May 2019

Generating Kernel Aware Polygons

Bibek Subedi
subedishankar2011@gmail.com

GENERATING KERNEL AWARE POLYGONS

By

Bibek Subedi

Bachelor of Engineering in Computer Engineering
Tribhuvan University
2013

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

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

University of Nevada, Las Vegas
May 2019

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 18, 2019

This thesis prepared by

Bibek Subedi

entitled

Generating Kernel Aware Polygons

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Laxmi Gewali, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

John Minor, Ph.D.
Examination Committee Member

Kazem Taghva, Ph.D.
Examination Committee Member

Henry Selvaraj, Ph.D.
Graduate College Faculty Representative

Abstract

Problems dealing with the generation of random polygons has important applications for evaluating the performance of algorithms on polygonal domain. We review existing algorithms for generating random polygons. We present an algorithm for generating polygons admitting visibility properties. In particular, we propose an algorithm for generating polygons admitting large size kernels. We also present experimental results on generating such polygons.

Acknowledgements

I am grateful to my thesis advisor Dr. Laxmi Gewali for his continuous guidance and support throughout this thesis. His dedication and willingness to make himself available whenever I need his support has been very much appreciated. I would like to thank Dr. Kazem Taghva, Dr. John Minor, and Dr. Henry Selvaraj for being in my committee and providing me with valuable suggestions. I would like to provide my special thanks to Mario and Leslie for supporting me on doing administrative tasks needed for this thesis.

To my family, thank you for encouraging and inspiring me to follow my dreams. Your love and support always keeps me motivated and directs me in the right path.

At last, I would like to thank all my friends in Las Vegas for being there whenever I need and supporting me emotionally and socially throughout this thesis. Special thanks to my roommates Jiwan and Laxmi for your advice, suggestions, and support in the entire process.

Bibek Subedi

University of Nevada, Las Vegas

May 2019

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

Chapter 1

Introduction

Development of efficient algorithms for generating simple polygonal shapes has wealth of applications in applied computational geometry. Any algorithm that takes input a simple polygon should be tested for its performance by executing on several polygons with different numbers of vertices. What is really needed is a set of randomly generated polygons of various shapes and structures. While there are several algorithms for generating pseudo-random integers [Leh92], papers, and research outlets addressing the issue of generating random polygons are very rare. Only a few papers on generating simple polygons has been reported [ZSSM96, AH98, Soh99]. Since no polynomial time algorithm for generating random polygon is known, researchers either use heuristics [AH98] or generate the restricted classes of polygons [ZSSM96, Soh99]. In this thesis, we consider the problem of generating simple polygons that satisfy certain visibility properties. Specifically, we examine the problem of generating polygons that are star-shaped. In a star-shaped polygon P, there is a connected set k called *kernel* such that any point inside P is visible to all points inside k.

The thesis is organized as follows. In Chapter 2, we present a condensed review of algorithms for generating simple polygons. We also review several results dealing with the computation of kernel and techniques for searching kernels. We next review algorithms for constructing kernels under *stair-case* visibility which has applications in VLSI design.

In Chapter 3, we present the main contribution of the thesis. We introduce the problem of generating simple polygons that admit large size kernel. For this, we investigate the applicability of the Voronoi diagram induced by the candidate polygonal vertices. We present a space decomposition approach for searching and constructing polygons that are guaranteed to admit a kernel. Our technique is based on applying quad-tree decomposition for searching the candidate kernel points. The approach can be used to generate pseudo-random star-shaped polygons.

In Chapter 4, we present an implementation of the proposed algorithm for random generation of visibility aware simple polygons. The prototype program is developed by using the Python programming language. The prototype software supports a user-friendly graphical interface. Users can execute the program by entering data by mouse click and by reading pre-generated vertices from the input file. Chapter 4 also contains several experimental results on the generation of visibility aware simple polygons.

Finally, in chapter 5, we examine several scope of extending and generalizing the algorithms proposed in this research.

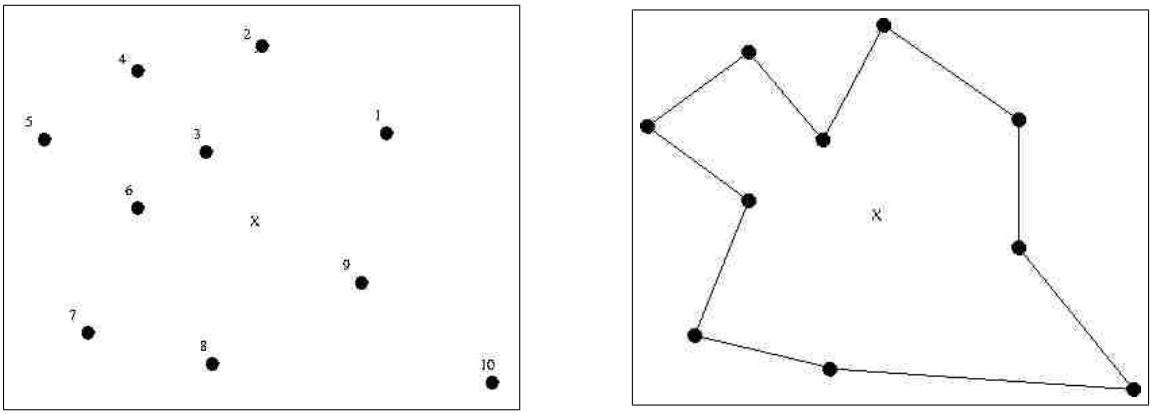
Chapter 2

Visibility Properties of 2D Shapes

In this chapter, we review existing algorithms for constructing and recognizing two aspects of polygonal shapes. In the first part, we review methods for constructing 2D shapes (simple polygons) from a given set of input points. In the second part, we review the visibility properties of polygons under standard and stair-case visibility.

2.1 Preliminaries

Generating two-dimensional shapes and structures of a given set of points is a central problem area in computational geometry. One such problem is connecting points to form a simple polygon. A few algorithms have been reported for constructing a simple polygon from a given set of points [ZSSM96, AH98]. It is remarked that a simple polygon consists of a set of line segments enclosing a connected area that partition the plane into three parts (i) Polygon boundary (ii) unbounded exterior, and (iii) bounded connected interior [O'R87]. A very simple way of generating a simple polygon from a given set of points is to angularly sort and connect. This is illustrated in Figure 2.1. To sort the points angularly, an axis point is picked inside the convex hull of input points. In the left side of the Figure 2.1, a given input points are shown. The axis point is marked as 'x'. The integers adjacent to input points represent the ordering of the angularly sorted (about x) list. In the right part, the polygon implied by the sorted list is displayed. Since sorting is used in this procedure, this approach of constructing polygon needs $O(n \log n)$ time.



(a) (b)

Figure 2.1: Generating Simple Polygon by Angular Sort

2.2 Generation of Random Polygons

An efficient algorithm for generating x-monotone polygon uniformly at random has been proposed in [ZSSM96]. It is remarked that a simple polygon is called monotone with respect to a given direction $\vec{d}$, if the polygon boundary consists of two chains (*top chain* and *bottom chain*) each of which is monotone with respect to $\vec{d}$. Without loss of generality, we assume that direction $\vec{d}$ is x-axis. Alternatively, an x-monotone polygon is such that its intersection with any vertical line is either empty or one line segment as shown in Figure 2.2. The top chain and bottom chain of this polygon are vertices 1, 2, 3, 5, 7, 8 and 1, 4, 6, 8 respectively.

In [ZSSM96], it is argued that if one can count the number of monotone polygons that can be formed (for a given set of vertices $S_k = \{s_1, s_2, ..., s_k\}$ with $k > 2$) then a formal method for random generation is applicable. The counting is done by observing that a given input point can be present either in the top chain or in the bottom chain. The authors arrive at a relationship for counting the number of monotone polygons in terms of a recurrance relation given below.

$$TN(k) = \sum_{j \in V_T(k)} BN(j+1)$$

$$BN(k) = \sum_{j \in V_B(k)} TN(j+1)$$

Where, $TN(k)$ is the number of x-monotone polygons with vertex set S_k that have edge (s_{k-1}, s_k) on their top chain. $BN(k)$ is the number of x-monotone polygons with vertext set S_k that have edge (s_{k-1}, s_k) on their bottom chain. $V_T(k)$ is the set of points that are above visible from the

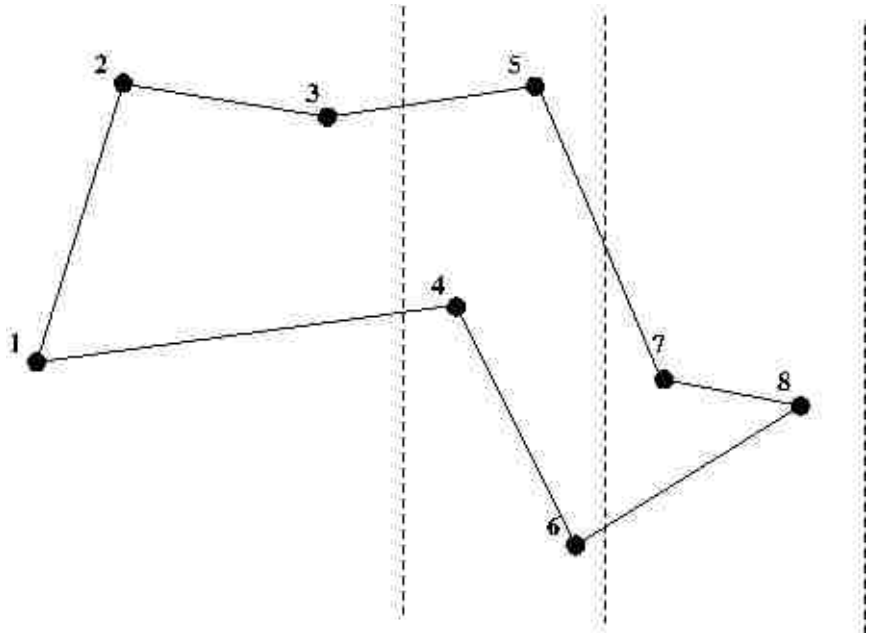


Figure 2.2: Illustrating x-Monotone Polygon

point s_k and $V_B(k)$ is the set of points that are below visible from the point s_k. The details are in [ZSSM96]

The time complexity of the algorithm is bound by $O(n^2)$, where n is the number of input vertices. The recursive relation counts the number of monotone polygons with the given set of vertices S_k by scanning the vertices in the forward direction. A random number between 1 and the total count is picked and the x-monotone polygon corresponding to the selected number is constructed by scanning the vertices in the backward direction. The execution trace for generating an example x-monotone polygon is shown in eight parts of Figure 2.3. The execution can proceed in several construction paths. Each of these paths lead to a particular x-monotone polygon. One such parths results in the x-monotone polygon shown in Figure 2.3h. Other monotone polygons (five more) generated by following admittable paths are shown in Figure 2.4. The authors in [ZSSM96] formally prove that after a pre-processing step needing $O(n^2)$ time, a random x-monotone polygon can be generated in $O(n)$ time. This paper also sketches an approach for generating random convex polygon of time complexity $O(n^3)$.

Algorithms for generating random star-shaped polygons has been given in [AH98, Soh99]. Auer and Held [AH98] gave an algorithm to enumerate all the star-shaped polygons from a given set

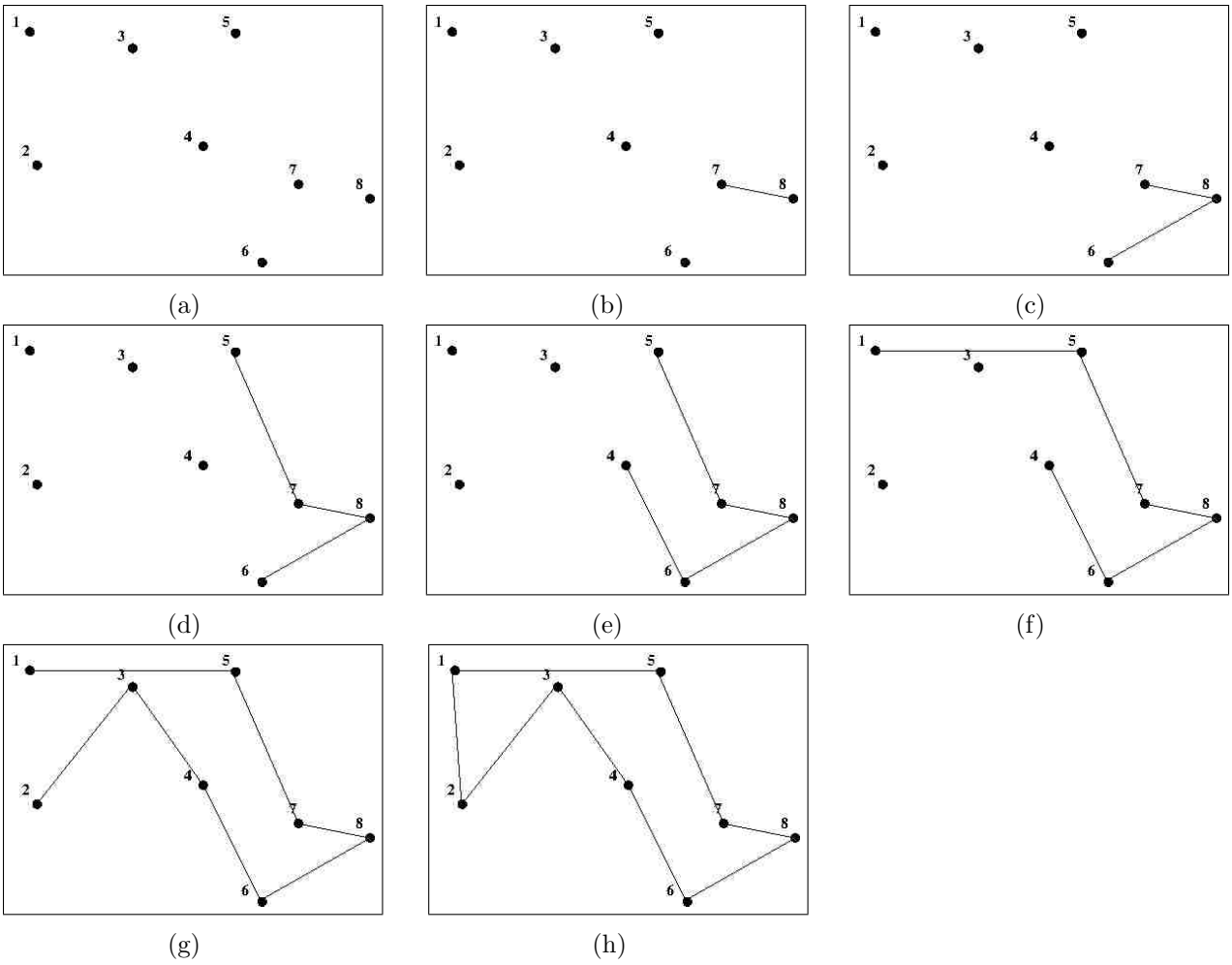


Figure 2.3: An Execution Trace of the x-Monotone Polygon Generation.

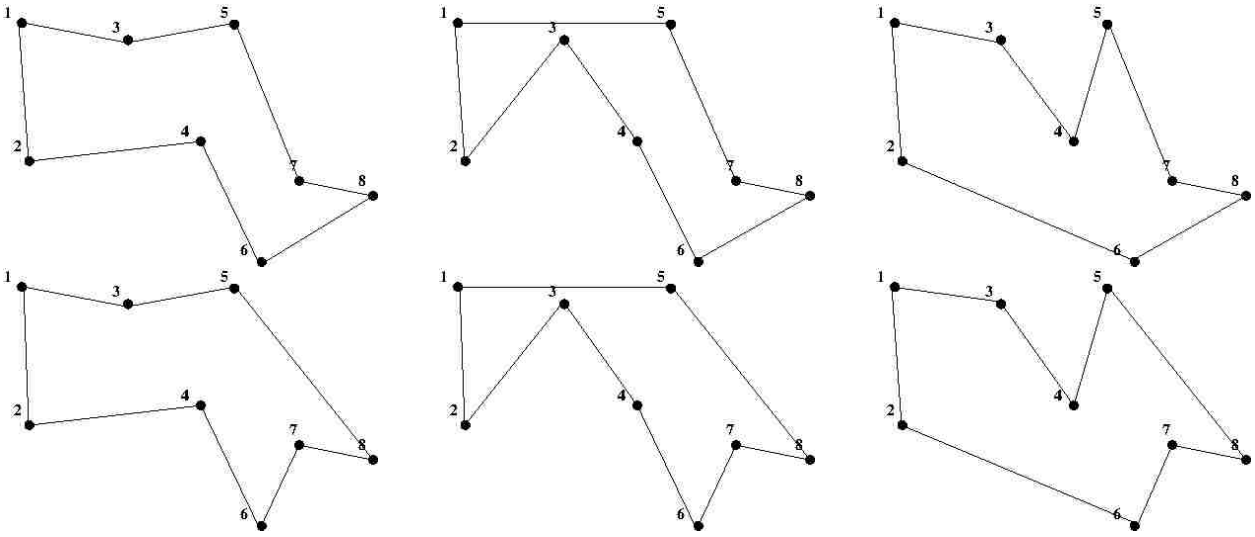


Figure 2.4: All Six x-monotone Polygons Generated from the Vertex Set in Figure 2.3a

of points S. The algorithm takes $O(n^4)$ time and generates polygons uniformly at random. The authors use the fact that a star-shaped polygon is fixed once the kernel of the polygon has been specified. That means each star-shaped polygon has a unique kernel. Once we find the kernel of the polygon, we can sort the vertices angularly around the kernel and construct the polygon in $O(n \log n)$ time. Auer and Held showed that for point sites of size n there can be at most $O(n^4)$ kernels. This can be verified by drawing the lines through each pair of points and counting the number of regions induced by the lines that lie inside the convex hull of the point sites S. This is illutrated in Figure 2.5. The total number of such lines is in the order of $\binom{n}{2} = O(n^2)$. Therefore, the total number of regions (cells) can be at most $O(n^4)$. In other words, there can be at most $O(n^4)$ kernels. One possible arrangement of points that achieve this bound can be obtained by placing $n-2$ points on the parabola $y = x^2$ equally in either side of y-axis and placing the remaining 2 points at $(-\infty, -\infty)$ and $(\infty, -\infty)$ [Soh99].

As we see in Figure 2.5, the lines passing through each pair of points create cells. These cells can be computed in $O(n^4)$ time [O'R87]. More than one cells may belong to the same kernel. Auer and Held gave a depth-first search technique to find the kernels from the cells. This runs in $O(n)$ time where n is the number of cells.

Sohler's [Soh99] algorithm generates a star-shaped polygon uniformly at random that runs in $O(n^2 \log n)$ time and $O(n)$ space. Instead of generating all the star-shaped polygons as in [AH98],

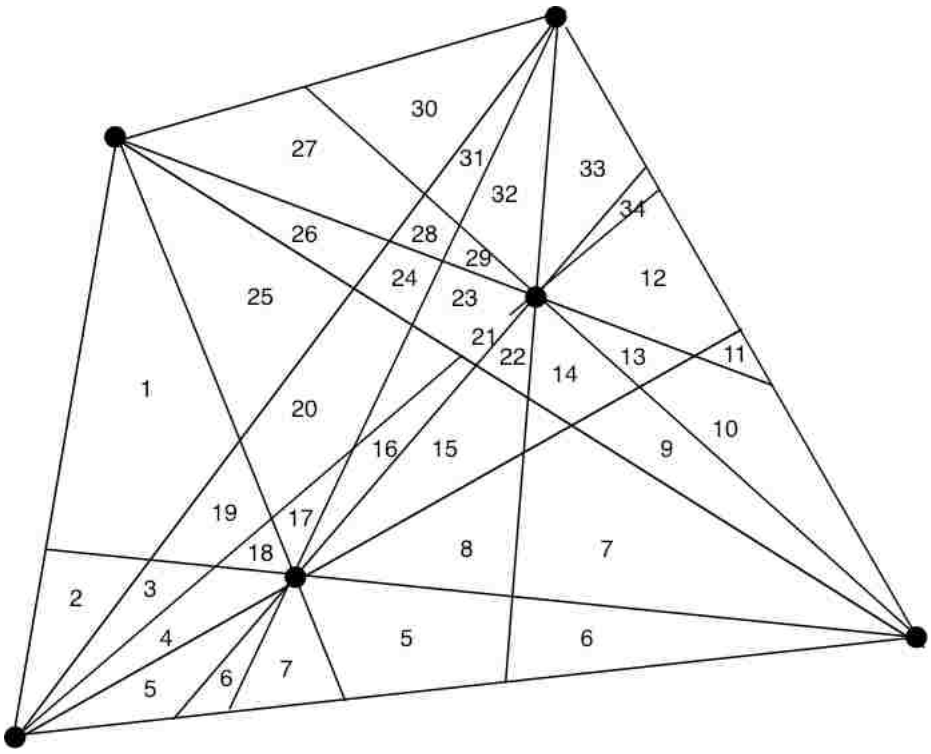


Figure 2.5: Lines Passing Through Each Pair of Points and Induced Cells

the algorithm generates one polygon at a time. A downside of this approach is that it only works for non-degenerate star-shaped polygons. Non-degenerate polygon is a polygon having the kernel of size > 0. The algorithm starts by creating line segments T that partition the convex hull of S as follows.

1. Compute the convex hull of S i.e. compute $CH(S)$.
2. For every pair of points, take two half-lines defined by the line through these points without the segment connecting them. Intersect the half-lines with $CH(S)$ to get two line segments.
3. These line segments and the bounding segments of $CH(S)$ constitute T.

Line segments in T induce the regions that define all the kernels of S. An example of regions defined by six points is given in Figure 2.6.

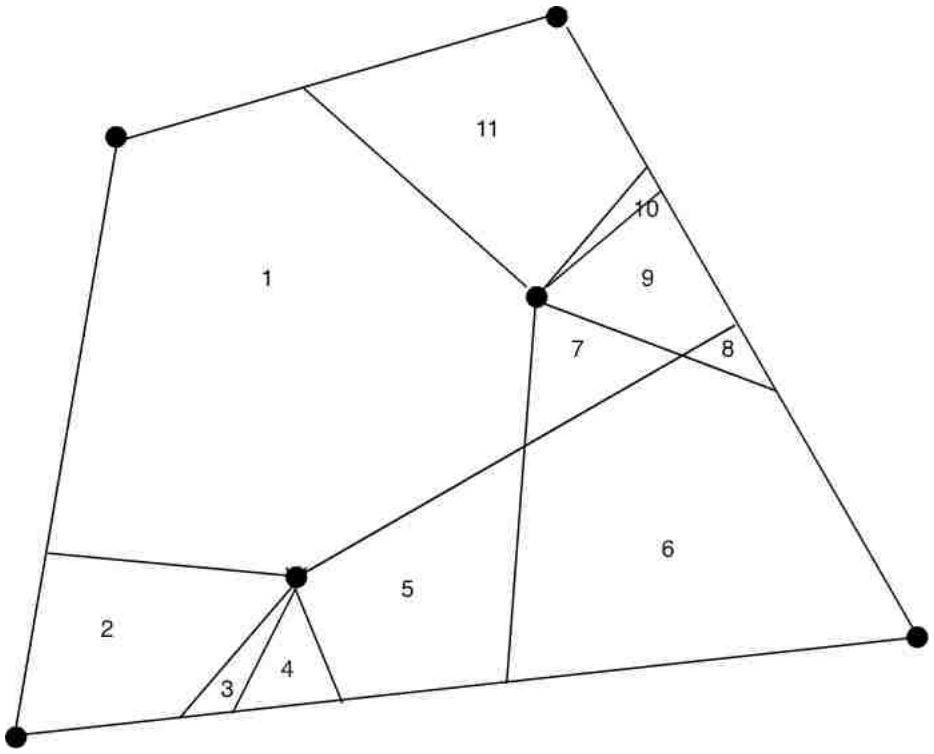


Figure 2.6: An Example of Kernel Subdivision Induced by the Line Segments

Since there are $O(n^2)$ line segments and the intersection of a segment with the convex hull can be computed in logarithmic time, the running time of computing T is $O(n^2 \log n)$. The algorithm then selects a random cell and constructs the polygon defined by the cell in $O(n^2 \log n)$ time.

Furthermore, Sohler showed that the number of star-shaped polygons formed from n input points can be counted in $O(n^5 \log n)$

2.3 Visibility Properties of Polygons

Understanding the visibility properties of polygonal shapes is perhaps the most widely investigated area of computational geometry [O'R87]. While exploring visibility properties of simple polygons, the boundary of the polygon is taken as an opaque wall: a polygon is viewed as a room with walls as a boundary. Under the standard definition of visibility, two points p and q inside the polygon are mutually **visible** if the straight line segment connecting p and q does not intersect with the boundary. Under this definition, many practical algorithms having application in robotics and computer-aided manufacturing have been developed. One such algorithm is the computation of **visibility** polygon induced by a point q inside the polygon. The visibility polygon for an interior point q is the set of points visible from q. Figure 2.7 is an example of a visibility polygon. Visibility

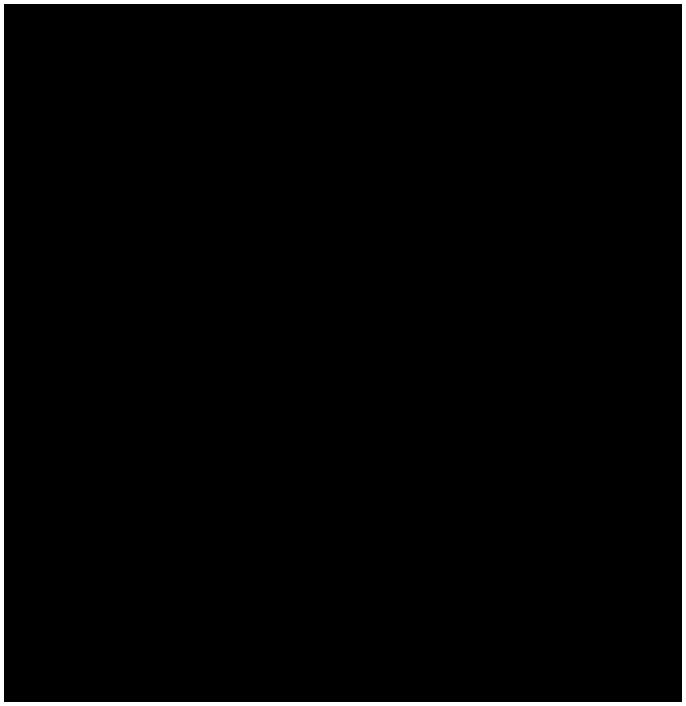


Figure 2.7: Illustrating Visibility Polygon

polygons can be computed in $O(n)$ time for simple polygons and for polygons with holes, algorithms for time complexity $O(n \log n)$ is known [Asa85, Lee83, EA81].

Another useful visibility property of a simple polygon is the concept of Kernel. The **Kernel** of a simple polygon is the set of points from which the polygon is visible. Not every simple polygon admit kernel. Those simple polygons that admit kernel are usually called **star-shaped** polygons. Figure 2.8 illustrates an example of a simple polygon admitting kernel.

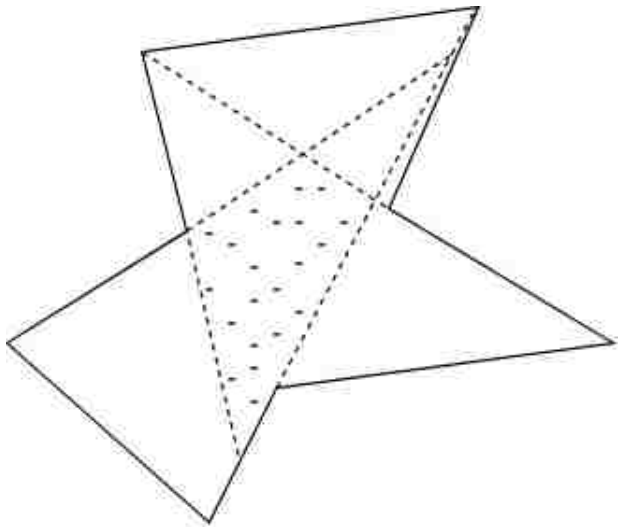


Figure 2.8: Illustrating Kernel of a Simple Polygon

Lee and Preparata [LPP79] gave an algorithm that runs in linear time in the number of the edges for finding the kernel. The authors took advantage of the order of the edges in the polygons to find the linear time algorithm. The algorithm given in [LPP79] scans in order the vertices of the polygon P and construct *kernel chains* $K_0, K_1, K_2, ..., K_{n-1}$. A kernel chain K_i bounds the intersection of half-planes defined by edges e_0 to e_i. The algorithm first finds K_0 and proceeds incrementally to find $K_1, K_2, ..., K_{n-1}$. Finally, using K_{n-1}, it finds the intersection of all the half-planes. The algorithm is optimal for finding the kernel of a star-shaped polygon.

The notion of visibility has been generalized [Vas15, Gew95]. One such generalization is the introduction of **stair-case** visibility [Vas15, Gew95]. Two points p and q inside an orthogonal polygon are called **s-visible** if p and q can be connected by a *stair-case* path. It is remarked that in an orthogonal polygon the boundary edges are parallel to x- and y- axis. A path p' is a stair-case path if its edges are parallel to x- or y- axis and if it is monotone with respect to one of the co-ordinate axes. Intuitively a stair-case path does not back-up. Under s-visibility, the problem of determining and recognizing s-star have been considered [Gew95]. An example of s-star polygon

is shown in figure 2.9.

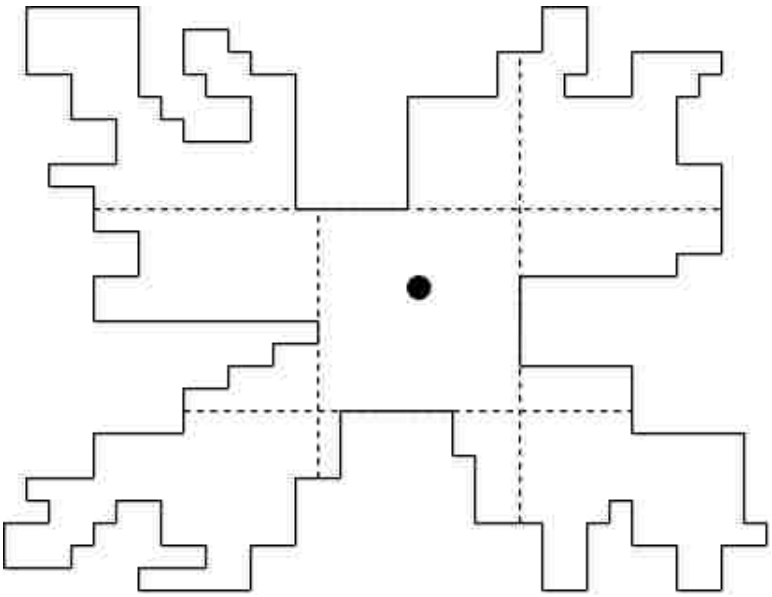


Figure 2.9: An Example of a S-Star Polygon

An algorithm for computing s-kernel in an orthogonal polygon is reported in [Vas15]. The time complexity of this algorithm is $O(n^2)$, where n is the number of vertices in the polygon. It is interesting to note that while polygons with holes never admit kernel under the standard notion of visibility, there could be s-kernels for s-star polygons with holes [Gew95]. Even though a s-kernel could have $\Omega(n^2)$ components, such polygon can be recognized in $O(n)$ time [Gew95]. It is known that the problem of computing visibility polygon under stair-case visibility inside a polygon containing holes has a lower bound of $\Omega(n \log n)$ [Gew95]. This result is extended to orthogonal polygons with holes in [Gew95]. This result is established by reducing sorting problem to the problem of computing s-visibility polygon [Gew95].

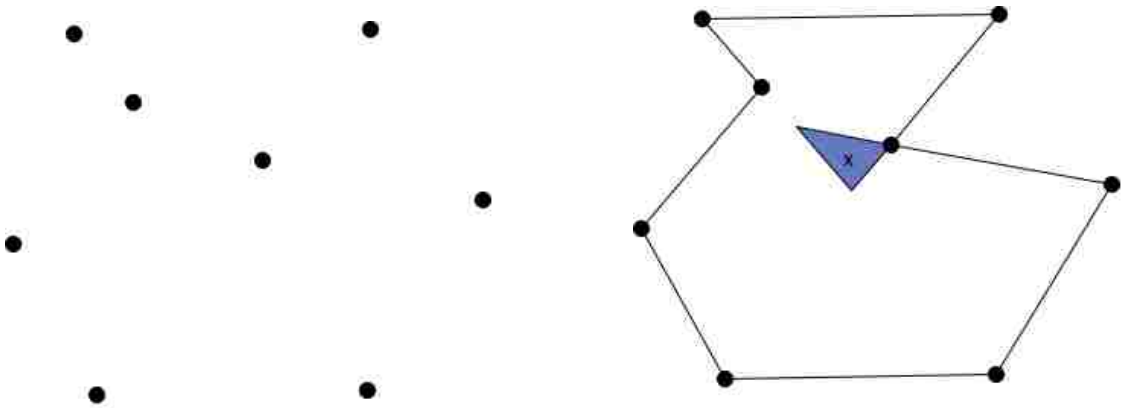
Chapter 3

Star Shaped Polygons with Large Kernels

In this chapter, we study the development of an efficient algorithm for generating polygons (for given point sites) that tend to have a large area kernel.

3.1 Preliminaries

Given a set S of n points $v_0, v_1, ..., v_{n-1}$, our objective is to construct a star-shaped polygon whose vertices are exactly the points in S. A very simple method for generating a star-shaped polygon is to use the technique described in Chapter 2. Specifically, a reference point r_0 is picked inside the convex hull of points in S. All points are angularly sorted to obtain an order list L'. The vertices are connected in the order implied by L' to obtain polygon Q. The star-shaped polygon so constructed admits a region called *kernel* k_r such that any point inside Q is visible from all points in k_r. It is straightforward to observe that the shape of the kernel k_r is convex. An example of the generated polygon, picked reference point, and the admitted kernel is shown in Figure 3.1. One interesting question arises for picking the reference point. If we pick another reference point, the admitted kernel will be much larger as shown in Figure 3.2. These observations motivate us to look for an algorithm that generates a star-shaped polygon that maximizes the area of the implied kernel. The problem can be formally stated as follows.



(a) Given point sites S

(b) Generated polygon, picked reference point (shown by 'x') and kernel

Figure 3.1: Illustrating Kernel

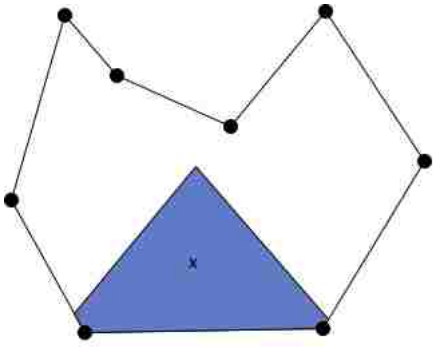


Figure 3.2: Another Instance of Kernel

Max Kernel Polygon (MK) Problem

Given: A set of points $S = \{v_0, v_1, ..., v_{n-1}\}$

Question: Construct a star-shaped polygon with vertices in S that maximizes the implied kernel.

It is not clear how to pick the reference point. We need to perhaps exploit structural, proximity, and distribution properties of points in S. A proximity structure widely used in computational geometry and robotics [O'R87] is the Voronoi diagram induced by the point sites. The Voronoi diagram of n point sites captures the proximity of point distribution. In fact, the region around Voronoi vertices tends to be void of point sites. The Voronoi diagram induced by point sites (filled dots) is shown in Figure 3.3. In order to get some clue about the relationship between Voronoi

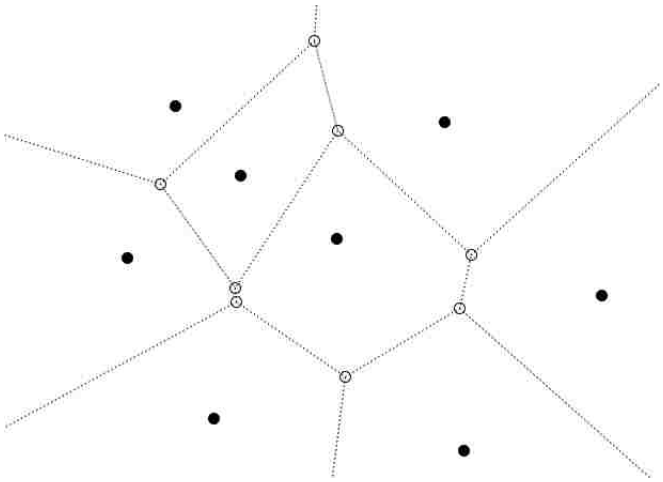


Figure 3.3: Illustrating Voronoi Diagram

vertices and family of kernels, we generated all possible kernels implied by the point sites in Figure 3.1a. The superimposition of Voronoi vertices and the family of kernels shows that some kernels may contain more than one Voronoi vertex and some may not contain any of them. An illustration of the superimposition is shown in Figure 3.4.

This example implies that the Voronoi vertices do not necessarily capture the position of reference point for maximizing the kernel area.

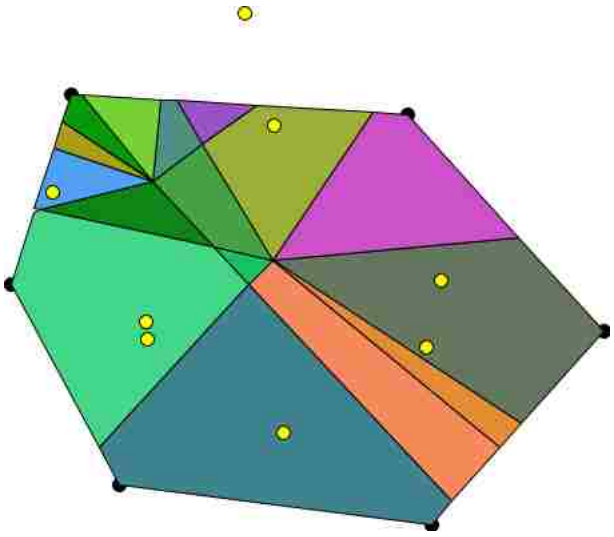


Figure 3.4: Superimposition of Voronoi Vertices and Family of Kernels

3.2 Quadtree Based Searching

We propose to use the quadtree technique introduced by Raphel Finkel and J.L. Bently [FB74]. In this technique, a two-dimensional bounded rectangular region R is orthogonally partitioned into four ortho-rectangular subregions which we refer as *quadruplets*. The process of partitioning into quadruplets proceeds recursively to arrive at a small *region of interest*. The implied result of this recursive partitioning is the formation of a quadtree where each internal node has four children.

In our kernel search context, the point of interest occurs when a small region is found that results in a large kernel. We illustrate this quadtree based kernel search with a running example. Figure 3.5a shows randomly generated seven-point sites with the smallest enclosing axis parallel rectangle (drawn by the dashed line). The edges of the resulting polygon implied by the angular sorting of the point sites are drawn by the solid lines. The corresponding kernel by selecting the axis point at the center of the bounding rectangle is shown shaded. The algorithm next proceeds to search for the other kernels in each of the quadruplets inside the current region R. The kernel corresponding to the north-east sub-region (child) is shown in figure 3.5b. This process is continued recursively until the region of interest is reached. A sequence of resulting kernels found in this way are shown in Figure 3.5b - 3.5i. Figure 3.6 shows the quad tree curresponding to Figure 3.5. The recursive partitioning stops whenever conditions A1 and A2 are satisfied. Based on the working of the above running example, the kernel searching algorithm can be described as follows.

The smallest ortho-rectangle R_0 enclosing all input vertices is determined. The first candidiate kernel point is the center $c(R_0)$ of R_0. The star-shaped polygon and corresponding kernel $k(R_0)$ are computed. Next, R_0 is partitioned into four parts (NE, NW, WS, SE) of equal areas. Star-shaped polygon construction, and the corresponding kernel construction is repeated recursively on each of the four NE-, NW-, WS-, and SE-parts. The recursion stops when the following condtions are satisfied.

Sub-condition A_1: The height of the quadtree is no more than the pre-determined integer value.
Sub-condition A_2: Area of running max kernel must be at least twice the area of the next candidate node of the quadtree.A formal sketch of the kernel searching algorithm is listed as Algoritm 1.

Algorithm 1 Kernel Searching Algorithm

1: **Input**: Point sites $P = p_0, p_1, p_2, ..., p_{n-1}$
2: **Output**: Star-shaped polygon S with large kernel
3: Compute enclosing ortho-rectangle R_0
4: Find kernel k_0 corresponding to R_0
5: $Q \leftarrow$ Empty Queue
6: Push R_0 into Q
7: $GlobalMaxKer \leftarrow k_0$
8: **while** Q is not empty **do**
9: $R_i \leftarrow$ Pop node from Q
10: Partition R_i into four children $R_{i_1}, R_{i_2}, R_{i_3},$ and $, R_{i_4}$
11: Push those children into Q that satisfy condition A1 and A2
12: Compute kernels $k_{i_1}, k_{i_2}, k_{i_3},$ and k_{i_4} corresponding to four children
13: $MaxKer \leftarrow$ Kernel with maximum area among $k_{i_1}, k_{i_2}, k_{i_3},$ and k_{i_4}
14: **if** area$(MaxKer) >$ area$(GlobalMaxKer)$ **then**
15: $GlobalMaxKer = MaxKer$
16: $S \leftarrow$ Ordered vertices corresponding to $GlobalMaxKer$

Complexity Analysis

The complexity analysis of Algorithm 1 can be done in a straightforward manner. In the **while** loop, the dominant step is finding the kernel (step 12). One execution of step 12 can be done in $O(n)$ time by using the algorithm given in [LPP79]. The **while** loop executes t times, where t is the total count of nodes in the quadtree constructed during the recursive partitioning in step 10. Hence the execution time of algorithm 1 is $O(tn)$.

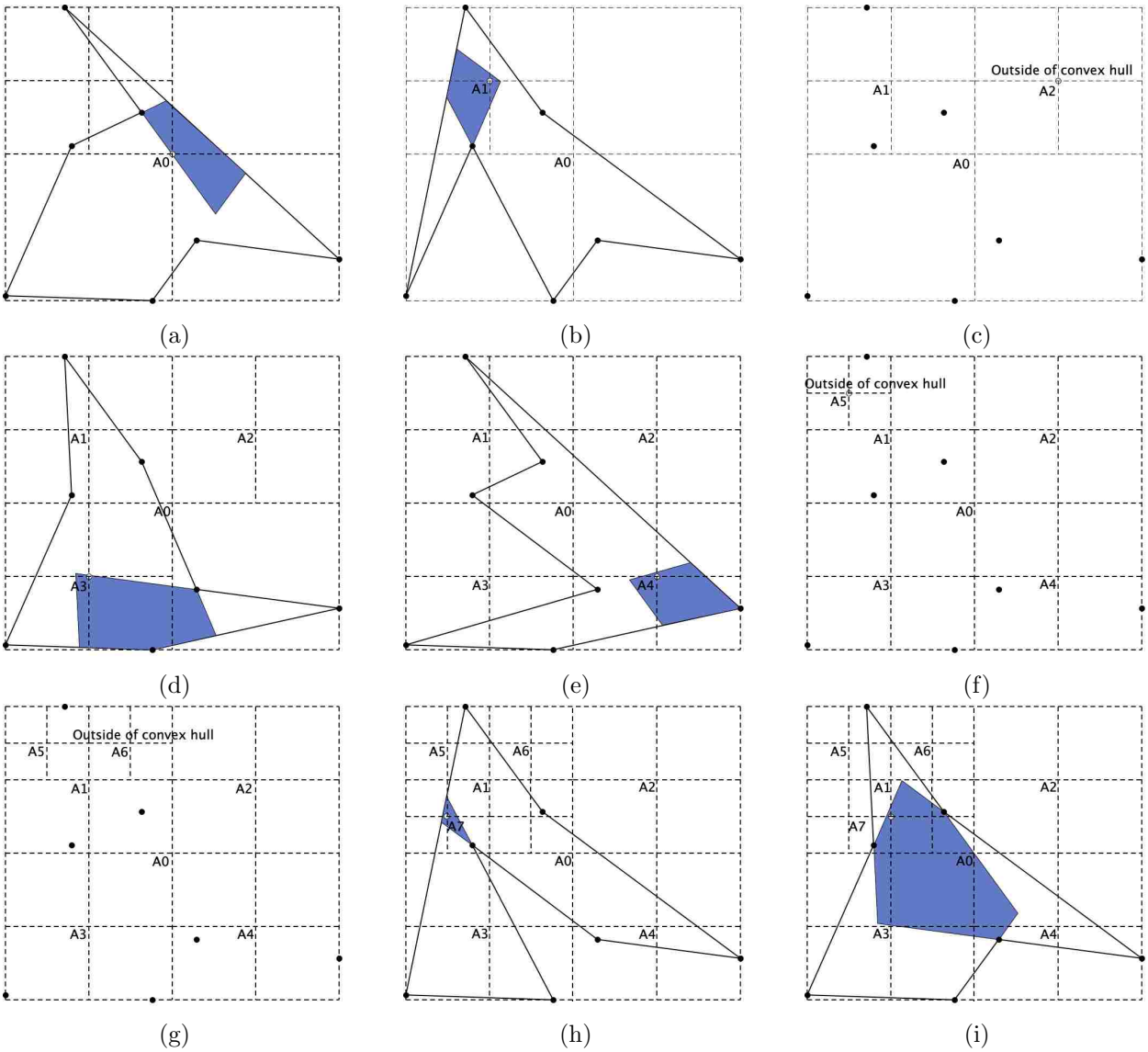
3.3 Sensitivity Analysis

Let $P(l)$ denote the polygon that maximizes the kernel as obtained by Algorithm 1. Consider the polygon $P^{'}(l)$ obtained by deleting a reflex vertex from $P(l)$. It can be observed that the kernel of $P^{'}(l)$ is at least as large as kernel of $P(l)$. This is stated in the following lemma.

Lemma 3.1: The polygon $P^{'}(l)$ obtained by deleting a reflex vertex from $P(l)$ is such that its kernel is no less than the kernel of $P(l)$.

Proof: Consider the boundary and the kernel of the polygon $P'(l)$ as shown in Figure 3.7. With respect to any point q inside the kernel, the vertices of $P'(l)$ are angularly sorted in the same order. By the definition of star-shaped polygon, the deleted vertex v' is in one of the angular edges formed by q, v_i, and v_{i+1}. We can distinguish the followng two cases.
Case 1: Vertex v' does not alter the existing reflex vertex or a new reflex vertex is not formed (as shown in Figure 3.7a). In this situation the kernel of $P'(l)$ and $P(l)$ are identical.
Case 2: Vertex v' alters an existing reflex vertex or creates a new reflex vertex (as shown in Figure 3.7b). In this situation, the reflex chord emanating from the new reflex vertex may chop off part of the kernel. This implies that the kernel of $P(l)$ is a subset of the kernel of $P'(l)$.

We illustrate this process by an example. Figure 3.8 is a randomly generated star shaped polygon (with indicated kernel) of 30 vertices. If reflex vertex V_{10} is deleted, a polygon of 29 vertices with much larger kernel is found which is shown in Figure 3.9.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 3.5: Illustrating the QuadTree Based Searching

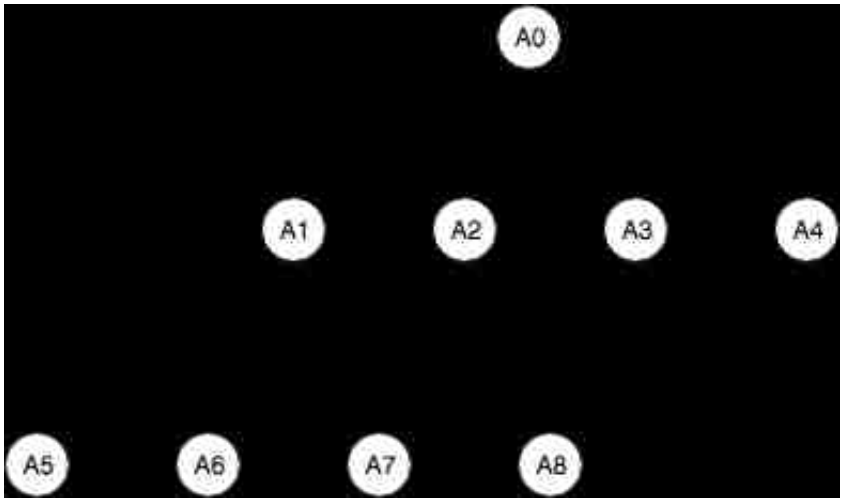


Figure 3.6: A QuadTree Corresponding to Figure 3.5

(a) (b)

Figure 3.7: Proof of Lemma 3.1

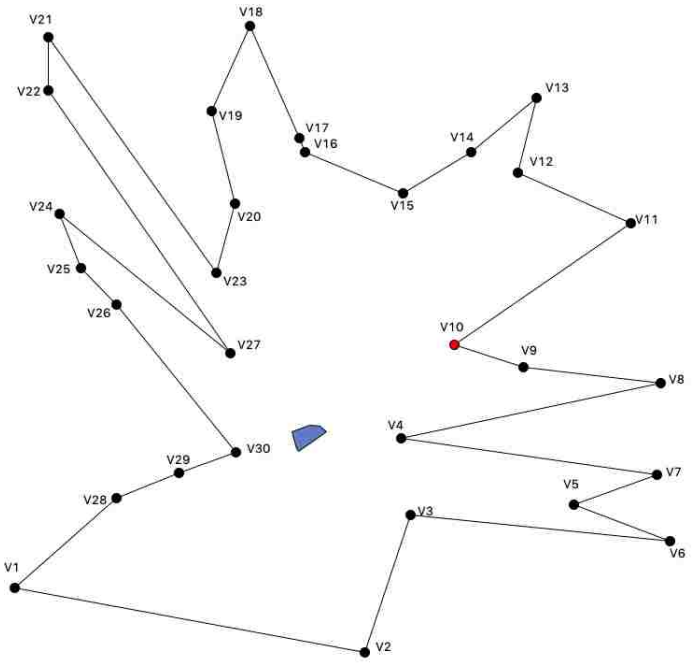


Figure 3.8: A Star Shaped Polygon Showing the Most Sensitive Vertex

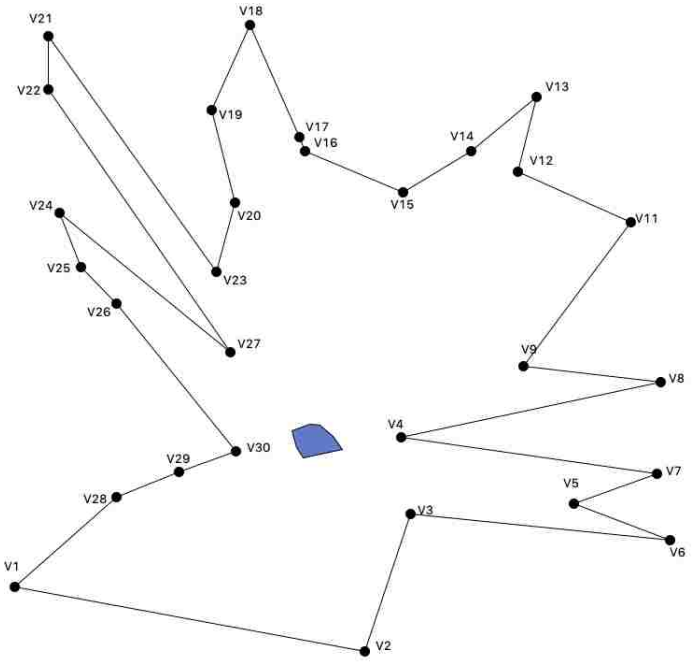


Figure 3.9: The Polygon After the Removal of the Most Sensitive Vertex

Chapter 4

Implementation and Experimental Results

In this chapter we present the implementation of some of the algorithms from chapter 2 and 3. This includes (i) algorithm for generating random x-monotone polygons, (ii) algorithm for calculating the kernel of a polygon, (iii) algorithm for generating a polygon with largest kernel, and (iv) algorithm to find the most sensitive vertex.

4.1 General Principles

We implemented the algorithms in Python programming language. Python is an interpreted, high level, object-oriented programming language with dynamic type checking. The version of the Python we used is 2.7. There are a couple of reasons for choosing Python over other programming languages. First, Python is good at building prototypes in a short period of time. Second, it offers a plethora of libraries for GUI and scientific computation. Third, it is a platform independent language which means the same code runs everywhere without any modification to the source code.

For generating a user interface, we used *wxPython* library. wxPython is a Python wrapper of *wxWidget* which is a C++ library for building GUI components. It has built-in support for most of the GUI components like Buttons, Textboxes, Labels, Menus, Drawing canvas, etc, and has an easy to understand event handling mechanism. wxPython is a cross-platform and open source library with a large and active developer community. In addition, it gives a native look and feel to the application across all the major platforms.

For computing the Voronoi vertices and intersection of halfplanes needed for two of our algo-

rithms, we used the QHull (http://www.qhull.org) library [BDH96]. QHull provides a fast, efficient and accurate implementation of many computational geometric algorithms including convex hull, Voronoi diagram, Delaunay triangulation, and halfspace intersection.

4.1.1 User Interface

As mentioned in section 4.1, we implemented the GUI using the wxPython library. The GUI allows us to interactively add point sites S and apply the algorithms on S. There is also an option for generating S randomly for some algorithms. Figure 4.1 shows the layout sketch of the main application window. It consists of seven container parts: menus, drawing controls, status bar, program handles, coordinates, results, and drawing canvas.

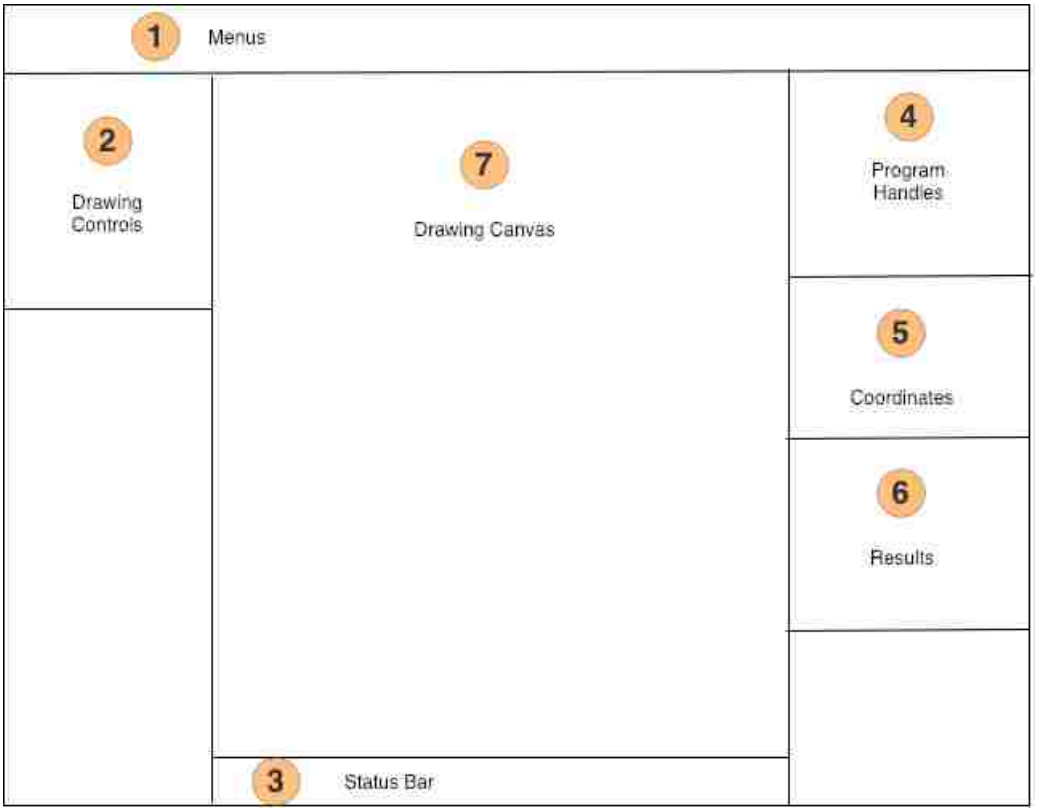


Figure 4.1: Structure of the Main Window

1. **Menus** Dropdown menus ("File", "Edit", "Polygon", and "Algorithms") are on the top of the window. The **"File"** menu has the following operations.

(a) **Open** loads the drawing items (points, lines, and texts) onto the drawing canvas.

(b) **Save** stores the drawing items on a text file.

(c) **Export** saves the canvas in other file formats. Currently it has options to export in PNG and xFig file format.

(d) **Quit** exits the application and returns the control to the calling program.

The **"Edit"** menu supports the following operations.

(a) **Clear** clears the drawing canvas. It clears all the temporary variables and resets the program.

Next, the **"Polygon"** menu offers the following options.

(a) **Random x-monotone** generates the x-monotone polygon from the point site S randomly. It draws the generated polygon one at a time.

(b) **Star shaped polygon** generates the star-shaped polygon from the given point site S and a reference point R. It also computes the kernel K of the generated polygon.

The **"Algorithms"** menu supports the following operations.

(a) **Voronoi Diagram** computes the Voronoi diagram from the point sites S and draws the diagram on the canvas.

(b) **Bucketing** generates the star-shaped polygon from the point sites S that has the largest kernel area using the bucketing algorithm presented in section 3.2.

(c) **Convex hull** computes and draws the convex hull of the given point sites S using the Graham scan algorithm.

2. **Drawing controls** This panel consists of four items. Three of them are the radio buttons: "Draw Point", "Draw Line", and "Draw Text". When the "Draw Point" is selected, only points can be drawn on the canvas. Similarly, when "Draw Line" is selected, only line segments can be drawn and so on. The last item is a checkbox. When the checkbox is ticked, we can only edit the existing items on the canvas but cannot draw the item. We can modify the properties, move the item around the canvas and remove it.

3. **Status Bar** The left part of the status bar displays the status of the operations we perform in various colors depending upon whether the operation succeeds or fails. For example, if the

operation is successful, it displays the message in a green color. Likewise, if the operation fails, it displays the message in a red color. The right part of the status bar prints the coordinates of the current mouse position.

4. **Program Handles** The contents on this panel are dynamic and specific to the algorithm being executed. It displays options available to the particular algorithm under execution.

5. **Coordinates** It presents the coordinates of the point sites drawn on the canvas. The coordinates can also be modified by double-clicking the respective coordinate.

6. **Results** After the computation is successful, the result of the computation are presented in this panel. It shows the necessary information needed for the computed algorithm.

7. **Drawing Canvas** The last and the most important part is the drawing canvas. The canvas allows us to interactively add drawing components, move them around, modify their properties, and remove them. It currently supports point, line, and text. The point is a filled circle with a specified radius. These items should be confined to the boundary of the panel. The implementation does not support the items that are out of the boundary.

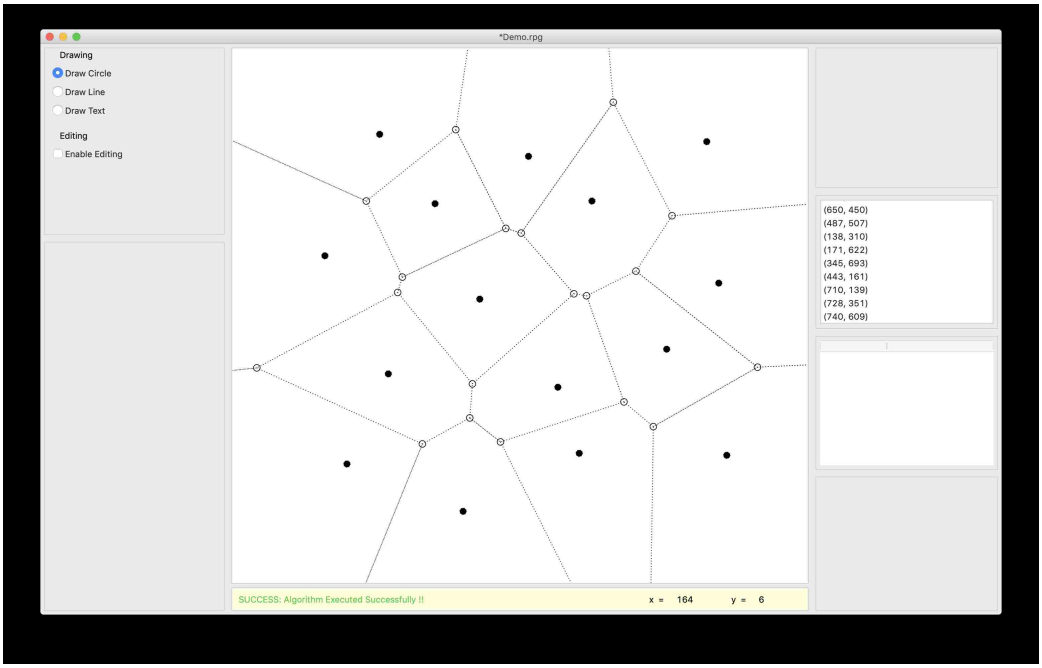


Figure 4.2: A Screen Shot of the Main User Interface

4.1.2 File Format

Our application allows us to export points, lines, and texts to a text file and to import them from the text file. Even though we save the file in *.rpg* extension, it is basically a standard text file. Each line of the file represents an item. The content of the line is separated by spaces. The line starts with a number that denotes the type of the item.

To save a point, we save its x coordinate, y coordinate, radius, and the color. The color is represented by RGB value. The line representing point always starts with 1 followed by x-coordinate, y-coordinate, radius, red value, a green value, and blue value respectively as shown below.

1 x y Radius Red Green Blue

Similarly, to save a line segment, we save the xy coordinates of both the endpoints, a width of the line, color, and the style. The style offers two choices: solid and dotted. The line starts with 2 followed by x and y coordinates of one end, x and y coordinates of the other end, width, red value, green value, a blue value, and style respectively.

2 x1 y1 x2 y2 width Red Green Blue style

Finally, to save the text, we save the x and y coordinates of the starting letter and the actual raw text. The actual text is stored by replacing space by \w. The line starts with a number 3 followed by x and y coordinates, and raw text respectively as follows.

3 x y text

The coordinates are integer numbers. Our implementation does not support floating point coordinates. Similarly, the color values must be an integer value between 0 to 255. The style is either 100 or 103; 100 denotes the solid line and 103 denotes the dotted line. A sample output file is given below.

3 312 485 A1

3 294 277 B2

1 219 202 5 140 154 82

1 452 153 5 0 0 0

1 497 284 5 0 0 0

1 379 410 7 127 0 127

1 377 271 5 0 0 0

1 268 348 5 0 0 0

1 307 558 5 0 0 0

1 451 528 5 0 0 0

2 175 357 307 144 1 0 0 0 100

2 322 314 569 394 5 251 2 254 103

In addition to the *.rpg* format, we can export the items in two additional file formats. The first is the PNG format which is a widely supported image format, and the second is the format that is required by a graphics editor called xFig.

4.2 Data Structures

In this section, we discuss the data structure that we created to model the drawing components like point, line segment, polygon, etc.

4.2.1 Drawable Point

Drawable Point is used to model a point/vertex that can be drawn on the canvas. It is a derived class inherited from the **Point** data type. It inherits all the mathematical computations and coordinates from the **Point** data type and adds new methods so that the point can be shown on the drawing panel. The class interface diagram of **DrawablePoint** is shown in Figure 4.3.

4.2.2 Drawable Segment

Drawable Segment models the line segment that can be drawn and shown on the drawing panel. The **DrawableSegment** extends from the base class **Segment**. The base class specifies the coordinates of the endpoints as instances of the class **Point**. It also provides methods that are needed to perform the computations corresponding to the mathematical properties of line segments. The derived class **DrawableSegment** derives all these properties and methods from **Segment** class and provides additional functionality in order to give it a visual appearance. Figure 4.4 shows the class diagram of **DrawableSegment**.

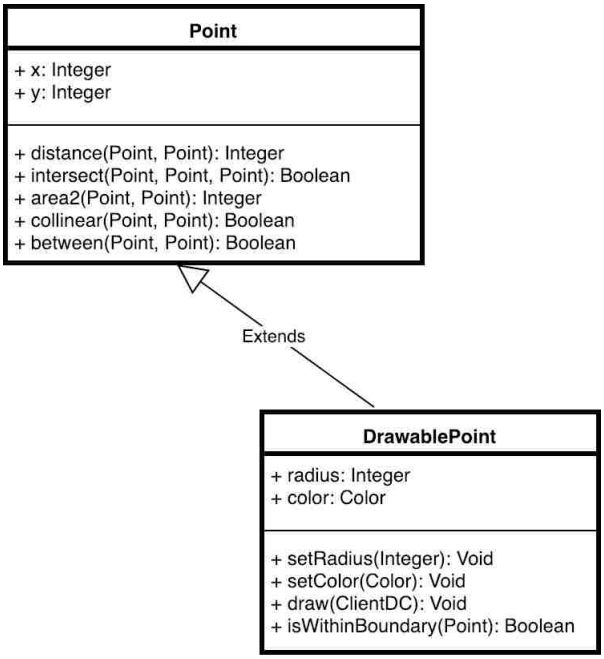


Figure 4.3: A Class Interface Diagram of DrawablePoint

4.2.3 Polygon

A polygon is represented by an array of points ordered in an anticlockwise fashion. The **Polygon** data structure is used to model the polygon. At present, it does not offer the rich set of operations as in **Point** and **Segment** but it can be easily extended to include additional properties. Figure 4.5 shows the class interface diagram of **Polygon**.

4.2.4 QuadTree

A quadtree is used in the implementation of the algorithm described in Section 3.2. A node in a quadtree has at most four children. We divide the region of context into four parts recursively and each part is represented by a child node. This is described in detail in Section 3.2. Figure 4.6 shows the class interface diagram of a node. The first four properties are self-explanatory. They describe the rectangular region of given height and width with the coordinates of the center. Level holds the level of the node in the quadtree, the kernel is the kernel area of the star-shaped polygon taking the reference point at (x, y), and children holds the four child nodes.

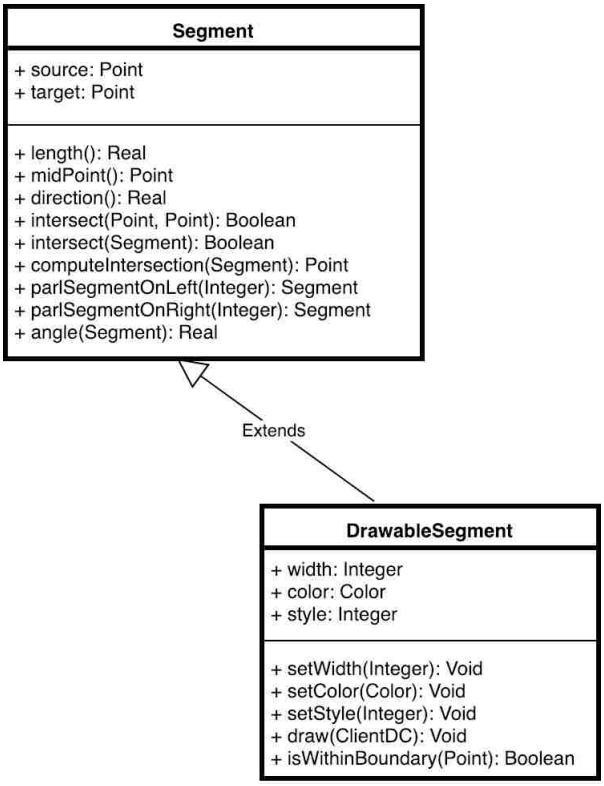


Figure 4.4: A Class Interface Diagram of Drawable Segment

4.3 Implementation of Random Generation of x-Monotone Polygon

We implemented the random generation of an x-monotone polygon discussed in Section 2.2. The x-monotone polygon is a combination of top and bottom monotone chains. We store top and bottom chains on separate linked lists and combine them to make a polygon afterward.

The first phase of the implementation is the *counting*. We count how many x-monotone polygons can be generated with a given set of vertices. We start by computing the *above-visible* (V_T) and *below-visible* (V_B) sets. We use a brute-force approach to find these sets even though a much more efficient approach exists to do the same as described in [ZSSM96]. Once we have V_T and V_B, we can calculate T_N and B_N by calling the method **calc_TN_and_BN** as given in Listing 4.1.

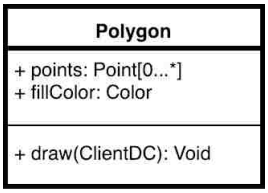


Figure 4.5: A Class Interface Diagram of Polygon

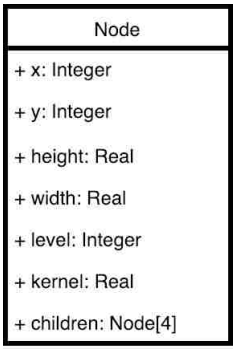


Figure 4.6: A Class Interface Diagram of Node

```
def calc_TN_and_BN(n)
        TN[1] = 1
        BN[1] = 1
        for i in range(2, n + 1):
            TN[i] = 0
            BN[i] = 0
            for j in VB[i]:
                TN[i] += BN[j + 1]
            for j in VT[i]:
                BN[i] += TN[j + 1]
```

Listing 4.1: A Python Program to Compute the Set TN and BN

The total number of polygons generated is the sum of $TN(n)$ and $BN(n)$, where n is the index of the rightmost vertex. The second phase of the implementation is the *generation* of the polygons. We generate the top chain and the bottom chain separately and combine them to generate the final monotone polygon. The algorithm generates one polygon at a time based on the random number picked between 1 and n. The Python implementation is given in the Listing 4.2. The total number of polygons generated is the sum of $TN(n)$ and $BN(n)$, where n is the index of the rightmost vertex.

```
def generate():
    # total number of polygons that can be generated
    N = BN[n] + TN[n]
    # pick random number between 1 and N
    x = random.randint(1, N)
    # the right most point should be both in
    # top chain and bottom chain
    top_chain.append(points[n])
    bottom_chain.append(points[n])
    # generate top and bottom chain
    if x <= TN[n]:
        top_chain.append(points[n - 1])
        generate_top(n, x)
    else:
        x = x - TN[n]
        bottom_chain.append(points[n - 1])
        generate_bottom(n, x)
    # insert the left most point in both the chain
    # if it is not already there.
    if top_chain[-1] != points[0]:
        top_chain.append(points[0])
    if bottom_chain[-1] != points[0]:
        bottom_chain.append(points[0])
```

Listing 4.2: A Python Program to Generate the x-Monotone Polygon

Figure 4.7 shows the x-monotone polygon with 50 vertices generated using the implementation discussed above.

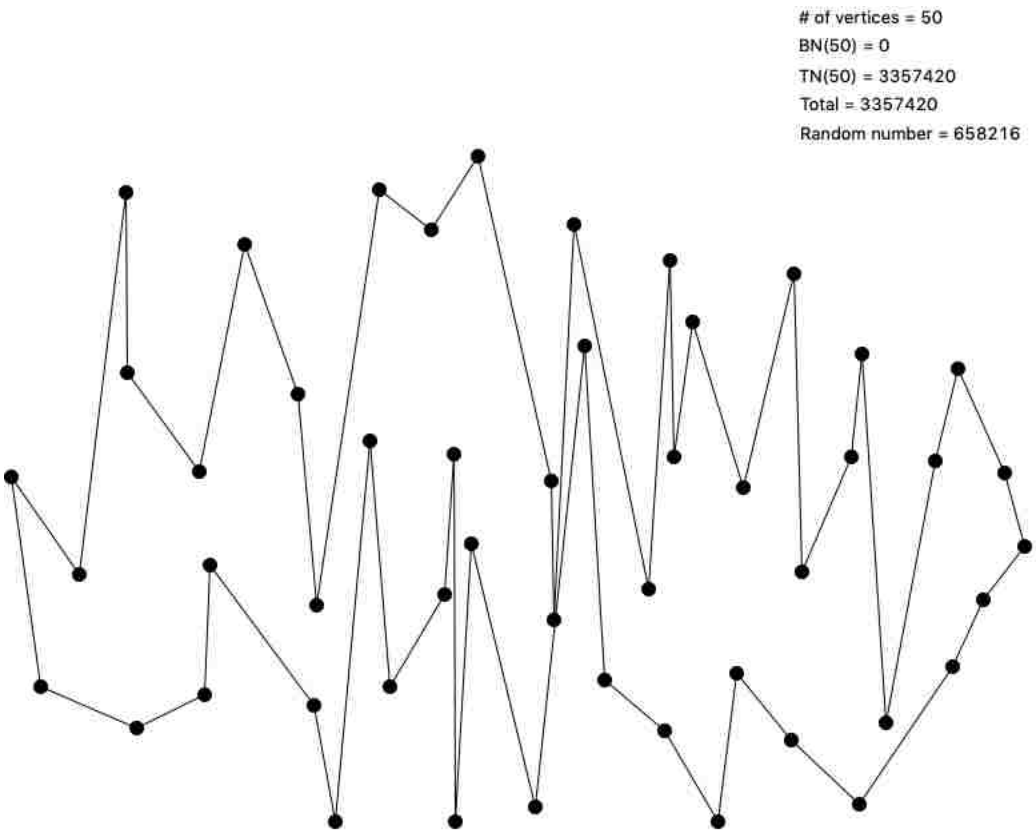


Figure 4.7: A x-Monotone Polygon with 50 Vertices

4.4 Finding the Kernel of a Polygon

Two of the algorithms discussed in Chapter 3 need the area of the kernel. Before finding the area, we need to first find the kernel itself. Using the fact that a kernel is a convex polygon, we can then use the Shoelace formula [Wik19] given below to find the area.

$$A = \frac{1}{2}\left|\sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n\right|$$

Where, $(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})$ are the coordinates of the vertices ordered in an anti-clockwise direction.

A kernel of a star-shaped polygon can be calculated in $O(n)$ time using the approach proposed by Lee et al. [LPP79]. We followed a different approach where the kernel is calculated by computing the intersection of half-planes. We used the method **HalfspaceIntersection** provided by the Qhull library. This method requires two parameters: one is the equations of all the half-planes and another

is a reference point R. Given two vertices $V_1(x_1, y_1)$ and $V_2(x_2, y_2)$, the equation of the half-plane passing through these points is,

$$Ax_1 + By_1 + C = 0 \text{ Where, } A = -(y_2 - y_1) \text{ and } B = x_2 - x_1$$

In a computer program, we represent the half plane by the list [A, B, C]. Listing 4.3 shows the Python program that computes the kernel.

```
def compute_kernel(points, ref_point):
    # angularly sort the point wrt the reference point
    spoints = angular_sort(points, ref_point)
    # store the coefficients A, B, C for all the halfplanes
    coeffs = []
    for i in range(len(spoints)):
        p1 = spoints[i]
        p2 = spoints[(i + 1) % len(spoints)]
        # calculate A, B and C
        a = -(p2.y - p1.y)
        b = p2.x - p1.x
        c = -(a * p1.x + b * p1.y)
        coeffs.append([a, b, c])
    halfspaces = np.array(coeffs)
    # compute the intersection of halfplanes
    hs = HalfspaceIntersection(halfspaces, np.array([float(ref_point.x)
        , float(ref_point.y)]))
    kernel_polygon = hs.intersections.tolist()
    # angulary sort the kernel vertices again
    kernel_polygon = angular_sort(kernel_polygon, ref_point)
```

Listing 4.3: A Python Program to Compute the Kernel of a Star-Shaped Polygon

The vertices need to be angularly sorted with respect to the reference point R before finding the coefficients A, B, and C. The method **HalfspaceIntersection** returns the vertices of the kernel.

These vertices should be sorted angularly with respect to R in order to get the correct polygon. The Python implementation of the Shoelace formula is given in Listing 4.4.

```
def poly_area(vertices):
    psum = 0
    nsum = 0
    for i in range(len(vertices)):
        sindex = (i + 1) % len(vertices)
        prod = vertices[i].x * vertices[sindex].y
        psum += prod
    for i in range(len(vertices)):
        sindex = (i + 1) % len(vertices)
        prod = vertices[sindex].x * vertices[i].y
        nsum += prod
    return int(abs(1/2*(psum - nsum)))
```

Listing 4.4: A Python Program to Find the Area of a Polygon

Figure 4.8 shows an example of output of the implementation. The boundary of the halfplanes are shown by the dotted lines and the reference point is shown by a red circle.

4.5 Implementation of QuadTree Based Searching

We implemented the quadtree-based searching described in Section 3.2. In this approach, we draw the smallest iso-rectangle enclosing the point sites and find the center point M of the rectangular region. We compute the midpoint $M(x_{mid}, y_{mid})$ of the rectangular region as follows.

$$x_{mid} = (x_{low} + x_{high})/2$$
$$y_{mid} = (y_{low} + y_{high})/2$$

Where, x_{low} is the x-coordinate of the left-most point, x_{high} is the x-coordinate of the right-most point, y_{low} is the y-coordinate of the top-most point and y_{high} is the y-coordinate of the bottom-most point. We take the midpoint M as a reference point and compute the kernel of the resulting

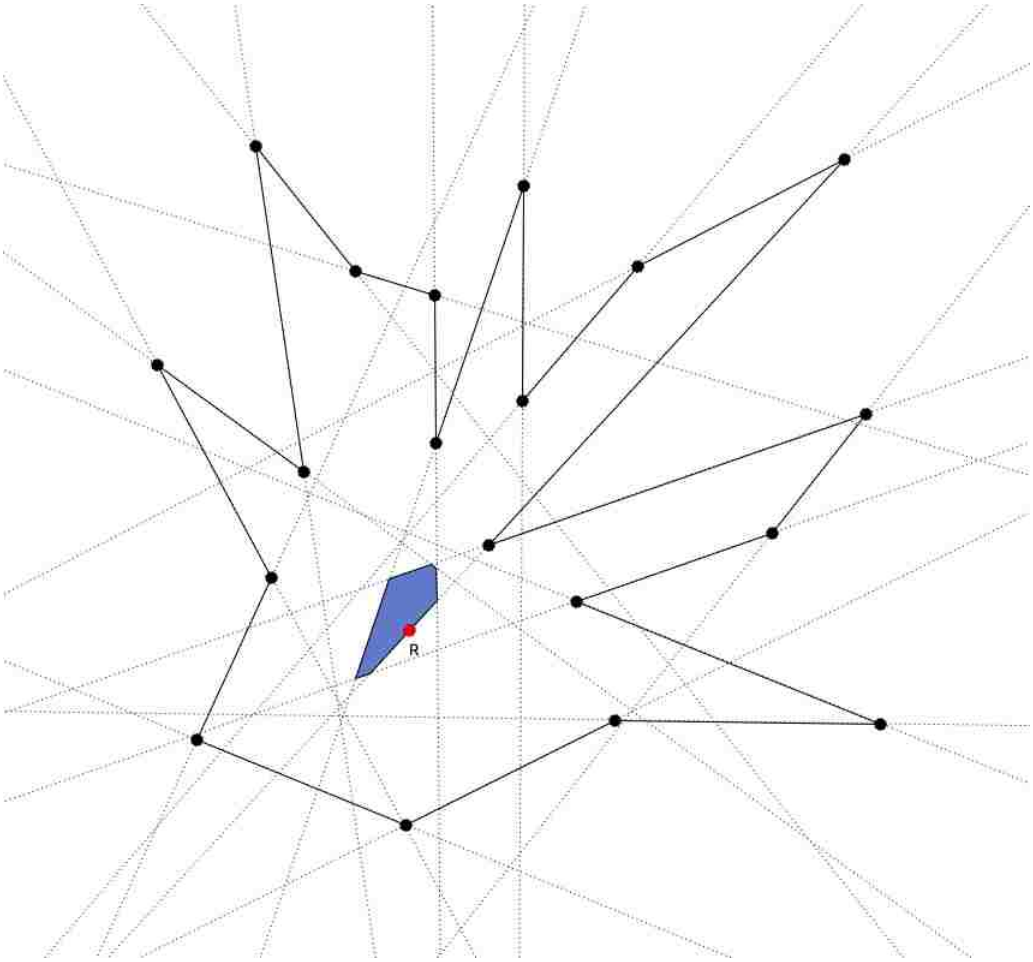


Figure 4.8: A Star-Shaped Polygon with the Kernel Computed by Our Implementation

star-shaped polygon. The rectangular region is divided into four equal small rectangular regions and we repeat the same process recursively in all of them updating the largest kernel every time we find one. Each rectangular region acts as a node of the quadtree and has four child nodes. As mentioned in section 4.2.4, the node has seven properties. The Python implementation of a node is given in listing 4.5.

```
class Node():
    def __init__(self, x0, y0, w, h, level, handle):
        self.x0 = x0 # x_mid
        self.y0 = y0 # y_mid
        self.width = w # width of the rectangular region
        self.height = h # height of the rectangular region
        self.level = level # level of the node
        self.handle = handle # handle for computing the kernel
        self.kernel = self.handle.compute_kernel() # compute kernel
            taking (x0, y0) as a ref point
        self.children = [] # child nodes
```

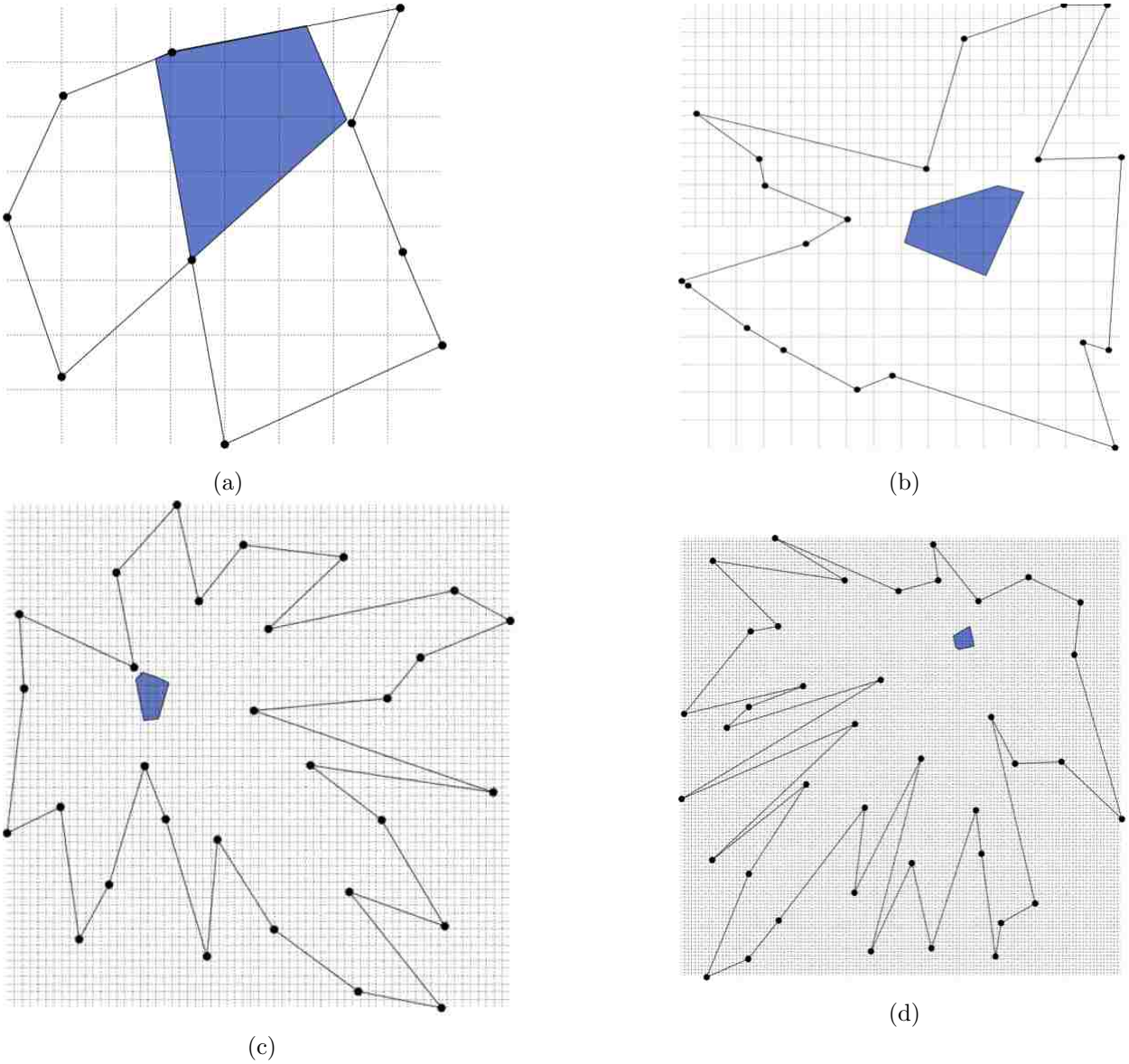
Listing 4.5: A Python Class That Models the Node Data Type

The partitioning follows the breadth-first order i.e. we compute the kernel on the sibling nodes before going to the child nodes. We use the queue data structure from the python **Queue** library. Every time we pop a node, we push its child nodes into the queue. This process goes on as long as the queue is not empty. The method **subdivide** which finds the largest kernel by partitioning the rectangle is given in Listing 4.6.

```
def subdivide(node, k, max_kernel, handle):
    queue = Queue.Queue(0)
    queue.put(node)
    while(not queue.empty()):
        node =  queue.get()=
        if max_kernel[1] < node.kernel[1]:
            max_kernel[0] = node.kernel[0]
            max_kernel[1] = node.kernel[1]
            max_kernel[2] = node.kernel[2]
            max_kernel[3] = node.kernel[3]
        if max_kernel[1] < 2 * node.width * node.height:
            w_ = float(node.width/2)
            h_ = float(node.height/2)
            # Top-left region
            x1 = Node(node.x0 - w_/2, node.y0 - h_/2, w_, h_, node.
                level + 1, handle)
            queue.put(x1)
            # Top-right region
            x2 = Node(node.x0 + w_/2, node.y0 - h_/2, w_, h_, node.
                level + 1, handle)
            queue.put(x2)
            # buttom-left region
            x3 = Node(node.x0 - w_/2, node.y0 + h_/2, w_, h_, node.
                level + 1, handle)
            queue.put(x3)
            # buttom-right region
            x4 = Node(node.x0 + w_/2, node.y0 + h_/2, w_, h_, node.
                level + 1, handle)
            queue.put(x4)
            node.children = [x1, x2, x3, x4]
```

Listing 4.6: A Python Program that Finds the Largest Kernel

The result of the implementation for four randomly generated point sites with size 10, 20, 30, and 40 are given in Figure 4.9. The grids in the figures are the partitioning of the region produced by the implementation.



(a) (b) (c) (d)

Figure 4.9: The Star Shaped Polygons with Largest Kernel Produced by Our Implementation

We did some experiments with the largest kernel for a given set of points. We randomly generated points and computed the area of the maximum kernel, a ratio of the area of the maximum kernel to the area of the generated polygon, the type of the kernel whether it is floating (F) or a

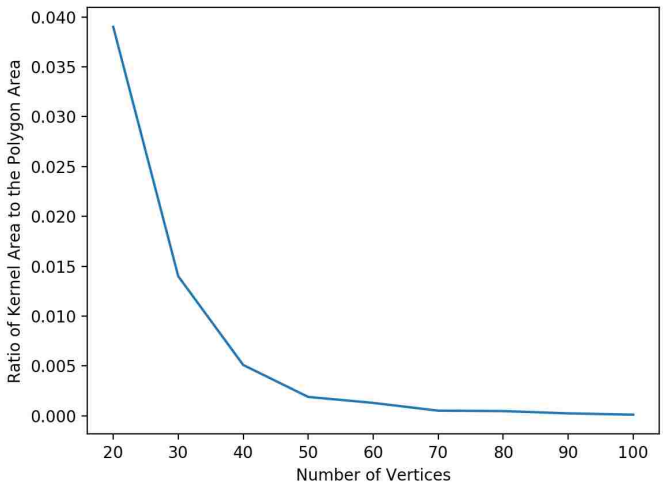


Figure 4.10: Change in Ratio of Kernel Area to Polygon Area with respect to the Change in Number of Vertices

boundary (B) kernels and the height of the QuadTree. Table 4.1 shows the result of the experiment.

As we see in the table, the size of the kernel decreases with the increase in the number of point sites provided the area on which the point sites lie remain the same. The graph in Figure 4.10 illustrates this where we plot the number of vertices in x-axis and ratio of kernel area to polygon area in y-axis. We also notice that the height of the tree increases as the ratio decreases. This is an expected behaviour as the kernel size tends to decrease with the increase in height. This is illustrated in Figure 4.11. Another thing we notice from the table is the type of the kernel. The kernel tends to become floating a kernel as the number of point sites grow.

4.6 Finding the Most Sensitive Vertex

We introduced the concept of the *most sensitive vertex* in Section 3.3. In this section, we discuss the implementation. Once we have a polygon P with the largest kernel, we perfom the following steps to find the most sensitive vertex.

1. Pick the left-most vertex v_0.

2. Remove v_0, join the anti-clockwise neighbour v_{-1} and clockwise neightbour v_1 of v_0 to make the polygon P_1.

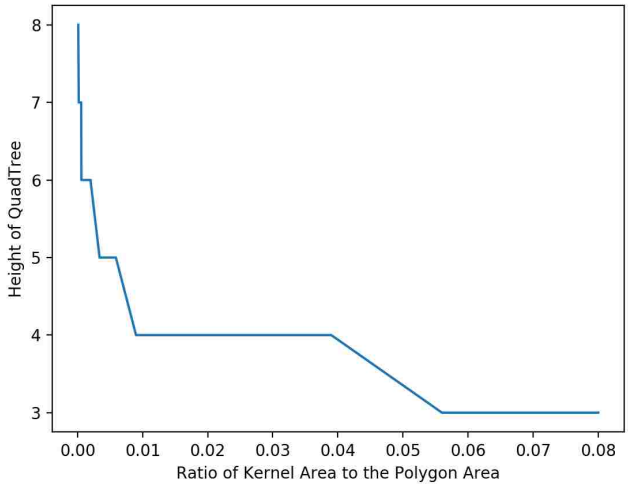


Figure 4.11: Change in Height of QuadTree with respect to the Change in Ratio of Kernel Area to Polygon Area

3. Find the kernel of P_1 and the area (A_0) of the kernel.

4. Store the area A_0 and the vertex v_0 temporarily.

5. Put back v_0 in P_1 to get the original polygon P.

6. Pick the next clockwise vertex of v_0 and repeat steps 2 - 5. Continue this until all the vertices have been processed.

7. Find the maximum area $A_{max} = \max(A_0, A_1, ..., A_{n-1})$. The corresponding vertex is the most sensitive vertex.

No of vertices	Set	Kernel area (sq. unit)	Ratio	Type of Kernel	Height
20	Set 1	6573	0.027	B	4
	Set 2	9279	0.039	B	4
	Set 3	11202	0.037	F	3
	Set 4	25147	0.080	B	3
	Set 5	11181	0.056	F	3
30	Set 1	2862	0.009	F	4
	Set 2	5891	0.014	F	4
	Set 3	5932	0.022	F	4
	Set 4	7260	0.025	F	4
	Set 5	3549	0.012	B	4
40	Set 1	1525	0.0059	F	5
	Set 2	955	0.0040	F	5
	Set 3	1272	0.0053	F	5
	Set 4	1570	0.0047	F	5
	Set 5	1535	0.0051	F	5
50	Set 1	489	0.0016	B	6
	Set 2	337	0.0014	F	6
	Set 3	536	0.0017	F	6
	Set 4	780	0.0034	F	5
	Set 5	575	0.0018	F	6
60	Set 1	446	0.0015	F	6
	Set 2	342	0.0013	F	6
	Set 3	301	0.0019	F	6
	Set 4	613	0.0019	F	6
	Set 5	570	0.0020	F	6
70	Set 1	267	0.00089	F	6
	Set 2	146	0.00049	F	7
	Set 3	135	0.00052	F	7
	Set 4	145	0.00043	F	7
	Set 5	175	0.00060	F	6
80	Set 1	179	0.00065	F	6
	Set 2	130	0.00039	F	7
	Set 3	140	0.00048	F	7
	Set 4	68	0.00027	F	7
	Set 5	105	0.00037	F	7
90	Set 1	56	0.00022	F	7
	Set 2	167	0.00042	F	6
	Set 3	62	0.00020	F	7
	Set 4	64	0.00025	F	7
	Set 5	149	0.00057	F	7
100	Set 1	71	0.00024	F	7
	Set 2	63	0.00022	F	7
	Set 3	61	0.00020	F	7
	Set 4	54	0.00021	F	7
	Set 5	31	0.00012	F	8

Table 4.1: The Result of the Experiemnt with the Largest Kernels for a Given Set of Vertices

The corresponding python implementation is given in Listing 4.7.

```
max_area = handle.compute_kernel()
temp_points = copy.deepcopy(points)
max_v = points[0]

for i in range(len(points)):
    del temp_points[i]
    area = ssp.compute_kernel(points = temp_points)
    if area > max_area:
        max_area = area
        max_v = points[i]
    temp_points = copy.deepcopy(points)
```

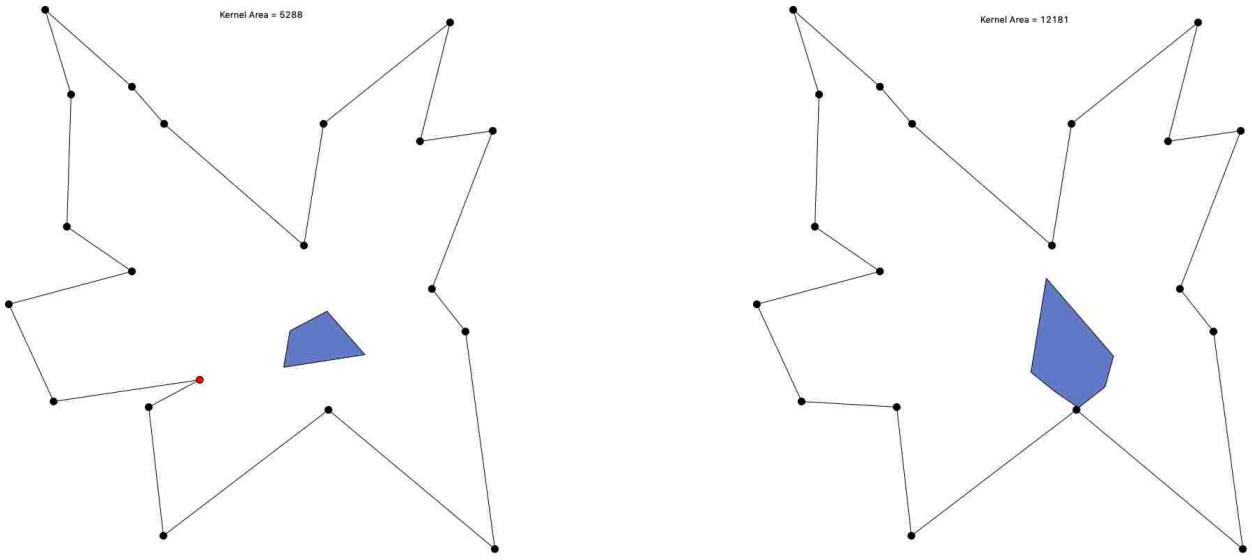
Listing 4.7: A Python Program to Find the Most Sensitive Vertex

Figure 4.12 shows the execution of our implementation before and after the removal of the most sensitive vertex.

Figure 4.12 shows that we can increase the area of the kernel significantly by just removing a single vertex. We did an experiment with randomly generated star-shaped polygons with largest kernel. We removed the most sensitive vertex and noted the percentage change in the kernel. Table 4.2 shows this result along with the type of vertex and type of the kernel.

No of vertices	Set	Before	After	Change	Vertex Type
10	Set 1	252203	273535	+8%	Reflex
	Set 2	122608	158255	+29%	Reflex
	Set 3	261508	381782	+46%	Reflex
	Set 4	27563	543784	+97%	Reflex
	Set 5	43766	207916	+375%	Reflex
20	Set 1	20955	33182	+58%	Convex
	Set 2	31595	43368	+38%	Reflex
	Set 3	16233	31166	+92%	Reflex
	Set 4	9270	11908	+28%	Convex
	Set 5	12497	26478	+112%	Convex
30	Set 1	4496	9289	+107%	Reflex
	Set 2	3898	5130	+32%	Reflex
	Set 3	6230	8563	+37%	Reflex
	Set 4	1777	2444	+38%	Convex
	Set 5	1657	3845	+132%	Reflex
40	Set 1	905	1355	+50%	Reflex
	Set 2	1010	1693	+68%	Reflex
	Set 3	1104	2307	+109%	Reflex
	Set 4	1978	3437	+74%	Convex
	Set 5	1882	3957	+110%	Convex

Table 4.2: Percentage Change in the Area of the Kernel with the Removal of the Most Sensitive Vertex.



(a) (b)

Figure 4.12: Illustrating the Most Sensitive Vertex (a) The Kernel Before the Removal of Most Sensitive Vertex and (b) The Kernel After the Removal.

Chapter 5

Conclusion

We summarized a critical review of existing algorithms for random generation of polygonal shapes. We also reviewed algorithms related to visibility properties (particularly kernel counting and detection) of simple polygons. We proposed an algorithm for generating random polygons that tend to have large size kernels. The algorithm for generating such polygons is based on quad-tree searching.

We performed a thorough experimental investigation of an algorithm for generating kernel-aware random polygons. As expected the size of the kernel decreases as the number of vertices increases. This is depicted in Table 4.1 and Figure 4.10. We tried to use the Voronoi Diagram to look for the position of a large-size kernel. It appears that the Voronoi diagram is not capable of computing the large-size kernel positions. But this issue needs further investigation. For certain distributions of input nodes, the Voronoi diagram may be able to spot large-size kernel positions. this is an interesting question for further work.

The proposed quad-tree based algorithm stops after reaching a certain value for the height of the implied quad-tree. How to pick the bound for such height is an important question in itself. In our experimental investigation, the algorithm stops after reaching the height in the range 6 - 8. How to determine this stopping height range is a very critical question which also needs further exploration.

An important related question is the selection of a sub-set Q of input node set S such that the corresponding polygon has large size kernel. One possible way of selecting such sub-set is using the convex layers of S. We recursively find the convex hull of S as illustrated in Figure 5.1. We can use the following greedy strategy to remove the layers.

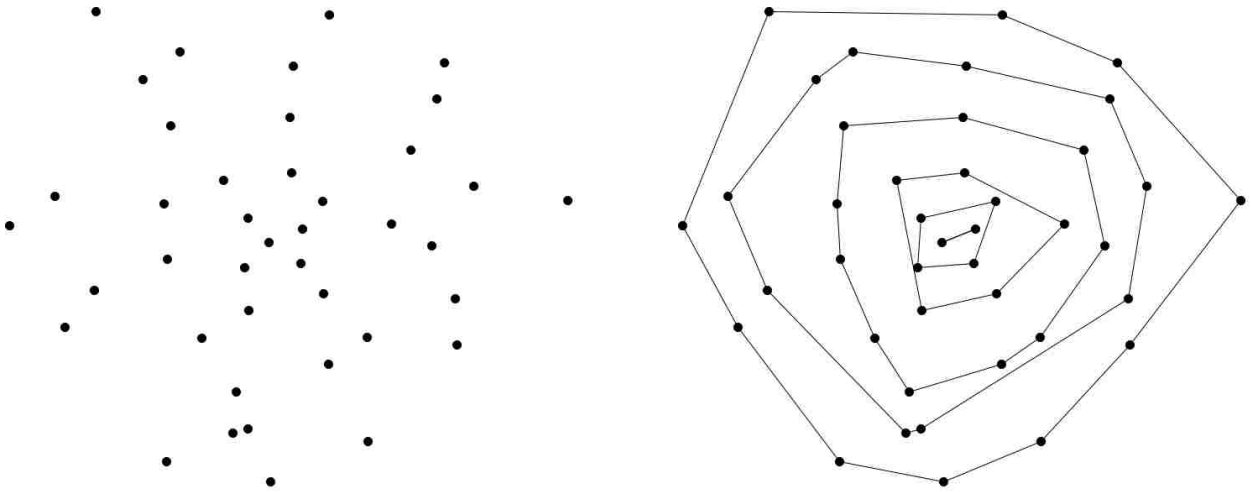
1. Remove the innermost layer l_1 from S and compute the largest kernel K_1 on the remaining

points.

2. Put l_1 back in S. Remove the next layer in an outward direction and repeat the same process.

3. Remove the layer l_i corresponding to the maximum value of K_i from S to get Q.

We can use the above strategy multiple times to remove more than one layer.



(a) (b)

Figure 5.1: Illustration of Convex Layers (a) Given set of points S and (b) Corresponding Convex Layers

Bibliography

[AH98] Thomas Auer and Martin Held. Rpg – heuristics for the generation of random polygons. 12 1998.

[Asa85] Tetsuo Asano. An efficient algorithm for finding the visibility polygon for a polygonal region with holes. *Transactions of the Institute of Electronics and Communication Engineers of Japan. Section E*, 68, 09 1985.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, 22(4):469–483, 1996.

[EA81] Hossam A. ElGindy and David Avis. A linear algorithm for computing the visibility polygon from a point. *J. Algorithms*, 2:186–197, 1981.

[FB74] R. A. Finkel and J. L. Bently. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, Mar 1974.

[Gew95] Laxmi P. Gewali. Recognizing s-star polygons. In *Pattern Recognition*, pages 1019–1032, 1995.

[Lee83] D Lee. Visibility of a simple polygon. *Computer Vision, Graphics, and Image Processing*, 22:207–221, 05 1983.

[Leh92] Jürgen Lehn. Pseudorandom number generators. In Peter Gritzmann, Rainer Hettich, Reiner Horst, and Ekkehard Sachs, editors, *Operations Research '91*, pages 9–13, Heidelberg, 1992. Physica-Verlag HD.

[LPP79] D Lee and F P. Preparata. An optimal algorithm for finding the kernel of a polygon. *Journal of the ACM (JACM)*, 26:415–421, 07 1979.

[O'R87] Joseph O'Rourke. *Art gallery theorems and algorithms*, volume 57. Oxford University Press Oxford, 1987.

[Soh99] Christian Sohler. Generating random star-shaped polygons. In *CCCG*, 1999.

[Vas15] Tzvetalin S. Vassilev. Visibility : Finding the staircase kernel in orthogonal polygons. 2015.

[Wik19] Wikipedia contributors. Shoelace formula — Wikipedia, the free encyclopedia, 2019. [Online; accessed 1-April-2019].

[ZSSM96] Chong Zhu, Gopalakrishnan Sundaram, Jack Snoeyink, and Joseph S.B. Mitchell. Generating random polygons with given vertices. *Computational Geometry*, 6(5):277 – 290, 1996. Sixth Canadian Conference on Computational Geometry.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Bibek Subedi

subedishankar2011@gmail.com

Degrees:

Master of Science in Computer Science 2019

University of Nevada Las Vegas

Thesis Title: Generating Kernel Aware Polygons

Thesis Examination Committee:

Chairperson, Dr. Laxmi Gewali, Ph.D.

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

Committee Member, Dr. Kazem Taghva, Ph.D.

Graduate Faculty Representative, Dr. Henry Selvaraj, Ph.D.