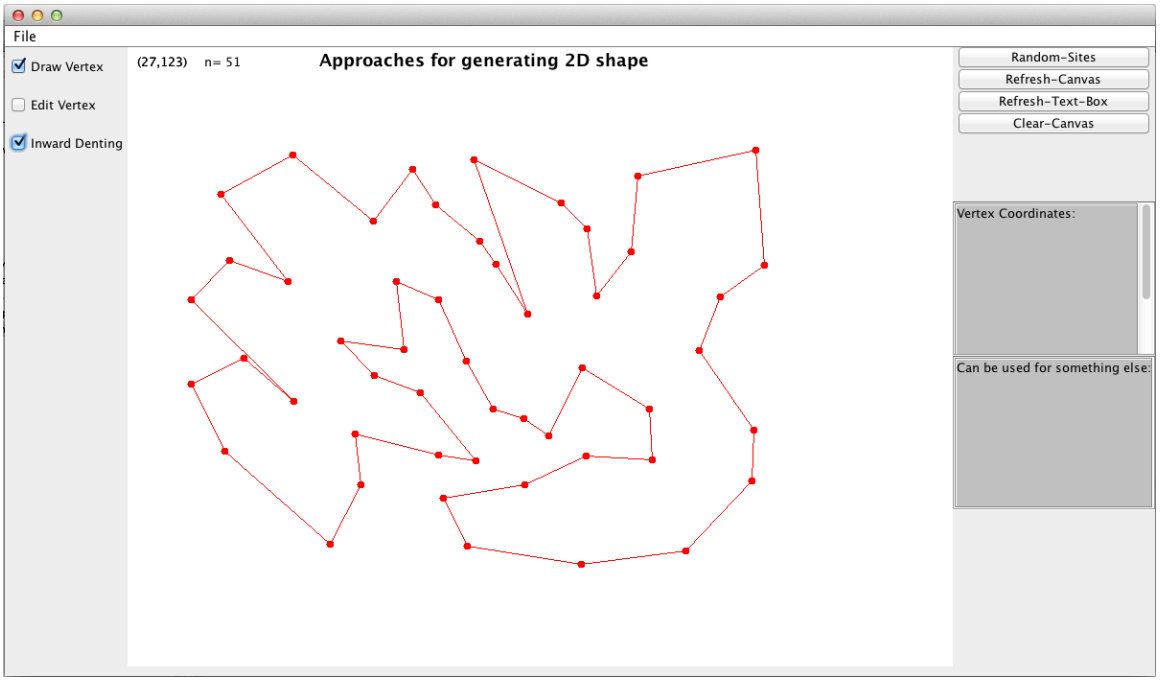


Figure 4.3: Normal Output of Inward Denting Algorithm.

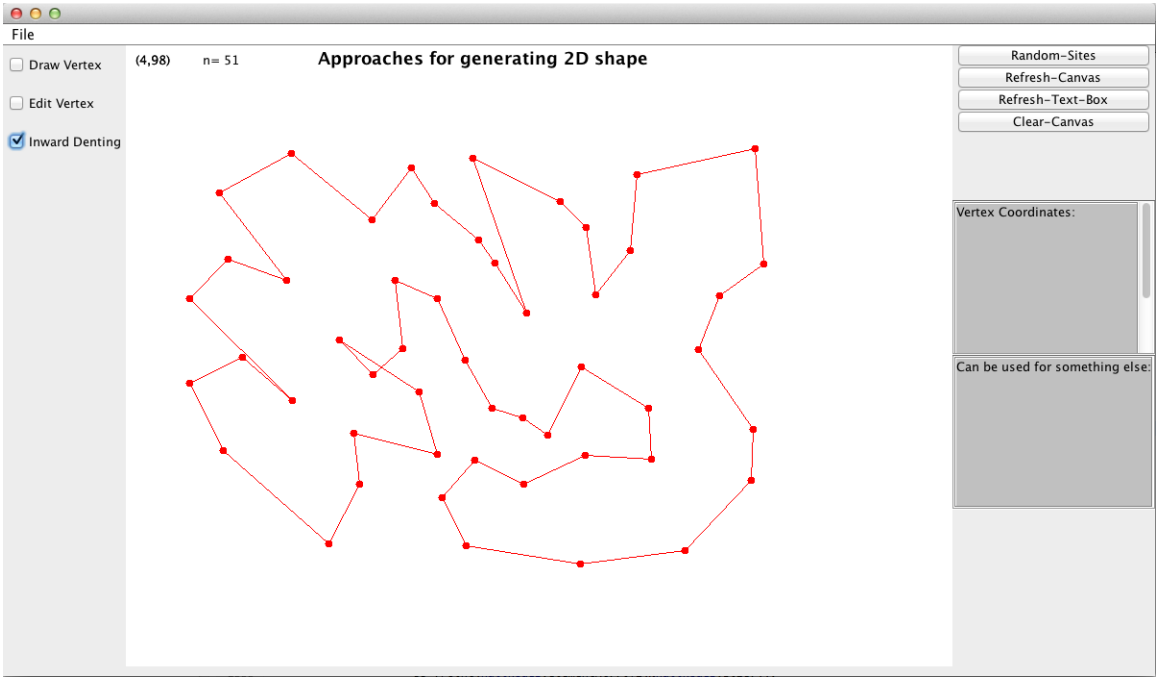


Figure 4.4: Output from Tracing Version of Inward Denting Algorithm.

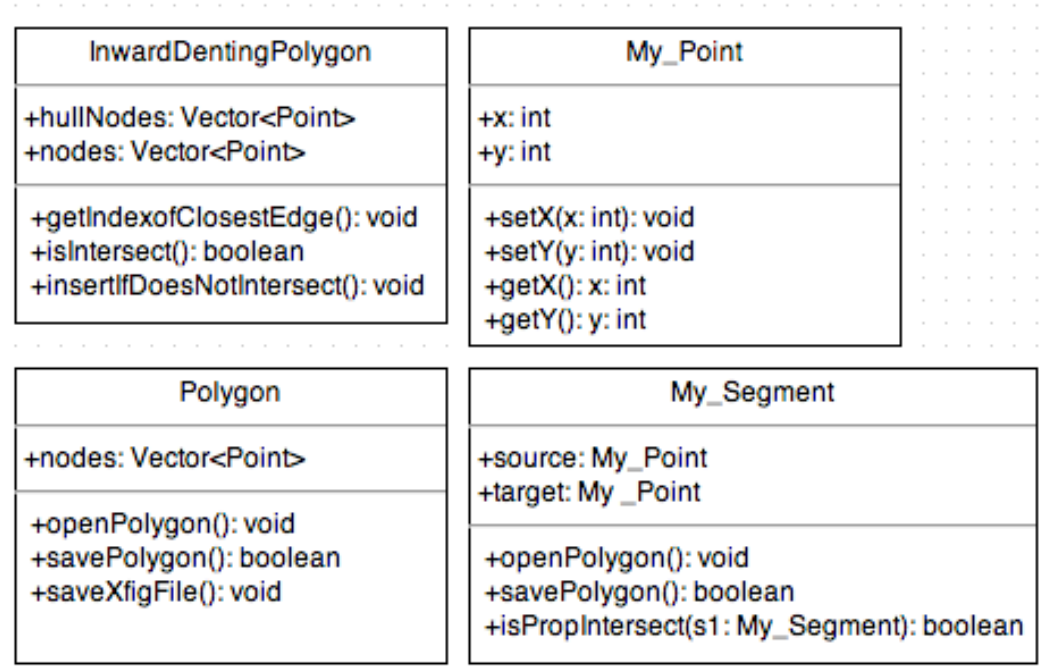
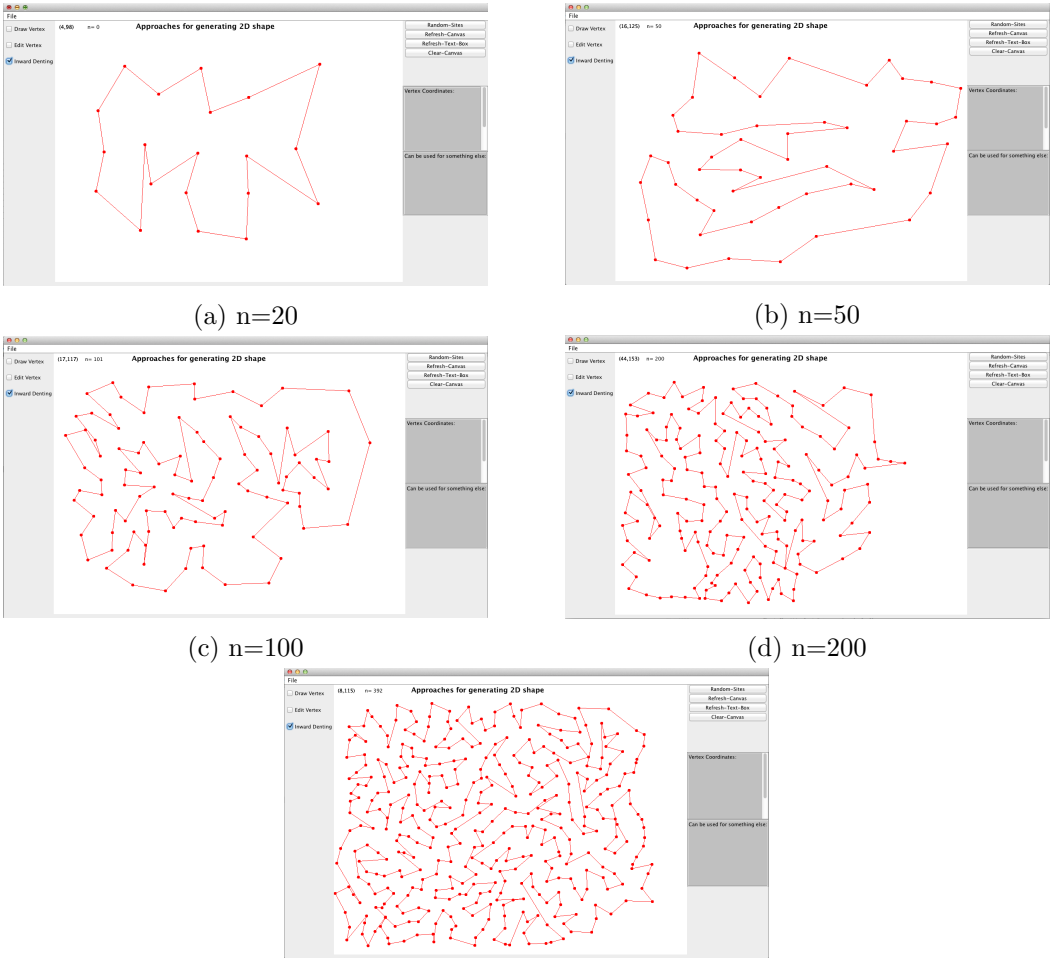


Figure 4.5: Class Interface Diagram of the Implementation.

We generated nodes randomly. For this purpose two integers in the range 0-1000 were randomly generated that represent the x- and y- coordinates of a random node. For random generation, Java function Math.Random() available in the Java language library was used. The snap-shots of the polygonal shapes generated for the number of nodes n = 20, 50, 100, 200, 500, 1000 are shown in Figure 4.6.



(a) n=20 (b) n=50

(c) n=100 (d) n=200

(e) n=500

Figure 4.6: Generated Polygons for Various Number of Nodes.

4.4 Results and Statistics

We generated various polygons to test the performance of inward denting algorithm. We used the method described in Section 3.4, Chapter Three for measuring the performance. The length of the boundary of the polygon (L_1), the length of the polygon tree (L_2), and the length of the minimum spanning tree (T) were measured corresponding to each randomly generated point sites. Five set of point sites with number of points n = 10, 20, 30, 40, 50, 100, 200, 300, 400, and 500 were considered. The values of $R_1 = (L_1/T)$ * 100, and $R_2 = (L_2/T)$ * 100 for these points are listed in Table 4.4. A plot of R_1, for various values of n is shown in Figure 4.7.

Table 4.4: Comparing Polygon and MST with 10 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	$R_1 = (L_1/T) * 100$	$R_2 = (L_2/T) * 100$
10	1981	1706	1417		
	1246	1011	986		
	2114	1917	1805		
	1762	1379	1333		
	1877	1317	1251		
	1796	**1466**	**1358.4**	**132.21**	**107.92**

Table 4.5: Comparing Polygon and MST with 20 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	$R_1 = (L_1/T) * 100$	$R_2 = (L_2/T) * 100$
20	2661	2444	2144		
	2606	2379	2130		
	3124	2774	2304		
	2753	2245	2004		
	2786	**2460.5**	**2145.5**	**129.85**	**114.68**

Table 4.6: Comparing Polygon and MST with 30 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	R_1= $(L_1/T) * 100$	R_2= $(L_2/T) * 100$
30	4296	3963	3205		
	3318	2978	2409		
	3581	3294	2813		
	3924	3681	2822		
	3191	2892	2572		
	3662	**3361.6**	**2764.2**	**132.47**	**121.61**

Table 4.7: Comparing Polygon and MST with 40 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	R_1= $(L_1/T) * 100$	R_2= $(L_2/T) * 100$
40	5034	4740	3988		
	4298	3958	3160		
	4279	4057	3497		
	4516	4285	3425		
	3671	3453	2845		
	4359.6	**4098.6**	**3383**	**128.86**	**121.15**

Table 4.8: Comparing Polygon and MST with 50 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	R_1= $(L_1/T) * 100$	R_2= $(L_2/T) * 100$
50	5468	5174	4314		
	4935	4687	3731		
	4928	4650	3849		
	4760	4519	3711		
	3861	3691	3098		
	4790.4	**4544.2**	**3740.6**	**128.06**	**121.48**

Table 4.9: Comparing Polygon and MST with 100 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	R_1= $(L_1/T) * 100$	R_2= $(L_2/T) * 100$
100	5849	5712	4652		
	6116	6400	4994		
	6805	6602	5131		
	6798	6607	5251		
	6194	6057	5086		
	6352.4	**6275.6**	**5022.8**	**126.47**	**124.94**

Table 4.10: Comparing Polygon and MST with 200 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	R_1= $(L_1/T) * 100$	R_2= $(L_2/T) * 100$
200	8841	8674	6888		
	9131	8933	6918		
	8978	8836	6969		
	8935	8805	6747		
	9230	9063	7069		
	9023	**8862.2**	**6918.2**	**130.42**	**128.09**

Table 4.11: Comparing Polygon and MST with 300 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	R_1= $(L_1/T) * 100$	R_2= $(L_2/T) * 100$
300	11119	10938	8544		
	11055	10784	8258		
	11110	10943	8423		
	10689	10187	8169		
	10915	10656	8483		
	10977.6	**10701.6**	**8375.4**	**131.06**	**127.77**

Table 4.12: Comparing Polygon and MST with 400 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	R_1= $(L_1/T) * 100$	R_2= $(L_2/T) * 100$
400	12652	12362	9741		
	12643	12489	9897		
	12655	12516	9727		
	12704	12567	9745		
	11998	11891	9665		
	12530.4	**12365**	**9755**	**128.45**	**126.75**

Table 4.13: Comparing Polygon and MST with 500 Nodes.

No of nodes	Length of Polygon Boundary (L_1)	Length of Polygon Tree (L_2)	Length of MST (T)	R_1= $(L_1/T) * 100$	R_2= $(L_2/T) * 100$
500	14384	14259	11333		
	14180	14050	11341		
	14630	14453	11372		
	14663	14457	11227		
	14729	14504	11352		
	14517.2	**14344.6**	**11325**	**128.18**	**126.66**

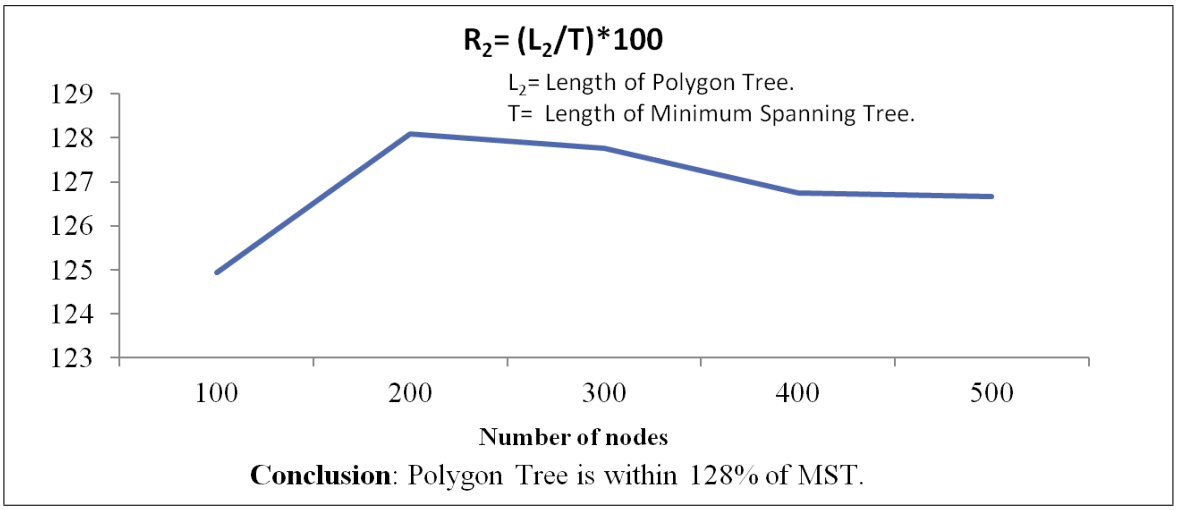


Figure 4.7: Plot of the Length of the Polygon Tree compared to the MST.

Chapter 5

Conclusion and Discussion

We presented a critical review of published algorithms for generating polygonal shapes from a given set of point nodes in two dimensions. We formulated a new approach for generating polygonal shapes for a given set of nodes. The formulated approach leads to an algorithm based on the concept of *inward denting* starting from the convex hull boundary. The first version of the algorithm performs the denting process by selecting splitable edges that minimize the distance to the nearest node. In this version, the nearest node is computed at each iteration. In the second version of the inward denting algorithm, the algorithm makes use of the precomputed Voronoi diagram of input nodes. The improved algorithm picks the nearest node by walking through the Voronoi polygons of the nodes in the proximity of the endpoints of the candidate edge. This leads to a faster algorithm for performing of the denting process. The time complexity of the improved denting algorithm is $O(n^2)$.

We conceptualize a new approach for modeling polygons with holes. The hole modeling algorithm uses the largest empty circle to locate the region where a hole can be present inside the convex hull. To locate the center of candidate empty circles, the hole modeling algorithm makes use of the Voronoi diagram. The vertices of the Voronoi diagram inside the convex hull boundary are taken as the center of a possible empty circle. The algorithm examines the empty circles among all Voronoi vertices in the interior of the convex hull to identify the largest empty circle. The empty triangle T_0 is dented outward to connect more nodes to the boundary of the hole. This algorithm was not implemented.

We presented experimental results for constructing 2D polygonal shapes for various numbers

of input point sites. The length of the boundary of the generated polygon was compared with the length of the corresponding minimum spanning tree. The nodes were generated randomly. The analysis was done on 20 sets of data and the results show that the length of the perimeter of the generated polygon is within 33% of the length of the minimum spanning tree induced by the input points. The denting approach can be extended to develop polygonal shapes that have some pre-specified properties. For example, we could modify the inward denting algorithm for generating monotone polygons, monotone in a given direction. Another useful extension would be to apply the algorithm to three dimensions. Then the faces of the surface can be dented inward to include interior nodes. The details would be very complicated but can be pursued to develop an effective algorithm.

The hole modeling approach suggested in Chapter Three is not adequate enough to capture complex shaped holes. A thin and long hole can not be modeled by a straightforward application of the largest empty circle. It would be interesting to come up with an effective method for capturing holes with complicated shapes.

Bibliography

[1] Thomas Auer and Martin Held, "Heuristics for the Generation of Random Polygons." *Proceedings of Eighth Canadian Conference on Computational Geometry*, pp. 38-44, 1996.

[2] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf *Computational Geometry : Algorithms and Applications*, Springer, 1997.

[3] W. Berger and M. J. Bursa, *Principles of Digital Image Processing*, Springer, 2013.

[4] Robin Kerrod, *The Book of Constellations: Discover the Secrets in Stars*, Barron's Educational Series, 2002.

[5] J. O'Rourke, *Computational Geometry in C, Second Edition*, Cambridge University Press, 1998.

[6] J. O'Rourke, J. Booth, and R. Washington, "Connect the Dots: A New Heuristic", *Computer Vision, Graphics, and Image Processing*, 39(1987) pp. 258-266.

[7] M. Shunji, N. Hirobumi, and Y. Hiromitsu, *Optical Character Recognition*, Wiley 2007.

[8] Chong Zhu, Gopalakrishnan Sundaram, Jack Snoeyink and, Joseph S.B. Mitchell, "Generating Random Polygon with Given Vertices." *Computational Geometry: Theory and Applications*, Vol. 6, pp. 277-290, 1996.

Vita

Graduate College

University of Nevada, Las Vegas

Pratik Shankar Hada

Degrees:

Bachelor of Engineering in Computer Engineering 2007

Tribhuvan University

Institute of Engineering, Pulchowk Campus

Thesis Title: Approaches for Generating 2D Shapes

Thesis Examination Committee:

Chairperson, Dr. Laxmi Gewali, Ph.D.

Committee Member, Dr. Ajoy Datta, Ph.D.

Committee Member, Dr. John Minor, Ph.D.

Graduate Faculty Representative, Dr. Rama Venkat, Ph.D.