May 2016

# Efficient Algorithms for Clustering Polygonal Obstacles

Sabbir Kumar Manandhar
*University of Nevada, Las Vegas*, manans1@unlv.nevada.edu

# EFFICIENT ALGORITHMS FOR CLUSTERING POLYGONAL OBSTACLES

by

Sabbir Kumar Manandhar

Bachelor of Computer Engineering
Tribhuvan University, Nepal
Institute of Engineering, Pulchowk Campus
2010

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 2016

UNLV | GRADUATE COLLEGE

## Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 21, 2016

This thesis prepared by

Sabbir Kumar Manandhar

entitled

Efficient Algorithms for Clustering Polygonal Obstacles

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 Interim Dean*

John Minor, Ph.D.
*Examination Committee Member*

Justin Zhan, Ph.D.
*Examination Committee Member*

Henry Selvaraj, Ph.D.
*Graduate College Faculty Representative*

# Abstract

Clustering a set of points in Euclidean space is a well-known problem having applications in pattern recognition, document image analysis, big-data analytics, and robotics. While there are a lot of research publications for clustering point objects, only a few articles have been reported for clustering a given distribution of obstacles. In this thesis we examine the development of efficient algorithms for clustering a given set of convex obstacles in the 2D plane. One of the methods presented in this work uses a Voronoi diagram to extract obstacle clusters. We also consider the implementation issues of point/obstacle clustering algorithms.

# Acknowledgements

I extend my sincere gratitude to my supervisor *Dr. Laxmi Gewali, PhD* for guiding me through every small details of the thesis and make me understand the thesis concept clearly. I would also like to thank *Dr. Ajoy Datta, PhD* who has been around to help regarding all the administrative and academic problems since the beginning. I am grateful to *Dr. John Minor, PhD* and *Dr. Justin Zhan, PhD* for being part my thesis committee.

I am thankful to my parents and my wife who have always been encouraging during the difficult times.

At last, but not the least, I sincerely thank to all my friends, seniors and juniors for all their support, love and invaluable suggestions.

Sabbir Kumar Manandhar

*University of Nevada, Las Vegas*

*May 2016*

# Table of Contents

# List of Tables

# List of Figures

# List of Algorithms

# Chapter 1

# Introduction

With the advent of mobile robots like drones, it becomes important to plan collision-free paths for their motion. Robots cannot make intelligent decisions about where to move. Robots have to traverse paths that do not collide with obstacles.

Several algorithms exist for path planning in the presence of obstacles [GM91][HS97][HSY13][O'R98]. Grid-based search, interval-based search, geometric reasoning, potential fields, and sample-based constructions are some of the widely used motion planning techniques [GO97][SU00].

In this thesis, we investigate the development of efficient algorithms for aggregating a given distribution of polygonal obstacles into clusters, which has application in planning collision free paths.

Clustering is the process of organizing the given distribution of entities into a few groups such that members in a group are closely related in some measure. The proximity measure to put two entities in the same group depends on the intended applications. One of the most frequently used proximity measures is based on Euclidean distance. In this approach, an entity $x$ belongs to a group $g$ if the Euclidean distance between $x$ and a member of $g$ is small.

The most commonly used objects for clustering are the distribution of points in Euclidean space. While the problem of clustering points in Euclidean space is a well-investigated problem [Lly82][GRS][KMN$^+$02], only a few authors have considered the clustering of polygonal obstacles [Jos11]. A simple illustration of obstacle clustering is shown in Figure 1.1. This figure shows that obstacles could be clustered in more than one way.

This thesis is organized as follows. In Chapter 2, we present a critical review of i) geometric structures for developing path planning algorithms and ii) widely used algorithms for clustering points distributed in two dimensions. In particular, we closely examine the quality of solutions obtained by using *k-means* algorithms and their variations. In Chapter 3, we describe the main contribution of this thesis. We discuss several approaches for clustering a given set of obstacles. We first develop and present two obstacle clusterings using *Closest Neighbor Computation* as a clustering algorithm. While the first one is based on direct computation of shortest separation between all pair of obstacles, the second one uses a visibility graph induced by polygonal obstacles to identify members belonging to the same cluster. The time complexity of these methods are

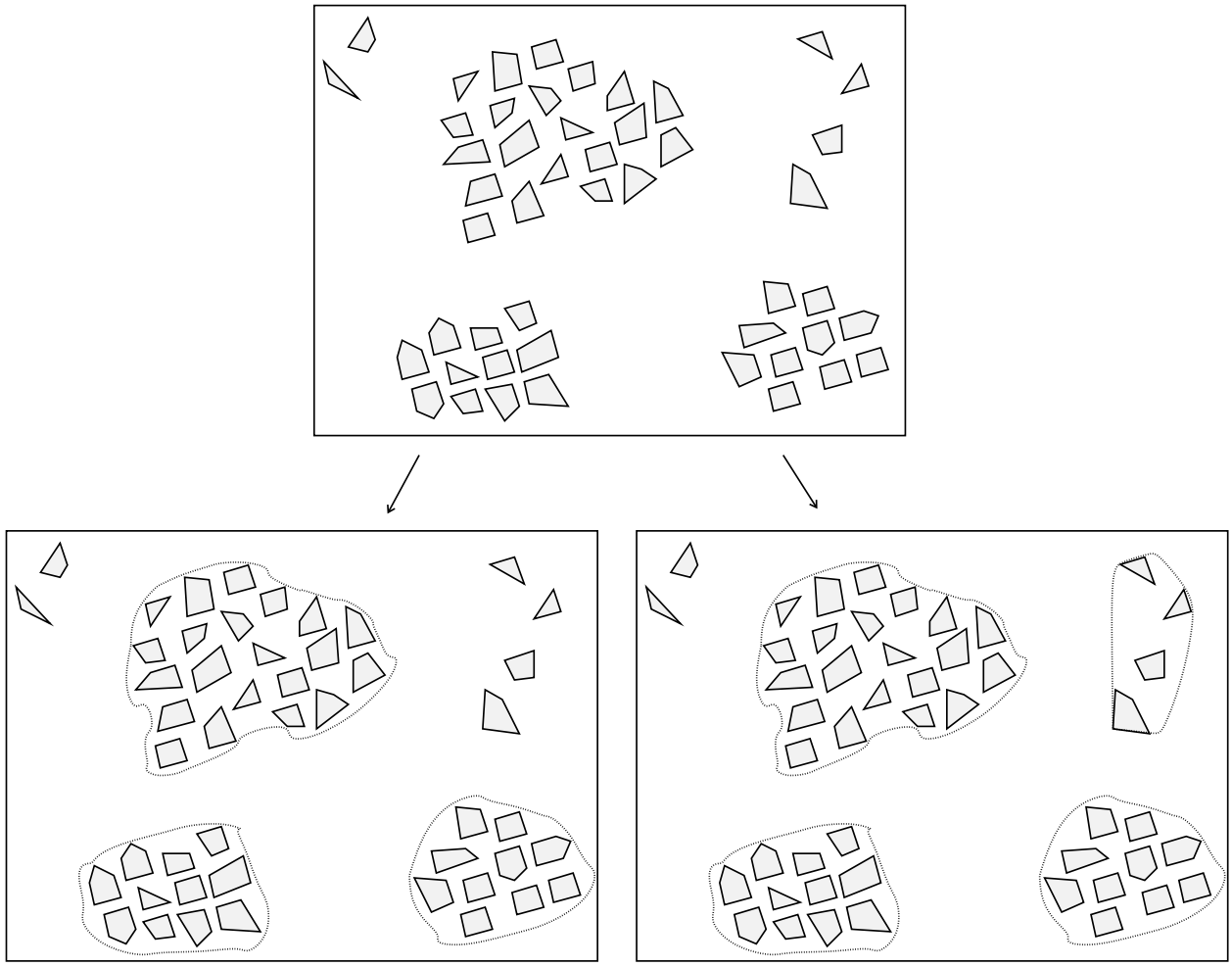

Figure 1.1: A Typical Obstacle Clustering Instance (Two Ways - or More)

rather high. We then present a faster algorithm based on the *Voronoi diagram*. To apply a Voronoi diagram for identifying nearest obstacle pairs, sample points on obstacles are used as *Voronoi Sites*. This approach leads to a faster algorithm for identifying obstacle clusters.

In Chapter 4, we address the implementation issues of obstacle clustering algorithms. The implementation is done in C++ programming language using the *Qt* framework for graphical user interface (GUI) design. The prototype program allows users to enter points interactively. It also allows users to save the entered point distribution and obstacle distribution for future use. The implemented algorithms include i) Shortest Distance Approach, ii) k-means Approach and iii) Voronoi Diagram Approach.

Finally, in Chapter 5, we present possible extensions for future research on problems and algorithms presented in this thesis. We also point out scopes for further improvement of implementation of the proposed algorithms.

# Chapter 2

# Background and Preliminaries

In this chapter we present an overview of algorithmic tools that are useful for capturing a cluster of points distributed in the two dimensional plane. The algorithmic tools we review include (i) Construction of visibility graphs, (ii) computation of collision-free paths, and (iii) cluster identification for points distributed in the two dimensional plane.

## 2.1 Visibility Graph

In Chapter 3, the visibility graph induced by a set of obstacles will be used to identify clusters of obstacles. It is therefore very critical to understand techniques for constructing visibility graphs. With this motivation we consider a brief review of algorithms for computing visibility graphs induced by polygonal obstacles.
If we have a set of polygonal obstacles, then the ***visibility graph*** induced by them can be defined in terms of the vertices of the obstacles and ***visibility edges*** connecting them.
Let $R = \{O_1, O_2, ...O_k\}$ be the given set of polygonal obstacles in the plane. The ***visibility graph*** induced by $\boldsymbol{R}$, denoted as ***VG(V, E)*** consists of vertices which are precisely the vertices of obstacles in $\boldsymbol{R}$. That is, the vertex set $\boldsymbol{V}$ is

$$V = \{v_i \mid v_i \in O_k\text{'s}\}$$

Two vertices $v_i, v_j \in V$ are connected to form a visibility edge $(v_i, v_j)$ if the line segment connecting $v_i$ to $v_j$ does not intersect with any obstacle. So the set of visibility edges $\boldsymbol{E}$ is

$$E = \{(v_i, v_j) \mid v_i, v_j \in V \text{ and line segment } (v_i, v_j) \text{ does not intersect with any obstacles}\}$$

Figure 2.1 shows an example of a visibility graph. It is noted that the edges of obstacles are also edges of the visibility graph.

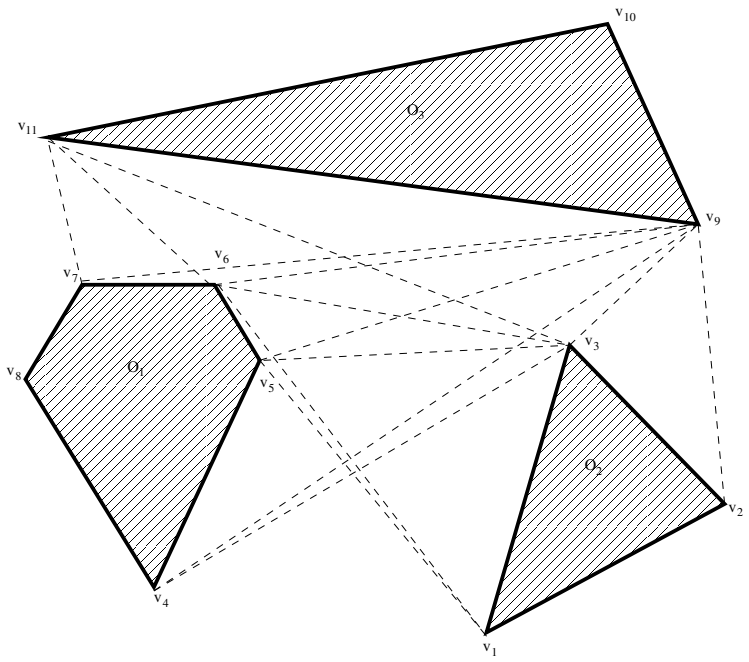


Figure 2.1: Example of Visibility Graph

### 2.1.1 Straightforward Computation of Visibility Graph

This is a rather brute force approach for computing visibility graphs. Any pair of vertices $v_i$ and $v_j$ can be a candidate for the visibility graph. If the line segment connecting $v_i$ to $v_j$ does not intersect with any obstacle then it is taken as a visibility edge. There can be $\binom{n}{2} = \mathcal{O}(n^2)$ candidate line segments as possible visibility edges. To verify one candidate segment we need to check its intersection with $\mathcal{O}(n)$ edges of the obstacles. Hence verification from one edge takes $\mathcal{O}(n)$ time. There are $\mathcal{O}(n^2)$ candidate segments and all verification takes $\mathcal{O}(n^3)$ time.

This approach is very simple to compute. However, this method is slow and therefore seldom used for computing visibility graphs induced by large numbers of obstacles. A formal sketch of this algorithm is listed as Algorithm 2.1.

**Algorithm 2.1:** Finds all the Visibility Graph Edges

**Input**: A finite set $S = \{O_1, O_2, \ldots, O_k\}$ of polygon obstacles with vertices $v_i, v_j, \ldots, v_n$

**Output**: A finite set of Visibility Edges E

```
1 V = {v_1, v_2, ..., v_k}, E = φ
2 for i ← 1 to n do
3     for j ← 1 to n do
4         if i ≠ j then
5             if (v_i, v_j) does not intersect any obstacle then
6                 include (v_i, v_j) in E
7 return E
```

### 2.1.2 Lee's Rotating Ray Algorithm

The trivial straightforward algorithm which runs in $\mathcal{O}(n^3)$ is inefficient and is not used in practice. D.T.Lee [Lee78][Kit] has given the first nontrivial solution to the computation of visibility graphs and it runs in $\mathcal{O}(n^2 \log n)$

Lee's algorithm uses a rotating ray to determine vertices visible from an obstacle vertex as shown in

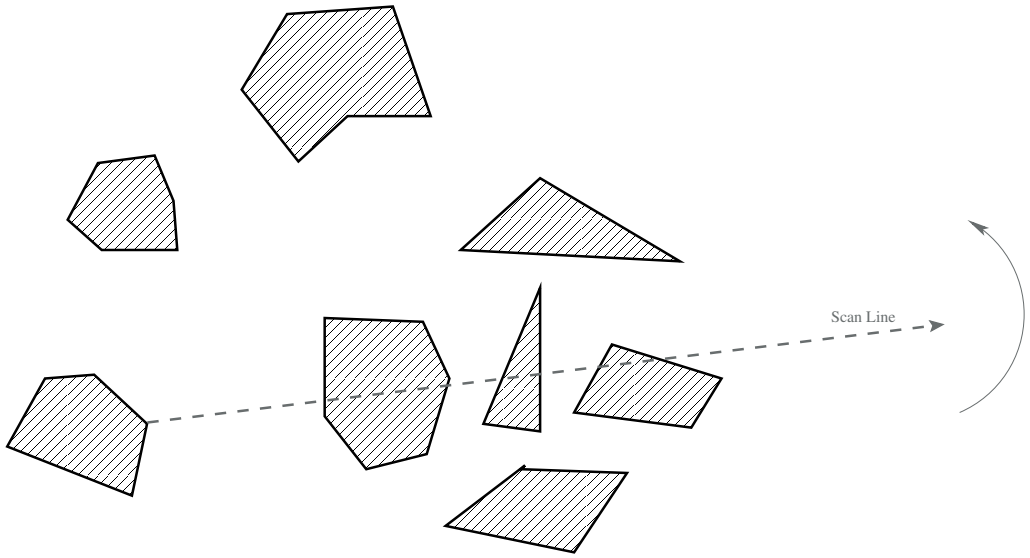


Figure 2.2: Example of Lee Scan

Figure 2.2. From each obstacle vertex $v_i$, a ray $r_i$ originating around $v_i$ scans other vertices by performing

rotation around $v_i$. During the rotation, the algorithm maintains the set of obstacle edges intersected by the ray. The intersected edges are maintained in a height balanced search tree such as am AVL tree or a Red-Black tree. This is done to attain the overall complexity of $\mathcal{O}(n^2 \log n)$. The list must support the insertion and retrieval of edges in $\mathcal{O}(\log n)$ time. This complexity is supported by the balanced search trees. The edges are stored in the tree in order of their distances from the origin of the ray. During angular rotation, the ray stops at the vertices of the obstacles. The events occurring when the ray stops at vertices can be distinguished into three kinds. Let $v_j$ be the vertex on which $r_i$ stops. *Type 1* event occurs when edges of obstacles incident at $v_j$ are to the right side of the ray as shown in vertex $v_j = b$ in Figure 2.3. In *Type 2* event, both obstacle edges incident on $v_j$ are on the left side of $r_i$ as shown in vertex $v_j = a$ in the figure. Finally, in *Type 3* event, one edge incident at $v_j$ is to the left of $r_i$ and the other is on the right as shown in vertex $v_j = c$ in the figure. Edge insertion/deletion operations are performed during these events. In *Type 1* event, both obstacle edges incident on the ray are deleted from the tree. On the contrary, in *Type 2* event, both obstacle edges incident on the ray are inserted into the tree. Finally, in *Type 3* event, the obstacle edge to the left of the ray is inserted into the tree while the edge to the right is deleted. The main working principle of this algorithm is, at any of the event discussed above, if the vertex belongs to the first edge in the tree, then that vertex is visible from the obstacle vertex around which the ray is rotating.

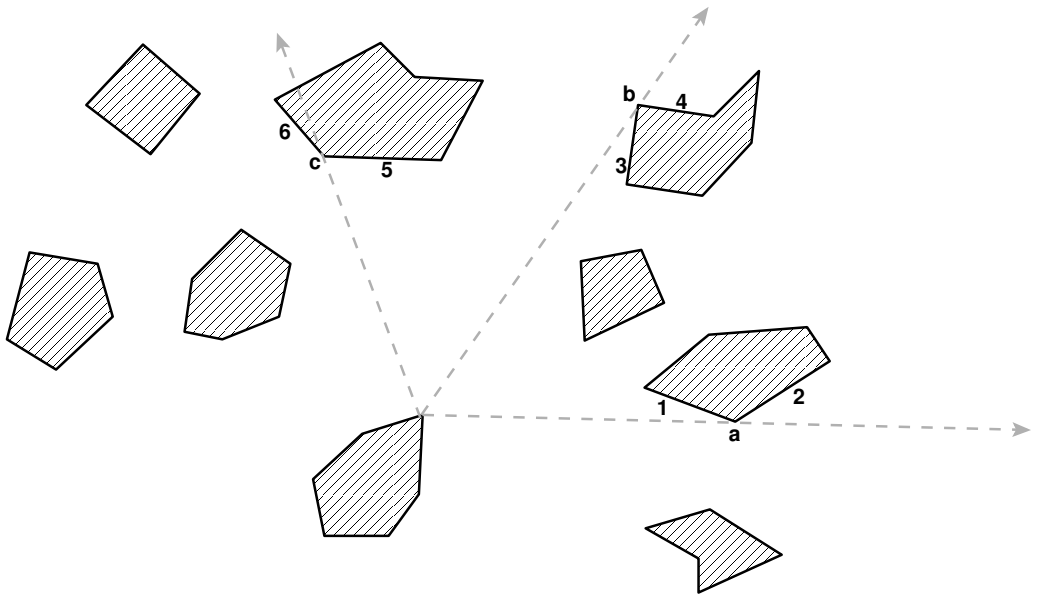


Figure 2.3: Update of Edge List with Lee Scan

A formal sketch of Lee's algorithm is listed as Algorithm 2.2.

**Algorithm 2.2:** Finds all the Visibility Graph Edges

**Input**: A finite set $S = \{O_1, O_2, \ldots, O_k\}$ of polygon obstacles with vertices $v_i, v_j, \ldots, v_n$

**Output**: A finite set of Visibility Edges E

```
1  V = {v_1, v_2, ..., v_n}, E = φ
2  Let T be a height balanced empty search tree.
3  for each vertex v_i ∈ V do
4      Sort vertices in { V - v_i } in angular fashion around v_i
5      Let W be the sorted vertices list.
6      for j ← 1 to n − 1 do
7          v_j ← next vertex in W
8          Delete v_j from W
9          Let e_1 and e_2 be obstacle edges incident at v_j
10         if both e_1 and e_2 are to the right of the ray (v_i, v_j)→ then
11             delete both e_1 and e_2 from the tree T
12         else if both e_1 and e_2 are to the left of the ray (v_i, v_j)→ then
13             insert both e_1 and e_2 into the tree T
14         else
15             //one edge is to the left and other is to the right of the ray (v_i, v_j)→
16             delete the edge that is to the right of the ray
17             insert the edge that is to the left of the ray
18         if v_j corresponds to the root of T then
19             include (v_i, v_j)‾ in E
20 return E
```

### Time Complexity

For processing one vertex (one execution of outer *for* loop), the algorithm performs at most $\mathcal{O}(n)$ insertion/deletion operations. One insert/delete operation takes $\mathcal{O}(\log n)$ time. This implies that the time for one execution of outer *for* loop is $\mathcal{O}(n \log n)$. The total time for processing rotation from all $n$ vertices is $\mathcal{O}(n^2 \log n)$ time.

### 2.1.3 Output Sensitive Algorithm

The fastest algorithm for computing the visibility graph of a polygon with holes was reported by Ghosh and Mount [GM91]. This algorithm runs in time $\mathcal{O}(E + n \log n)$ where $E$ is the size of the visibility graph, $n$ is the number of vertices. The term $n \log n$ is the time needed for triangulating the polygon. The main

ingredient of the algorithm is its clever way for finding the relationship between vertices visible from some point in a given edge and the structure of *funnel*. The concept of *funnel* can be understood by considering an example. Consider vertices visible from some point along a given edge $e_i = (v_i, v_{i+1})$. If we fix one of the vertices $v_r$ in the set of visible vertices, then the shortest path connecting $v_i$ and $v_r$ forms a convex chain shown by the dashed line in Figure 2.4. Similarly, the shortest path connecting $v_{i+1}$ to $v_r$ is another convex chain. It is noted that these convex chains can be simply single edges in some cases. The two convex chains and the given edge $e_i$ form a structure called a *funnel* induced by *apex* $v_r$ and base line $e_i$. When polygon contains holes, there could be more than one funnel induced by a given edge and an apex as shown in Figure 2.5

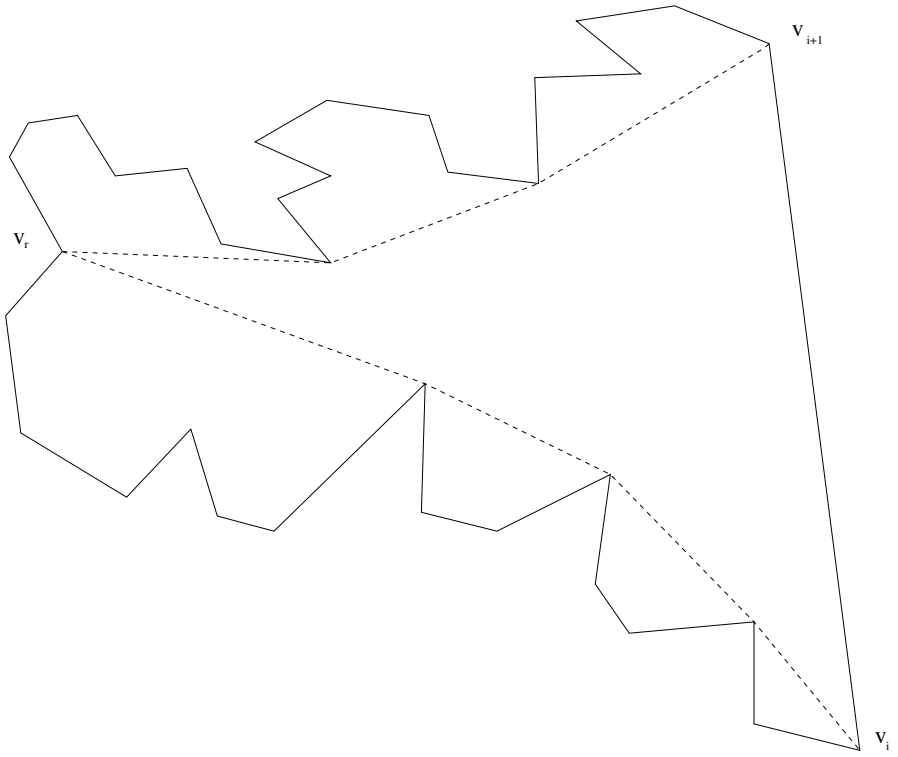


Figure 2.4: Illustration of Funnel Structure

Ghosh and Mount have established that the total number of funnels in a visibility graph with $E$ edges is $\mathcal{O}(E)$. This observation is one of the key steps in developing the improved algorithm.
The visibility edges forming the funnels from an edge have the structure of trees. For a given edge $e_i = (v_i, v_{i+1})$, the tree induced by $v_i$ is called the *lower tree* and that induced by $v_{i+1}$ is called the *upper tree* as shown in the Figure 2.6.

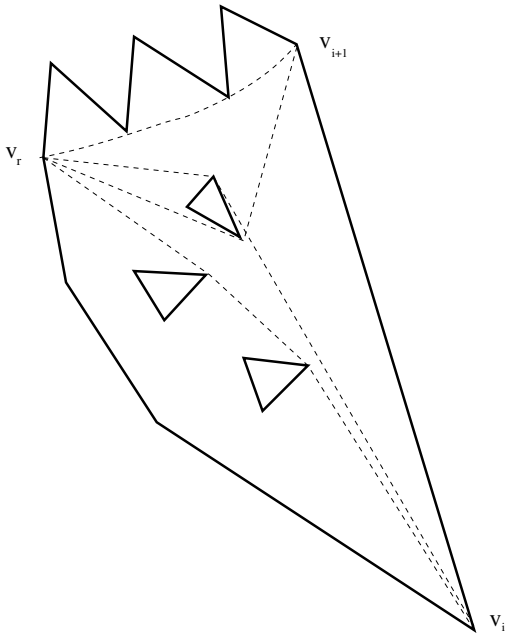


Figure 2.5: Formation of several Funnels

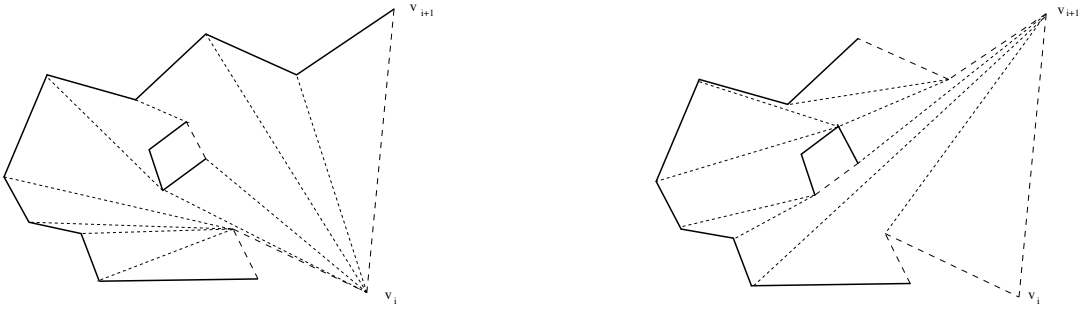


Figure 2.6: Illustration of Lower Tree (left) and Upper Tree (right)

Another key ingredient of their approach is that the clockwise pre-order traversal of the lower tree is the same as the clockwise post-order traversal of the upper tree.

## 2.2 Clustering Point Sites

Clustering is the process of partitioning a set of objects, usually points, into clusters, such that members close to each other are in the same group. Any member in a group should have its nearest neighbor in the same group.

Figure 2.7 and Figure 2.8 show distribution of points in two dimensions. This example illustrates that

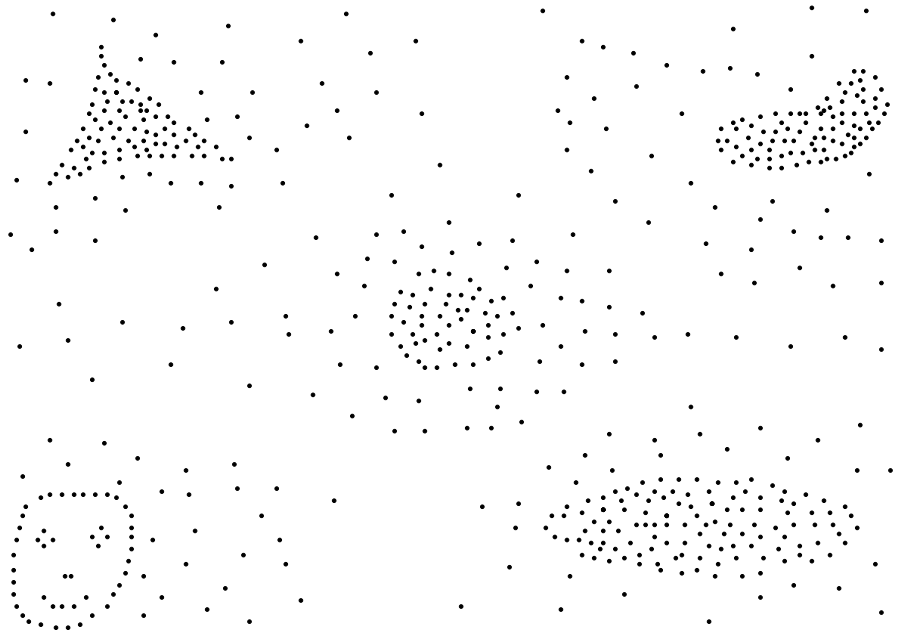

Figure 2.7: Distinct Clusters

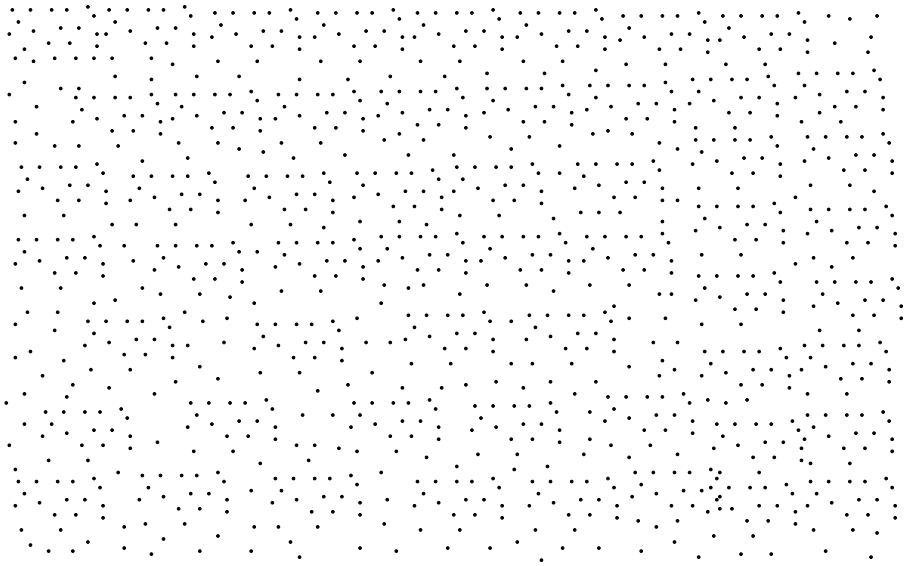

Figure 2.8: Non Distinct Clusters

in some cases there could be distinctly recognizable clusters (Figure 2.7), while in other cases the clusters are not obviously visible (Figure 2.8). Given a set of points $P$ distributed in Euclidean space, the clustering problem is the identification of clusters in $P$. Clustering has important applications in various areas that include pattern recognition, machine learning, image analysis and robotics.

Development of efficient algorithm for identifying clusters in Euclidean space has been considered by several researchers [Lly82][GRS][KMN$^+$02]. One of the earliest algorithms for identifying clusters in points

distributed in two dimension is the *K-means* algorithm proposed by S.P. Lloyd[Lly82]. Many variations of this algorithm have been reported [KMN+02][ARS].

*K-means* algorithm is an iterative algorithm in which clusters are progressively estimated by a series of refinement. In this algorithm, the number of clusters $K$ is assumed to be given. Furthermore, for each cluster, a representative location is somehow estimated. The algorithm assigns input points to one of the representative points. The points belonging to a particular representative are taken as members of that cluster.

The location of the representative is updated by computing the mean of the locations of points in the cluster. The updated point is next used to identify cluster membership.

The Euclidean distance between old location and new location of a representative point is taken as the measure of the progress of modification. The updating of cluster identification continues as long as the progress of modification is more than a certain threshold value. The threshold value can be determined in terms of the separation of the closest pair of input points. A formal sketch of *K-means* algorithm is written as follows and listed as Algorithm 2.3

**Algorithm 2.3:** k-Means Algorithm

**Input**: A finite set of Points $P = \{p_1, p_2, \ldots, p_k\}$ distributed in the plane
**Output**: Clusters $C = \{c_1, c_2, \ldots, c_k\}$ which is a partitioning of input points

```
1  Let k be the input representing number of clusters in C
2  Let δ be a threshold value
3  Representative points m_1, m_2, ..., m_k are somehow selected
4  Let CentroidMovement = δ + 1
5  while CentroidMovement >δ do
6      a. for i ← 0 to n − 1 do
7          Let m_j be the closest representative point to the point p_i
8          Associate the point p_i to the cluster c_j
9      b. Compute new representative points new_m_j's as the centroids of the new points associated
       with c_j's
10     c. set CentroidMovement by comparing m_j's and new_m_j's
11     d. Assign new_c_j's to c_j's
12 return C
```

Figure 2.9 shows an example of cluster estimation by using the *k-means* algorithm. The progress of the movement of cluster representatives is shown by directed line segments. It is observed that the extent of the movement of a cluster's representative becomes progressively smaller after each cluster refinement.

The *k-means* algorithm is fairly a simple algorithm to understand. However there are some limitations to the algorithm. The main drawback of the algorithm is the estimation of number of clusters $k$ and the initial location of their center. For the given input set of points, the output of the algorithm depends upon the initial location. Similarly, the determination of the value of the threshold to stop the iteration of the algorithm is not quite clear. For certain data distributions, using the mean of the points may not be valid, as in the Figure 2.10 (as observed in [GRS]).

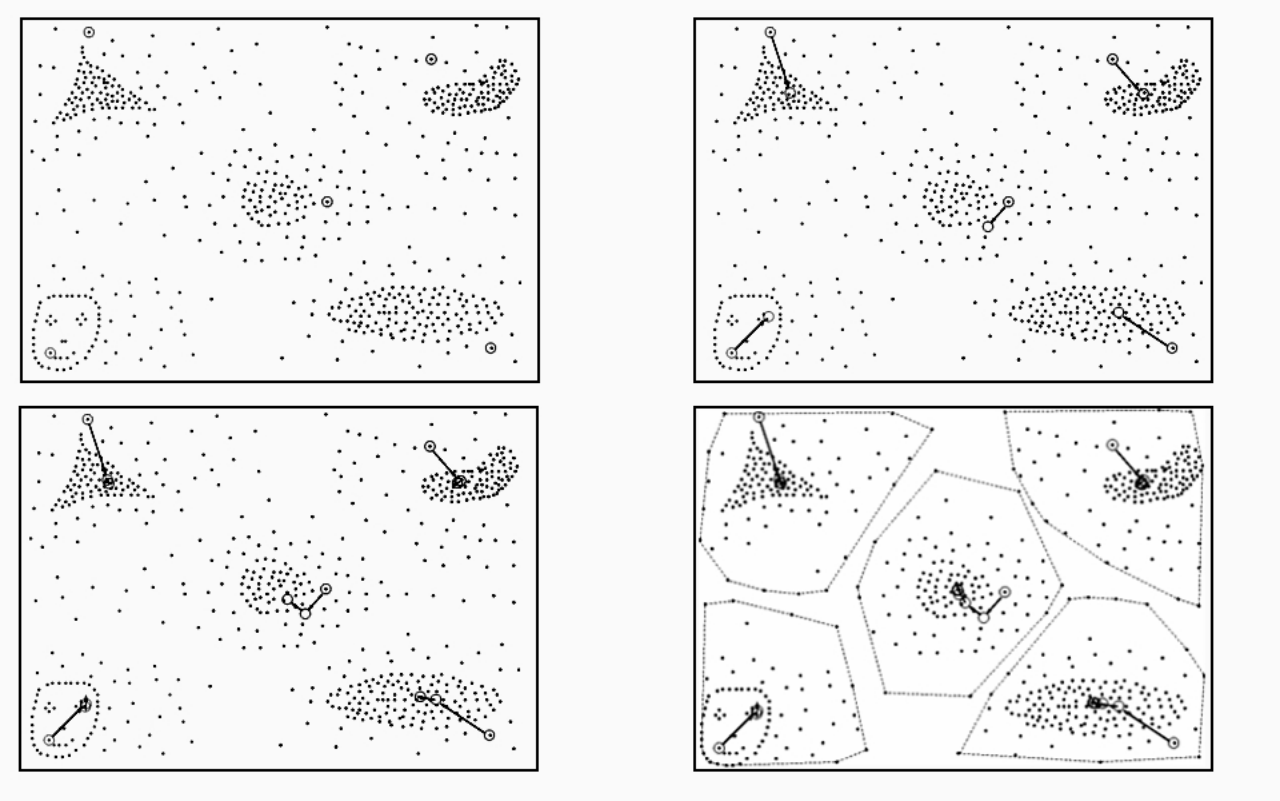

Figure 2.9: Clustering Example

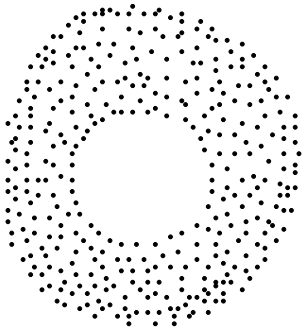

Figure 2.10: Invalid Center

## 2.3 Shortest Path in the Presence of Obstacles

The Euclidean Shortest Path problem is defined as the problem of computing the shortest path between the two points in a plane in the presence of polygonal obstacles such that the path does not intersect any of the obstacles. It is one of the classical problems in computational geometry and has several important applications.

Several research reports have been published on the efficient computation of the shortest path. One of the

most popular methods for computing shortest paths in the presence of polygonal obstacles is based on the use of a visibility graph induced by polygonal obstacles [HS97]. This method works correctly due to the fact that the shortest path is contained in the visibility graph. The actual shortest path is obtained by applying *Dijkstra's Algorithm* on the visibility graph [CLRS09]. The time complexity of the shortest path algorithm based on this method depends on the time complexity of the visibility graph which is $\mathcal{O}(\log n)$. The paper [HS97] gives the optimal-time algorithm known so far for computing shortest paths in the presence of polygonal obstacles. This paper has presented a solution to the problem using the concept of *Shortest Path Map*. For the given source point $s$ and some polygonal obstacles, *Shortest Path Map* is defined as the subdivision of free space into a certain number of regions such that the shortest path from $s$ to any point in the specific region has the same sequence of vertices. The paper discusses the construction of the map by using an efficient implementation of wave front propagation among polygonal obstacles in $\mathcal{O}(n \log n)$ time. The map thus constructed can be used for computing shortest paths from the source point $s$ to any point in the free space in $\mathcal{O}(\log n)$ time.

# Chapter 3

# Obstacle Clustering

## 3.1 Problem Formulation

Consider a collection of polygonal obstacles in two dimensions. For the purpose of clarity of presentation, we consider only convex obstacles. However, at the cost of some time complexity overhead, the algorithmic techniques presented in this chapter are applicable even if the obstacles are not convex. A configuration of around 240 polygonal obstacles is shown in Figure 3.1

A visual examination of the obstacles distribution in Figure 3.1 reveals five clusters. These clusters are

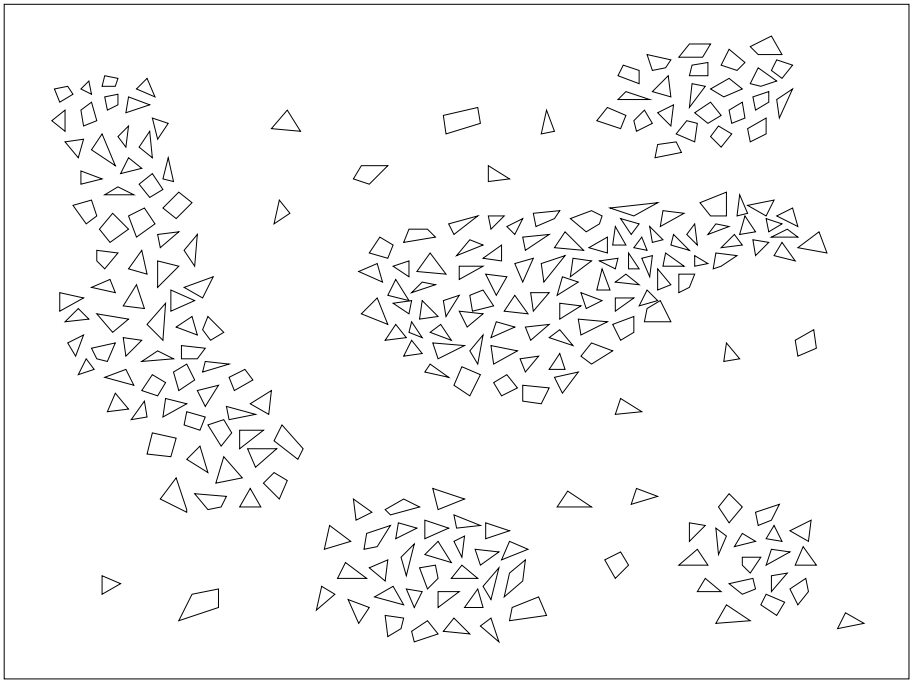

Figure 3.1: Distribution of Convex Obstacles

shown by boundary in Figure 3.2. The problem we investigate is the capturing of clusters of obstacles under

a certain measure.

Broadly speaking, given a parameter $\delta$, we want to group together obstacles whose distance to its nearest

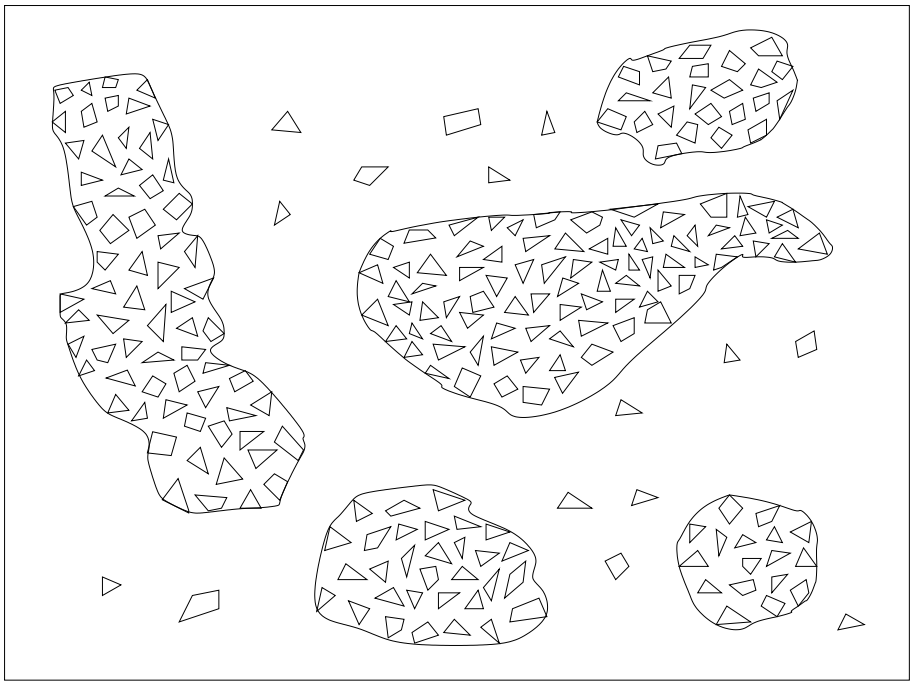

Figure 3.2: Illustrating the boundaries of obstacle clusters

neighbor is no more than $\delta$. The problem can be formally stated as follows.

**Obstacle Clustering Problem (OCP)**

**Given**,

i. A set of convex obstacles $Q_1, Q_2, \ldots, Q_m$ in the plane,

ii. A constant parameter $\delta$

**Question**: Find obstacle clusters such that any obstacle in a cluster has its nearest neighbor within distance $\delta$.

**Remark 3.1**: The distance between two obstacles $Q_i$ and $Q_j$ is the smallest distance between boundary points in $Q_i$ and $Q_j$. When we use the term *distance*, it is understood to be the Euclidean Distance, i.e. the distance $d(p_i, p_j)$ between two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is given by $d(p_i, p_j) = ((x_i - x_j)^2 + (y_i - y_j)^2)^{1/2}$. In some applications, distance between two obstacles is measured from their center of gravity. But in our application we measure distance from boundary points.

## 3.2 Shortest Distance Approach

A pair of obstacles $O_i$ and $O_j$ that are very close to each other should belong to the same cluster. Specifically, if the Euclidean distance between $O_i$ and $O_j$, denoted by $d(O_i, O_j)$ is smaller than the predefined threshold value $\delta$ then $O_i$ and $O_j$ are in the same cluster. We refer to such pairs of obstacles as *$\delta$-proximity pairs*. We can connect a *$\delta$-proximity pair* by the edge $e$ that corresponds to the shortest distance between them. So, to identify all obstacle clusters we could begin by connecting proximity pairs by corresponding shortest edges to obtain the *$\delta$-proximity graph* (or *$\delta$-graph* for short).

Each connected component in the $\delta$-graph is a cluster component. Figure 3.3 and Figure 3.4 illustrate these ideas. Figure 3.3 shows a distribution of convex obstacles with indicated value of predefined parameter $\delta$. The $\delta$-graph for this distribution is shown in Figure 3.4. A straight forward connection obtained by considering all $\delta$-proximity edges can lead to undesired consequences, which happens when a very small or thin obstacle lies between two $\delta$-proximity pairs. This is occurring in the top left corner of Figure 3.3, where the min-separation edge between $O_i$ and $O_j$ intersect with obstacles $O_k$. We need to discard such edges. Rather than considering all $\delta$-proximity pairs we should consider only those that do not squeeze other obstacles in between them. We refer to the graph obtained in this way as the *$\delta$-planar Graph*.

An efficient algorithm for computing minimum distance between convex polygons is a well investigated problem in computational geometry [O'R98][BKOS97]. It is shown that the minimum distance can be computed in $\mathcal{O}(\log n)$ time after a pre-processing overhead of $\mathcal{O}(n)$. So, we can use one of these two algorithms for computing minimum separation between a pair of convex obstacles.

A direct approach for computing the $\delta$-planar graph is to first find the minimum distance between all $\binom{n}{2}$ pairs and then check for intersection of min-separation edges with other polygons. A formal sketch of the algorithm based on this approach is shown as Algorithm 3.1. Once we have the connecting lines we can run the Breadth First Search (BFS) or Depth First Search (DFS) algorithm to find the connected components and hence identify the obstacle clusters.

**Algorithm 3.1:** Algorithm to Capture Connecting Shortest Distances

**Input**: i) Obstacles $Q_1, Q_2, \ldots, Q_m$, ii) Parameter $\delta$
**Output**: Connecting Shortest Edges List $E$

```
1 E = φ
2 Mark all pairs Unprocessed in M[][]
3 for all unprocessed pair Q[i]Q[j] do
4     Let e = distance(Q_i, Q_j)
5     if |e| < δ and e does not intersect with other obstacles then
6         E = E U e
7         Mark M[i][j] as Processed
8 return E
```

The time complexity of Algorithm 3.1 can be done in a straightforward way. Marking the $n \times n$ array

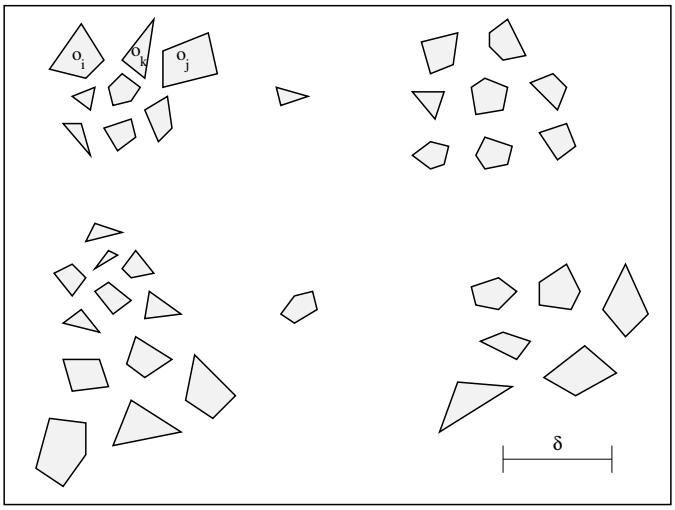


Figure 3.3: Given Obstacles

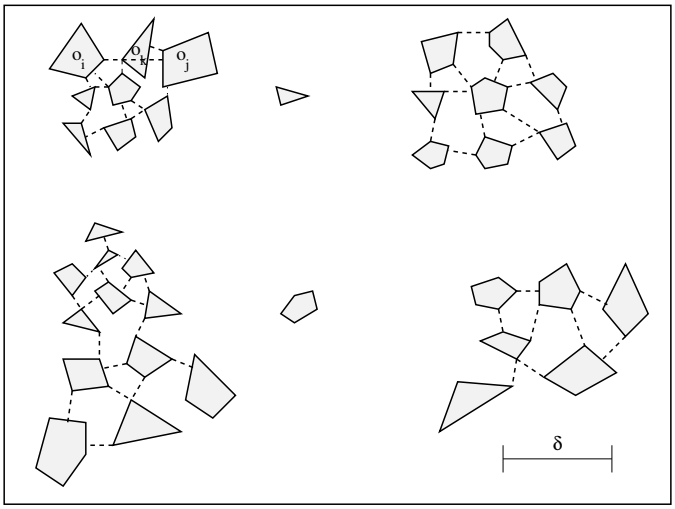


Figure 3.4: $\delta$-Planar Graph

in step 2 takes $\mathcal{O}(n^2)$ time. Step 3 executes in time $\mathcal{O}(n^2)$ since there are $\mathcal{O}(n^2)$ pairs. Checking the second condition in the if statement (step 5) takes $\mathcal{O}(n)$ time. Hence the total time for the for-loop is $\mathcal{O}(n^3)$. Thus the overall time complexity for Algorithm 3.1 is $\mathcal{O}(n^3)$.

The time complexity of Algorithm 3.1 is rather high. Repeatedly checking for intersection of candidate edges with all obstacles is the reason for the high time complexity. With the motivation of developing a faster algorithm, we next examine the feasibility of using a visibility graph (introduced in section 2.1) for developing a better algorithm for capturing obstacle clusters.

## 3.3 Visibility Graph Approach

The visibility graph induced by convex obstacles contains edges corresponding to all visible pairs. Nearer obstacles will have the shorter edges. Depending on the shape of the obstacles, the nearer obstacles can also have some edges which are very large. The edges of the visibility graph can be examined to discard those whose corresponding *separation length* is longer than $\delta$. Here the term *separation length* of a visibility edge $e_i$ should be clarified further. Let $e_i$ connects obstacle vertices $v_r$ and $v_p$ and let obstacle edges incident on $v_r$ and $v_p$ be $e_1$, $e_2$ and $e_3$, $e_4$ respectively as shown in Figure 3.5. Then the separation length corresponding to $e_i$ is either projection to one of the obstacle edges (left part in Figure 3.5) or simply $e_i$ itself.

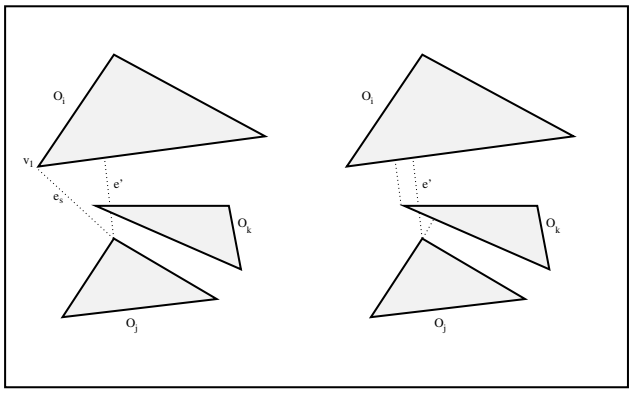


Figure 3.5: Illustrating Separation Length of Visibility Edges

Let $sep(e_i)$ denote separation length corresponding to visibility edge $e_i$. The elimination of visibility edge $e_i$ whose separation length $sep(e_i)$ is larger than $\delta$ is illustrated in Figure 3.7 and Figure 3.8. Figure 3.7 shows all visibility edges induced by the convex obstacles. Most of the edges in the visibility graph in this figure are such that their separation length is longer than $\delta$ and can be discarded. The *reduced visibility graph* is shown in Figure 3.8. The visibility edges in the reduced visibility graph are not edges corresponding to minimum distance. So it is not necessary that the shortest distance be present as the visibility edge of the visibility graph. However, these edges can be examined locally to construct the corresponding separation length. The actual separation edges are drawn thicker in Figure 3.9.
When separation edges are constructed, some of such separation edges could intersect obstacles. In such situations we should be able to find shorter separation edges as stated in the following Lemma 3.3.1.

**Lemma 3.3.1** *The shortest visibility edge $e_i$ from a vertex of an obstacle is such that $sep(e_i)$ can not intersect with another obstacle.*

**Proof:** Let $e_s$ be the shortest visibility edge emanating from vertex $v_1$ of obstacle $O_i$. Let $O_j$ be the obstacle corresponding to other end points of $e_s$. Suppose $sep(e_s)$ intersects with another obstacle $O_k$ (Figure3.6). Now we can observe that the visibility edge connecting $v_1$ and a vertex of $O_k$ is shorter than $e_s$. Contradiction. ■

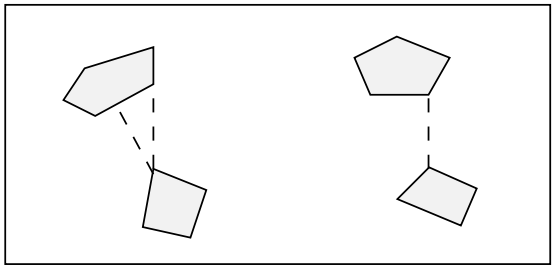

Figure 3.6: Rotated Shortest Distance Edge should not intersect other obstacles

For the purpose of constructing a reduced visibility graph we can simply consider the shortest visibility edge from each vertex and locally compute separation edges. The algorithm for capturing obstacle clusters from a visibility graph can be formally sketched as shown in Algorithm3.2.

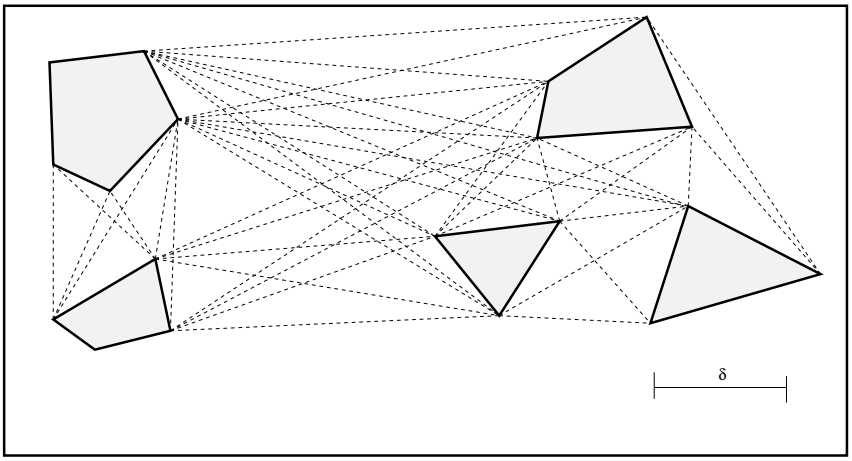


Figure 3.7: Visibility Graph

The time complexity of Algorithm 3.2 can be done as follows. The visibility graph can be computed in $\mathcal{O}(n^2)$ time by using the algorithm reported by Ghosh and Mount [GM91]. Visibility edges having length greater than $\delta$ can be discarded by examining each edges emanating from a vertex of the visibility graph. Hence a reduced visibility graph can be constructed within $\mathcal{O}(n^2)$ time. Furthermore, each visibility edge and the incident obstacle edge can be used to find the possible min-distance edge within the same time complexity. Hence the overall time complexity of Algorithm 3.2 is $\mathcal{O}(n^2)$.

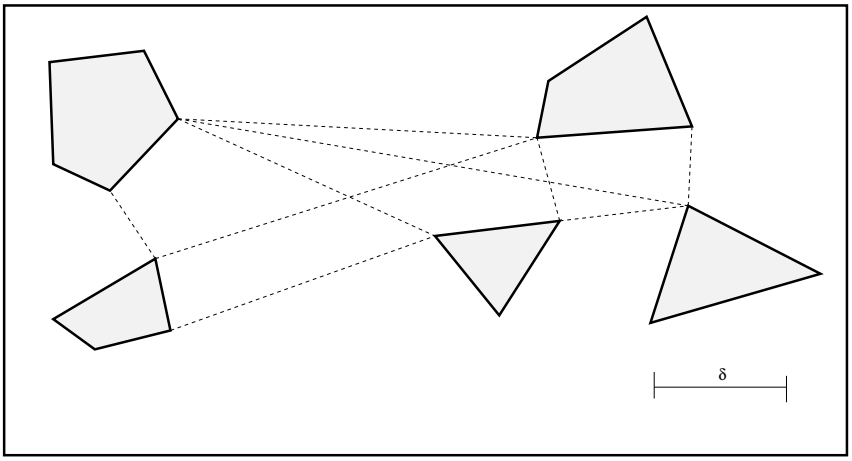


Figure 3.8: Reduced Visibility Graph

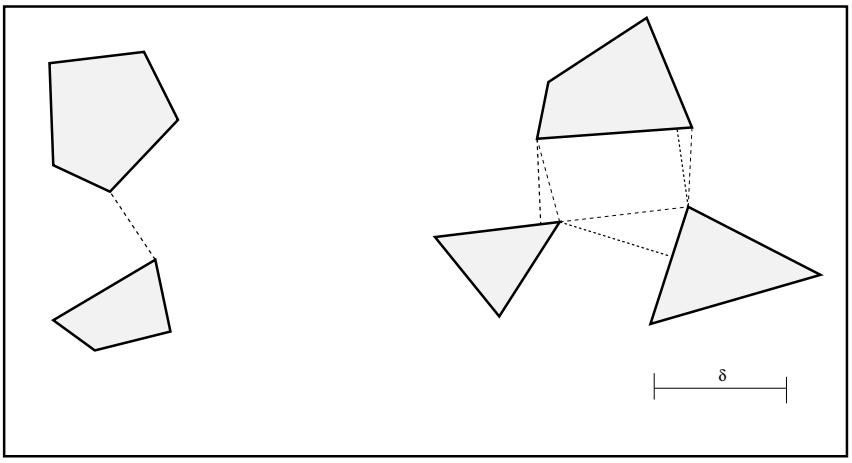


Figure 3.9: Reduced Visibility Graph With Reduced Distance

## 3.4 Adapting *K-means* Approximation

As mentioned in Chapter 2, the problem of clustering points distributed in two dimensions is a well investigated problem and several algorithms for solving this problem have been reported. So we seek to apply point clustering algorithms to develop obstacle clustering algorithms. For such application we need to find ways of converting obstacle distribution into point distribution which is described next.

### 3.4.1 Converting Obstacles to grid points

Based on parameter $\delta$, we can generate a group of points $Pt(Q_i)$ for each obstacle $Q_i$. The conversion should be such that the nearest neighbor of each point in $Pt(Q_i)$ is within distance $\delta$. One approach for generating such points is to consider equally separated parallel lines over obstacle $Q_i$. The line segment inside the

**Algorithm 3.2:** Capturing Obstacle Clusters via Visibility Graph

**Input**: i) Obstacles R = $\{Q_1, Q_2, \ldots, Q_m\}$, ii) Parameter $\delta$
**Output**: Obstacle clusters $C_1, C_2, \ldots, C_T$

```
E = φ
Construct visibility graph G(V, E')
for each vertex v_r in e in V do
    Let e_s be the shortest visibility edge originating from v_r that connects Q_i to Q_j
    if sep(e_s) < δ then
        E = E U {e_s}
Let V^δ = R U E be the reduced visibility graph
Use Depth First Search on V^δ to identify and output each clusters C_1, C_2, ..., C_T
```

obstacle $Q_i$ is partitioned by equidistant points as shown in Figure 3.10

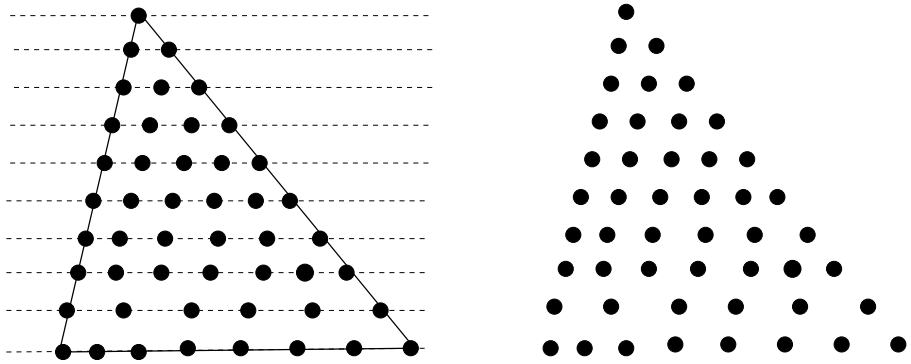

Figure 3.10: Obstacle/Point Conversion

This can be done in a straightforward way by using elementary geometry as follows. We can start with the side of an obstacle triangle as a candidate line segment $l$. Let $d_l$ denote the length of segment $l$. Then we need to generate $K = d_l/\delta$ equally spaced points on $l$. If $(x_1, y_1)$ and $(x_2, y_2)$ are the coordinates of the end points of $l$, then the coordinates of the point that divides $l$ in the ratio $1 : k$ can be computed as follows:

$$x_k = (x_2 + k \times x_1)/(1 + k)$$
$$y_k = (y_2 + k \times y_1)/(1 + k)$$

We can repeat this process of computing $k$ interior points for other parallel line segments to obtain the required set of points.

We can also generate points by considering a smallest enclosing rectangle $R(Q_i)$ for obstacle $Q_i$. We require that $R(Q_i)$ is an isothetic rectangle, i.e. it has edges parallel to coordinate axes. This is illustrated in Figure 3.11

The isothetic rectangle $R(Q_i)$ can be partitioned into a grid whose cells are of size within $\delta$. Given the coordinates of the vertices of $R(Q_i)$, the coordinates of grid points can be calculated easily. Once we

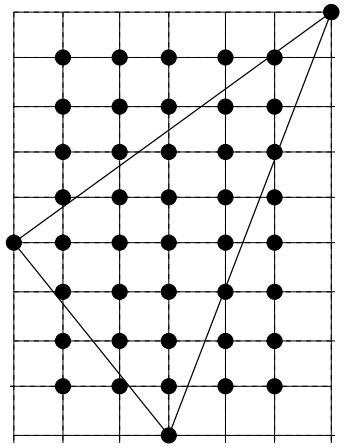

Figure 3.11: Conversion by Isothetic Grid

have the coordinates of the grid points, those that lie within the obstacle can be computed by using the straightforward polygon/point inclusion test. This is listed as *Point Generation Algorithm* (Algorithm 3.3)

**Algorithm 3.3:** Point Generation Algorithm

**Input**: i) Obstacles $Q_1, Q_2, \ldots, Q_m$, ii) Parameter $\delta$
**Output**: Grid points $H$ for all $m$ obstacles

```
1 H = φ
2 for each obstacle Q_i ∈ V do
3 |   Let R(Q_i) be the smallest enclosing isothetic rectangle for Q_i
4 |   Generate grid points G_i by partitioning R(Q_i) into cells of size δ × δ
5 |   for each point q in G_i do
6 |   |   if q is inside Q_i then
7 |   |   |   add q to H
8 return H
```

Remark 3.2: while determining smallest enclosing isothetic rectangle $R(Q_i)$, we make the rectangle slightly larger so that the length and width are each multiples of $\delta$.

Once we convert obstacles into points we can apply *k-means* algorithm to obtain clusters. The cluster of points can be converted back to a cluster of obstacles in a straightforward manner. An example of the application of a grid-point approximation for clustering of obstacles in Figure 3.1 is shown in Figure 3.12 and Figure 3.13. Figure 3.12 shows generated grid points and Figure 3.13 shows the clustering obtained by the *k-means* approach. The obstacle clusters extracted by using the guide line revealed by the point clusters is shown in Figure 3.14. The details of the results on various obstacle clusters is presented in Chapter 4.

While grid-point approximation works pretty well to capture obstacle clusters, it fails in certain kinds of distributions where the standard *k-means* algorithm does not produce good results [GRS]. If we consider

Figure 3.12: Generated Points

Figure 3.13: *k-means* Captured Point Clusters

obstacle distributions that form concentric circles as shown in Figure 3.15, the *k-means* point approximation algorithm fails to capture obstacle clusters. As observed in [GRS], *k-means* algorithm does not work properly for such types of point distributions.

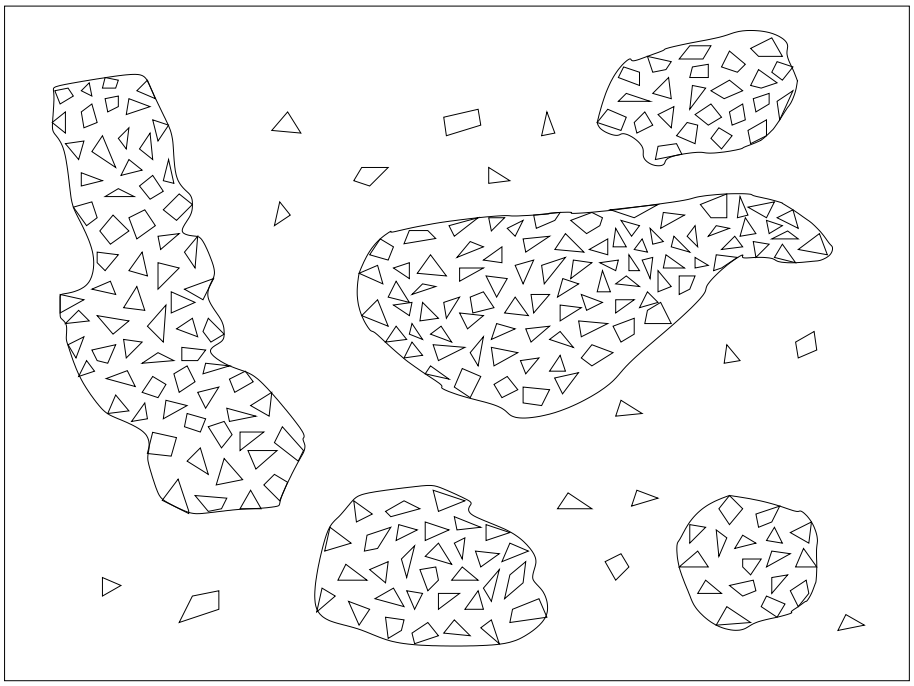

Figure 3.14: Obstacle Clusters captured using *k-means* point-grid approximation

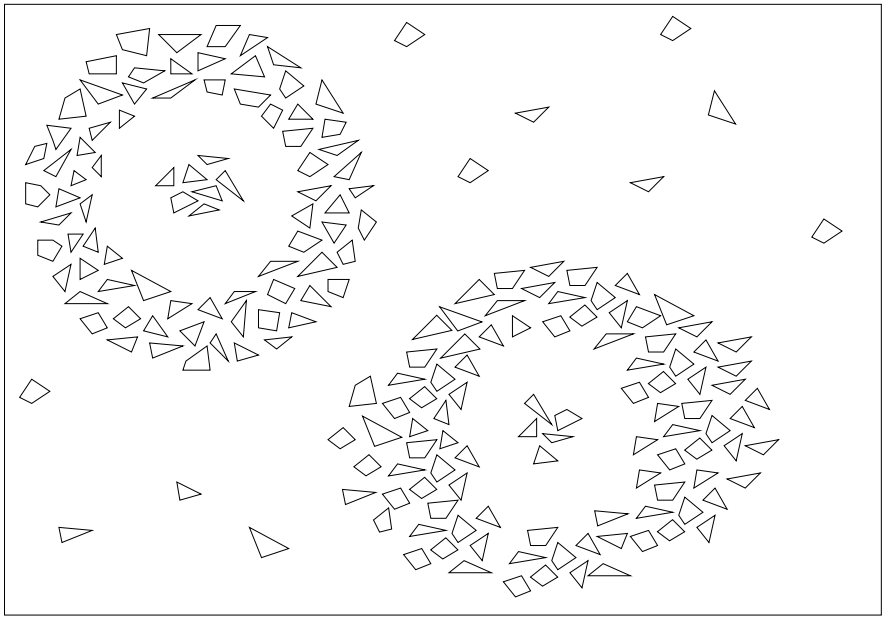

Figure 3.15: Obstacle distribution forming ring like clusters

## 3.5 Voronoi Diagram Based Approach

### 3.5.1 Introduction to Voronoi Diagram (VD)

The Voronoi Diagram (VD) is a partitioning of a plane into regions based on the euclidean distance from some point sites such that any point in a region is nearer to the point site corresponding to the region. In

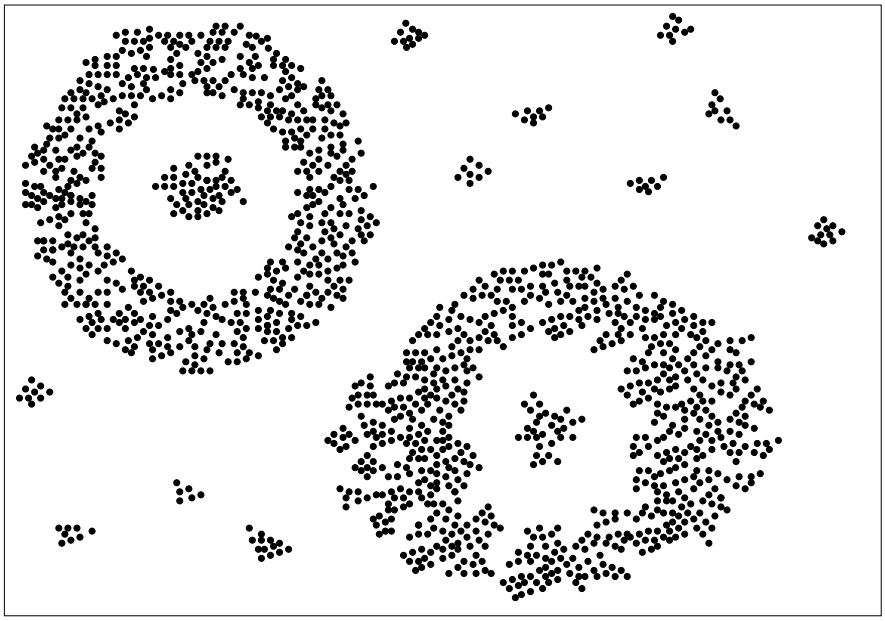

Figure 3.16: Digitization of ring like clusters

other words, VD of $n$ point sites $p_0, p_1, \ldots, p_{n-1}$ partitions the plane into $n$ convex cells $v_0, v_1, \ldots, v_{n-1}$ such that any point in a cell $v_i$ is nearer to $p_i$ than any other site.

Figure 3.17 shows a basic example of VD. Here the small white circles represent the point sites. The partitioning edges which in fact run perpendicularly between a pair of point sites are called Voronoi Edges. The intersection points of the Voronoi edges are called Voronoi vertices.

In VD, the number of Voronoi edges and the number of Voronoi vertices are linearly dependent upon the number of point sites.

## Data Structure for Storing VD

In computational geometry, VD can be represented by a data structure known as a *Doubly Connected Edge List (DECL)*. Figure 3.18 shows the DCEL representation of the VD shown in Figure 3.17.

In DCEL representation of the VD, each Voronoi edge is represented by a pair of twin edges (or half edges) as shown in the figure. We adopt the convention that bounded faces are traversed in a *counter clockwise* direction. In programming, 3 arrays are used to store the complete information of DCEL. These are:

1. Half Edges
2. Faces
3. Vertices

The record for each half edge $e_i$ consists of 5 fields:

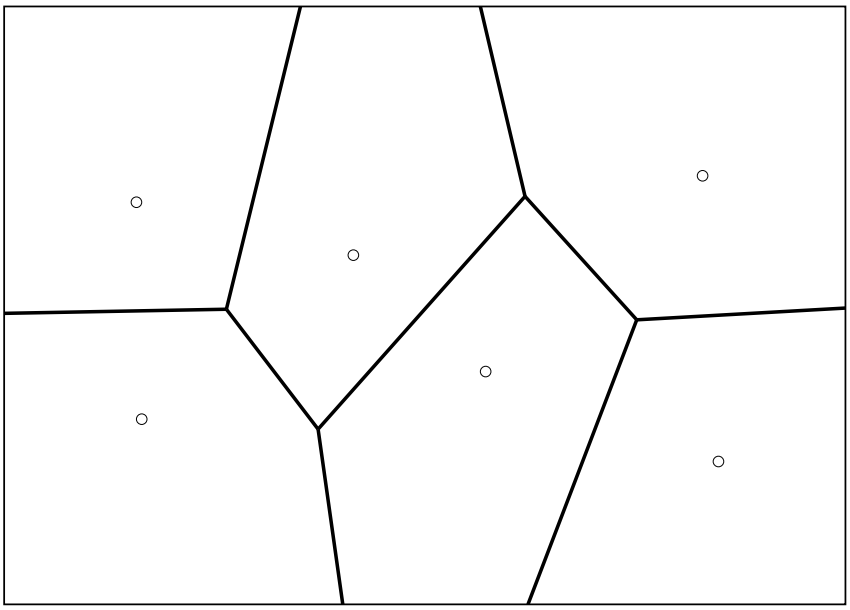

Figure 3.17: Introduction to Voronoi Diagram

1. Twin: twin edge of $e_i$ which is $\overline{e_i}$
2. Prev: previous edge of $e_i$ when face bounded by $e_i$ is traversed
3. Next: next edge of $e_i$ when face bounded by $e_i$ is traversed
4. Face: the face bounded by $e_i$
5. vertex: the vertex on which $e_i$ is incident

The record for each face is defined by any one of the bounding half edge and the record for each vertex is defined by any one of the incident half edges.
Using the DCEL data structure, we can conveniently traverse along the faces of VD.

### 3.5.2 VD Induced by Vertices of Obstacles

The Voronoi Diagram is defined for a set of point sites, not for the objects. As we are attempting to cluster the obstacles, our input is the set of polygonal obstacles. So we can not draw VD directly for our input. But we can treat the vertices of the obstacles as the point sites and therefore draw VD across them. So, in order to draw the VD corresponding to the input obstacles, we need to extract vertices from each obstacle and feed them into the Voronoi Algorithm.

For the given obstacle distribution as shown in Figure 3.19, the Voronoi diagram can be seen in Figure 3.20

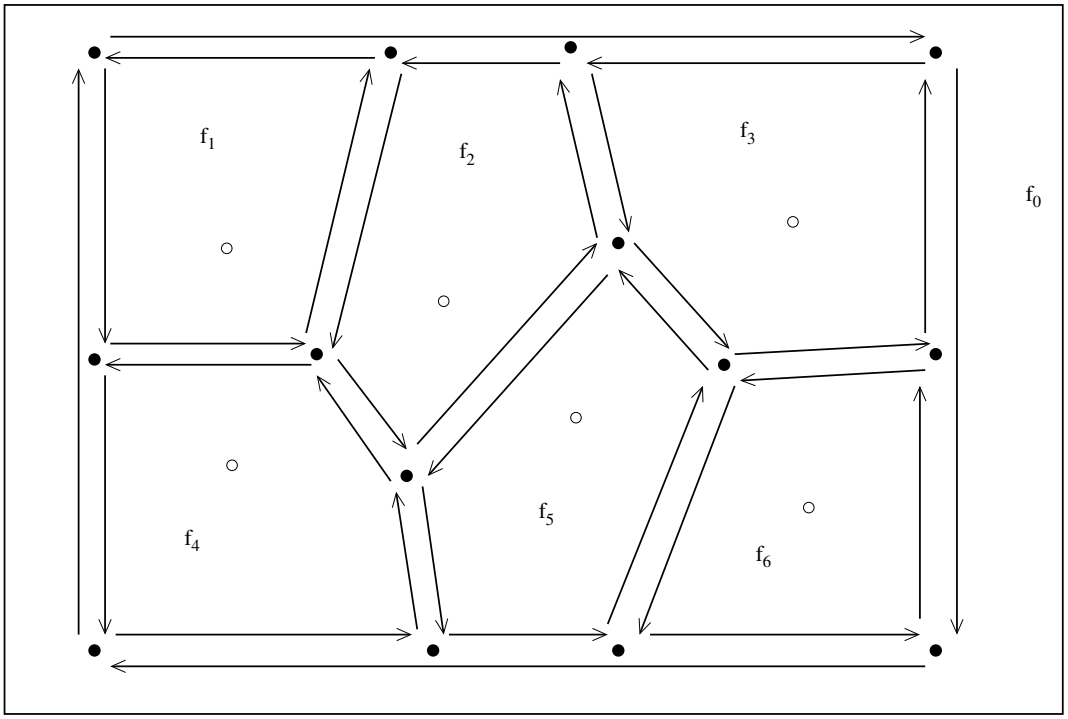


Figure 3.18: Doubly Connected Edge List (DCEL) to represent Voronoi Diagram

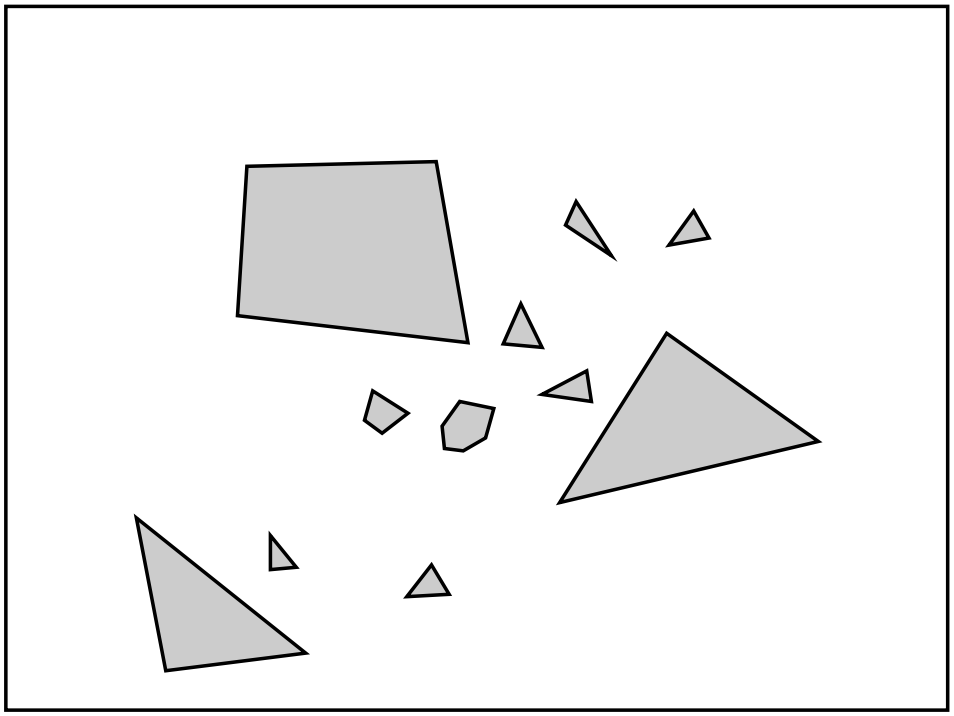

Figure 3.19: Obstacle Distribution

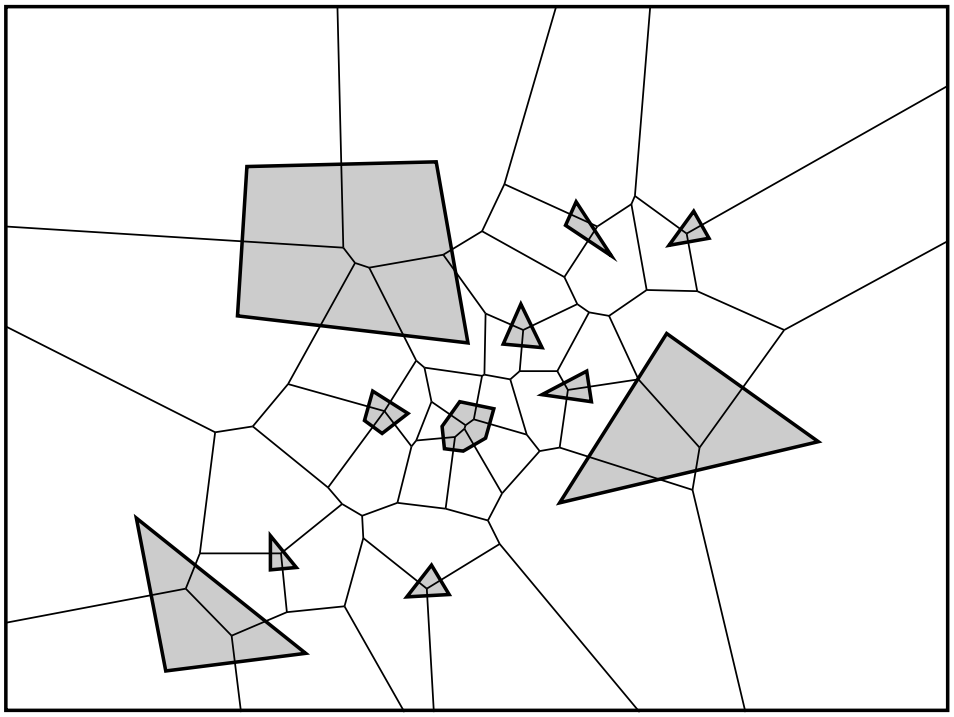

Figure 3.20: Overlay of VD and Obstacles

### Overlay of VD and Obstacles

We can observe that VD induced by the vertices of obstacles (Figure 3.19) also consist of the edges which intersect the obstacle edges. Our approach to cluster the obstacles using this approach is based on the traversing of the VD edges. So it is necessary that before clustering the obstacles, we eliminate those VD edges that intersect with the obstacles. We can then traverse along the non-intersecting VD edges to identify the cluster of obstacles. Elimination of intersecting VD edges is discussed next in detail.

## 3.5.3 Identifying VD-Edges intersecting with Obstacles

As discussed in the previous section, for our primary purpose of clustering the obstacles, we need to eliminate those VD edges which intersect with obstacles to get a modified VD. Let the modified VD be *m-VD*. Identification of the intersecting VD edges is a crucial part of this approach because this process can determine the time complexity of the whole process. We have studied two methods for this process. These are i) Straight Forward Method (Brute Force Checking) and ii) Plane Sweep Method

### Straight Forward Method

This is a simple to implement method that uses brute force checking of every possibility of intersection. This method considers each VD edge one by one and checks with all obstacles whether it intersects with any one

of the edges of any obstacle. If we find that an edge intersects with any obstacle edge, then we ignore this edge from further processing. This brute force method is depicted by the Algorithm 3.4

**Algorithm 3.4:** Intersection detection of Voronoi Edges with Obstacles using Brute Force Method

**Input**: i) Obstacles $Q_1, Q_2, \ldots, Q_m$, ii) Voronoi edges $E_1, E_2, \ldots, E_n$
**Output**: Mark those Voronoi edges $E_i$ which intersect with obstacles

```
1 for each Voronoi edges E_i do
2     for each obstacle Q_i do
3         if E_i intersects with Q_i then
4             mark E_i as an intersecting edge
```

**Time Complexity**

Implementation of this algorithm is very simple. However its time complexity is quadratic. For the given set of $n$ obstacles, the number of VD edges is linear with $n$. So for each VD edge, as we need to check with every obstacle, the average time complexity of the process will be $\mathcal{O}(n^2)$.

## Plane Sweep Method

Plane Sweep is an important technique in the field of Computational Geometry. It actually forms a programming paradigm which can be used to solve several problems in Euclidean Space[O'R98]. Construction of the Voronoi Diagram and identifying intersecting points for the given line segments are examples of application of the Plane Sweep method.

The idea behind algorithms of this type is to imagine that a line (often a vertical line) is swept or moved across the plane, stopping at some points. Geometric operations are restricted to geometric objects that either intersect or are in the immediate vicinity of the sweep line whenever it stops, and the complete solution is available once the line has passed over all objects.

With this *Plane Sweep* method, identification of intersecting points for the given set of line segments can be done efficiently. This technique is what we use to identify the intersecting VD edges.

Consider the overlay of VD and convex polygons as shown in Figure 3.20. Some of the Voronoi edges do intersect with obstacles. To develop a plane sweep algorithm, we use two data structures $R$ and $Y$ to maintain vertices and edges respectively. $R$ is a priority queue which can be used to store vertices in the priority of smallest x-coordinates. Structure $Y$ is a balanced tree that stores edge segments intersected by a sweeping vertical line. The intersecting edges are stored in $Y$ in order of the y-coordinates of the intersection points of the vertical line and the edges. Whenever a Voronoi edge is detected between edges of the same obstacle, that Voronoi edge is marked intersecting. A formal algorithm sketch based on this approach is listed as Algorithm 3.5. This algorithm is described with some cases which occur during the sweeping of the vertical line. During the process, 6 cases can be distinguished. The actions to be taken in these cases are explained next with the help of Figure 3.21

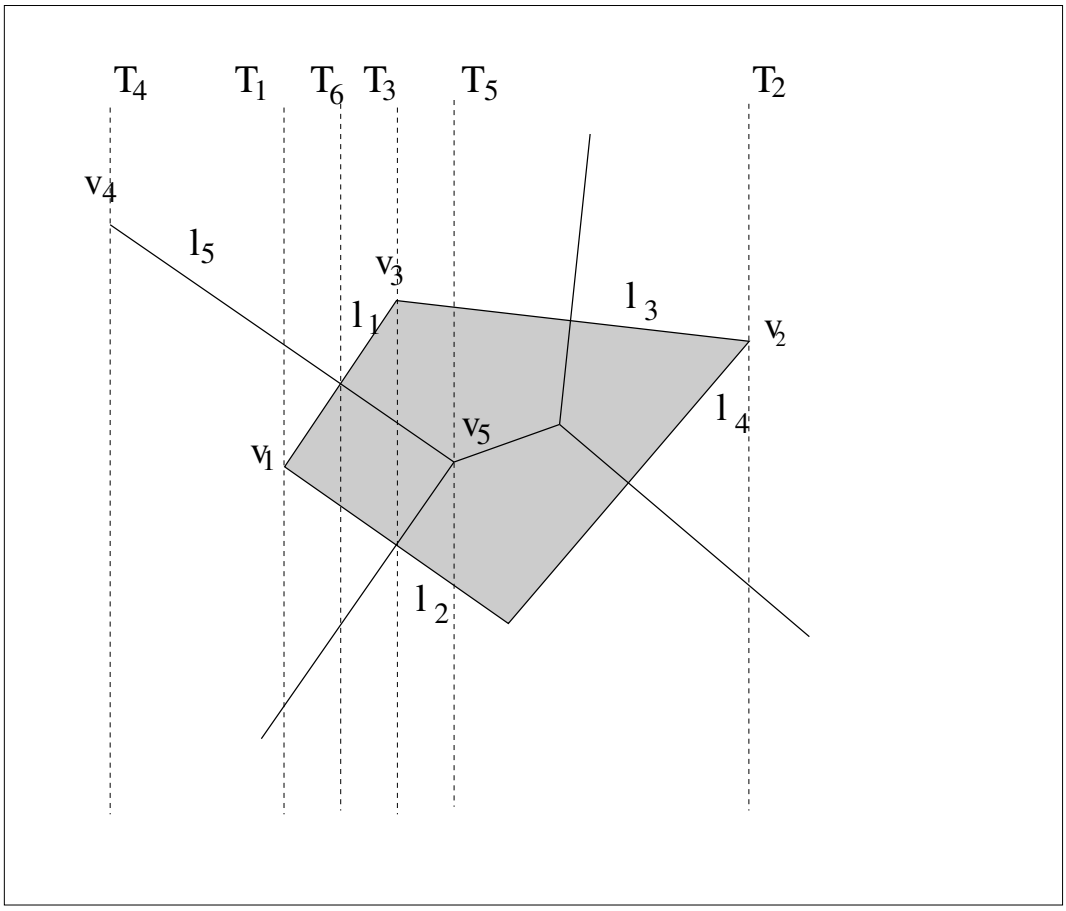


Figure 3.21: Different Cases in Sweep Line Algorithm

For the given obstacle distribution as shown in Figure 3.19, the Voronoi diagram can be seen in Figure 3.20

*Case 1*: Sweep Line $T_1$ is on the left support obstacle vertex $v_1$

- insert the obstacle edges $l_1$ and $l_2$ into the tree $Y$.
- if the obstacle edges intersect with the neighbor Voronoi edges, then insert the intersection vertices into the queue $R$

*Case 2*: Sweep Line $T_2$ is on the right support obstacle vertex $v_2$

- remove the obstacle edges $l_3$ and $l_4$ from the tree $Y$.

*Case 3*: Sweep Line $T_3$ is on the non-supporting obstacle vertex $v_3$

- remove the obstacle edge $l_1$ which lies to the left of the sweep line from the tree $Y$.
- insert the obstacle edge $l_3$ which lies to the right of the sweep line into the tree $Y$

- if the obstacle edge $l_3$ intersects with the neighbor Voronoi edge, then insert the intersection vertices into the queue $R$

*Case 4*: Sweep Line $T_4$ is on the left Voronoi vertex $v_4$

- insert the Voronoi edge $l_4$ into the tree $Y$
- if the Voronoi edge intersects with the neighbor obstacle edge, then insert the intersection vertex into the queue $R$

*Case 5*: Sweep Line $T_5$ is on the right Voronoi vertex $v_5$

- remove the Voronoi edge $l_4$ from the tree $Y$

*Case 6*: Sweep Line $T_6$ is on the intersection vertex $v_5$ of Voronoi edge and obstacle edge

- swap the position of edges $l_4$ and $l_1$ in the tree $Y$.

**Intersection Condition**: During the process of sweeping, if we detect that the Voronoi edge $q_i$ in $Y$ lies between two obstacle edges of the same obstacle, then we can mark the Voronoi edge to be intersecting edge. Let us refer this condition as *Condition 1*.

**Time Complexity**

The complexity of the sweep line algorithm depends upon the number of vertices (vertices of obstacles and end vertices of Voronoi edges) and the number of intersection points. The number of vertices of obstacles and end vertices of Voronoi edges are linearly related to the number of obstacles. By the nature of VD, the number of intersection points of Voronoi edges and obstacles is also linear with the number of obstacles. At each step when the sweep line steps, the balanced search tree $Y$ has insertion and deletion complexity of $\mathcal{O}(\log n)$. So the total time complexity of the sweep line algorithm is $\mathcal{O}(n \log n)$.

For the VD shown in Figure 3.20, *m-VD* is shown in Figure 3.20 where the intersecting VD edges are represented by dashed lines.

### 3.5.4 Voronoi Aided Cluster Extraction

The Voronoi Diagram induced by obstacle vertices can be used to capture obstacle clusters. The idea is to navigate the distribution of obstacles by following the Voronoi edges and examining the size of empty circles at each Voronoi vertex. It is noted that the Voronoi edges intersecting with obstacles (call them *cut edges*) should not be used for navigation of obstacles. The diagram obtained by removing the *cut edges* from the Voronoi Diagram is called the *Reduced Voronoi Diagram*. In Figure 3.22 the edges of the reduced Voronoi diagram are drawn by solid line segments. The edges drawn as dashed line segments are the *cut edges*. The input parameter $\delta$ can be used to identify obstacles that are very close to each other and belong to the

**Algorithm 3.5:** Intersection detection of Voronoi Edges with Obstacles using Sweep Line Algorithm

**Input**: i) Obstacles $Q_1, Q_2, \ldots, Q_k$, ii) Voronoi edges $q_1, q_2, \ldots, q_m$
**Output**: Mark those Voronoi edges $q_i$ which intersect with obstacles

```
Insert vertices of obstacles and Voronoi edges in a priority queue R in priority of smallest
x-coordinates.
Initialize a search tree structure Y to be empty.
while the queue R is not empty do
    delete a vertex v from the queue R
    if Case 1: v is type 1 then
        Insert into Y both edges incident on v
        If the incident edges intersect with neighbor Voronoi edges, insert the intersection vertices into
        R if not already inserted
    if Case 2: v is type 2 then
        Delete from Y both the edges incident on v
    if Case 3: v is type 3 then
        Delete from Y the obstacle edge with v as right end.
        Insert into Y the obstacle edge with v as left end.
        If the obstacle edge with v as left end intersects with any neighbor Voronoi Edge, then insert
        the intersection vertex into R if not already inserted and mark the Voronoi Edge as
        intersecting if not already marked.
    if Case 4: v is type 4 then
        Insert Voronoi edge with v as left end into Y
        If this Voronoi edge intersects with neighbor obstacle edge, insert the intersection vertex into
        R if not already inserted
    if Case 5: v is type 5 then
        Remove the Voronoi edge with v as the right end from Y
        Mark this Voronoi edge as intersecting if Condition 1 is true.
    if Case 6: v is type 6 then
        Swap the position of Voronoi edge and Obstacle edge which intersect at vertex v
        Mark this Voronoi edge as intersecting if Condition 1 is true.
```

same cluster. During the navigation, by following the reduced Voronoi diagram edges, we can examine the size of the empty circles. If the radius of an empty circle is smaller than $\delta$ then the separation between corresponding obstacles is too small and the navigation should back-track from that vertex and proceed. The navigation process can be explained in clearer terms by referring to the doubly connected edge list representation (*DECL* representation) of the reduced Voronoi diagram.

Figure 3.23 shows the general structure of the reduced Voronoi diagram surrounding an obstacle cluster.

Figure 3.24 shows the DCEL representation of the diagram shown in Figure 3.23. This diagram can be traversed in a counterclockwise direction by following the half-edges of the DCEL structure. During the traversal, the empty circle at each vertex is examined and its radius is compared with $\delta$ to determine the clearance. If the clearance radius is smaller than $\delta$ then the navigation proceeds by following the twin half edge. During the clearance check, the obstacles in the left side of the traversal are marked as boundary obstacles belonging to the same group. This process of i) clearance check ii) forward movement if possible iii) twin flip is continued until the start point is encountered again to complete a cycle. The obstacles marked

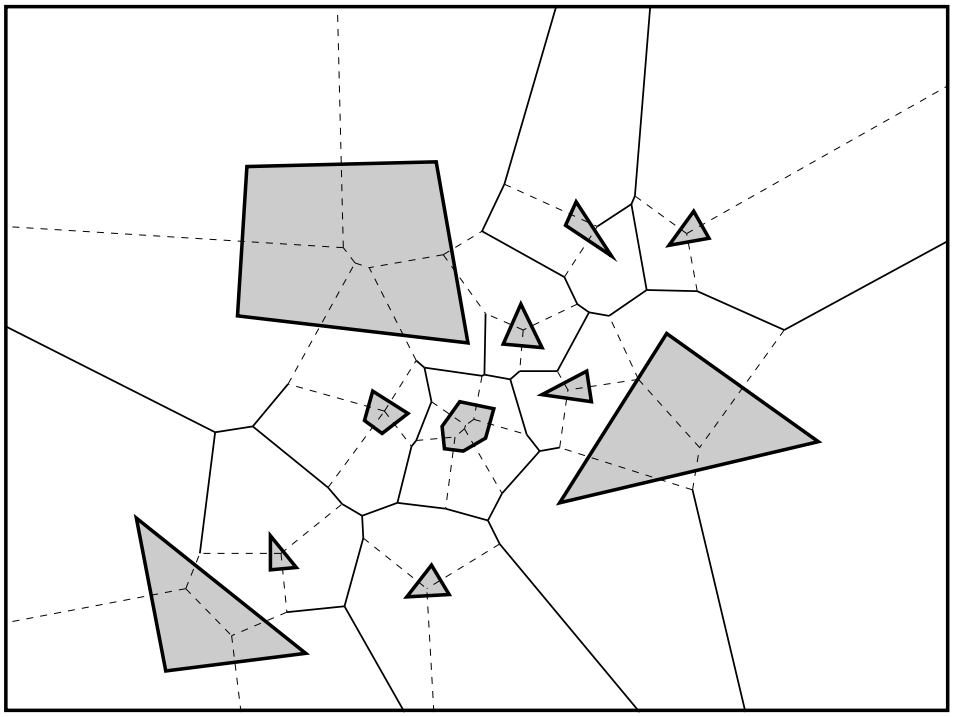

Figure 3.22: Reduced Voronoi Diagram and Cut-Edges

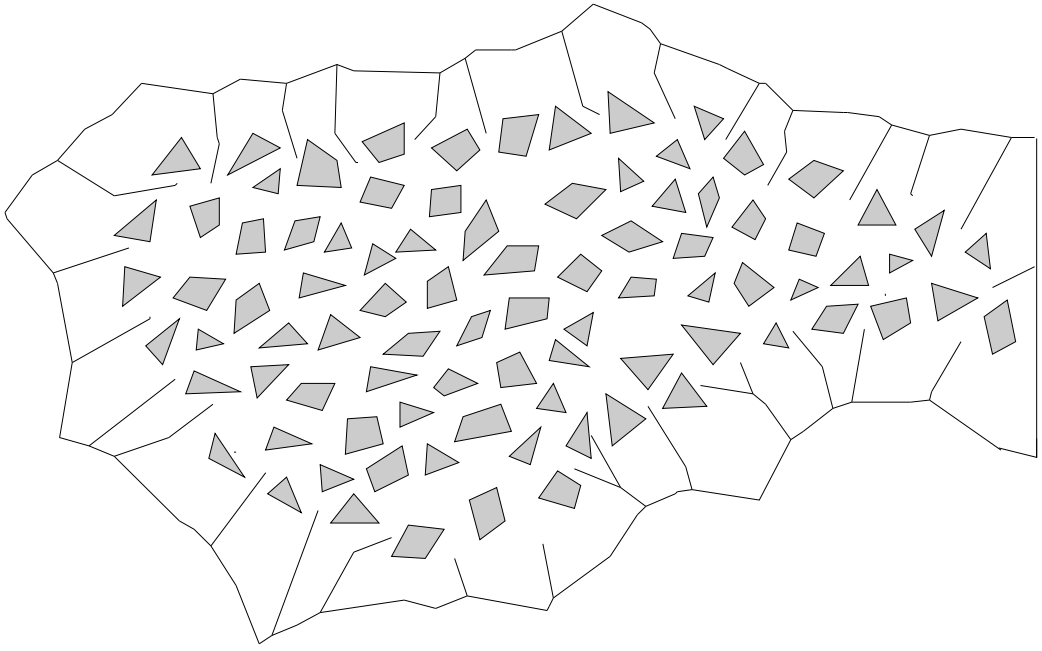

Figure 3.23: General Structure of Reduced Voronoi Diagram

during this cycle traversal are the boundary obstacles belonging to the same cluster.

A formal sketch of the algorithm is listed as Algorithm 3.6. In this algorithm, the sketch is described for

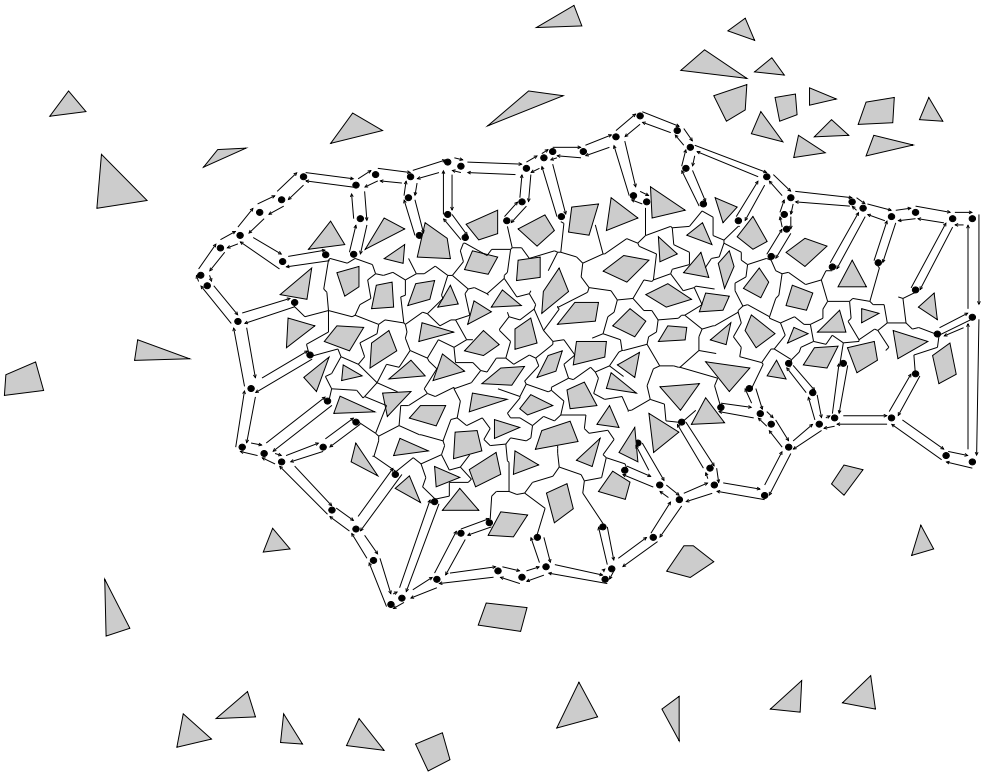

Figure 3.24: DCEL Representation of Reduced Voronoi Diagram

identifying the cluster corresponding to the starting face. But we can repeat to identify all clusters. Also in this algorithm, the method *Clearance(vertex v)* returns the radius of the empty circle at the Voronoi vertex $v$.

**Time Complexity**

The complexity of the Voronoi Aided Cluster Extraction Algorithm 3.6 depends on how we construct the VD and how we detect the intersecting Voronoi edges. VD construction in step 1 can be achieved in time $\mathcal{O}(n \log n)$ by using the *Fortune's Sweep-Line Algorithm*. Step 2 uses Algorithm 3.5 to identify the cut-edges. This process also takes $\mathcal{O}(n \log n)$ time. The *while* loop in the step 4 visits the Voronoi edges in the reduced Voronoi diagram for a constant number of times. Therefore the time complexity of the loop is $\mathcal{O}(n)$. So the overall time complexity of the algorithm is $\mathcal{O}(n \log n)$.

**Algorithm 3.6:** Voronoi Aided Cluster Extraction

**Input**: i) A collection of obstacles $Q_1, Q_2, \ldots, Q_m$, ii) Threshold parameter $\delta$
**Output**: Extraction of Obstacle Clusters

Obtain Voronoi diagram $VD$ of the vertices of obstacles $Q_1, Q_2, \ldots, Q_m$
Use plane-sweep algorithm (Algorithm 3.5) to determine and delete/mark cut-edges of the VD. Let $VD^r$ be the reduced Voronoi diagram in DCEL representation.
Let $e_1$ be a half edge of a face of $VD^r$, $L_1 = \phi$, eStart = $e_1$, $e_1 = e_1.next$
**while** *($e_1 \neq eStart$)* **do**
    $\alpha$ = Clearance($e_1.start$)
    **if** $\alpha < \delta$ **then**
        mark obstacles to the left of $e_1$ and insert them to $L_1$
        $e_1 = e_1.twin$
    **else**
        $e_1 = e_1.next$
Output $L_1$

# Chapter 4

# Implementation

This chapter presents the implementation details of the selected algorithms proposed/reviewed in Chapter 2 and Chapter 3. The implementation was done in C++ programming language. For efficient implementation of common geometric algorithms, the *CGAL (Computational Geometry Algorithms Library)* is used extensively. Using the geometric computational tools available in *CGAL*, the proposed clustering algorithmic methods, including *Nearest Neighbor Approach*, *K-means Clustering* and *Voronoi Diagram Approach*, have been implemented. For implementation of the Graphical User Interface (GUI), the *Qt* programming framework (*http://www.qt.io/*) was used.

## 4.1 Implementation Description

*Qt* is a cross platform development framework for Desktop and Mobile Applications. Qt is not a programming language itself, but a framework written in C++ to develop GUI applications. Qt has extended some features of other libraries like *OpenGL* (*https://www.opengl.org/*). One example of such a feature is *QGLWidget* which we have used in this project.

*CGAL* is a software project that provides easy access to efficient and reliable geometric algorithms in the form of a C++ library. CGAL is used in various areas needing geometric computation, such as geographic information systems, computer aided design, molecular biology, medical imaging, computer graphics, and robotics. CGAL provides several algorithms which can be efficiently used. Intersection detection of polygons, Distance between polygons and Voronoi Diagram creation are some of the algorithms that are included in CGAL and adopted in our implementation.

CGAL APIs are available only in C++. For this reason, we have done our implementation in C++. We have used the Qt framework to develop the GUI, and CGAL to do the back-end processing of program input/output.

We have modeled obstacles by convex polygons. Obstacle polygons are represented by keeping records of the coordinates of their vertices. The vertices are stored in an anticlockwise direction as they occur along the boundary of the polygons. The implemented algorithms use these representations to obtain intended outputs. The implementation of the *k-means Clustering Approach* is achieved by only considering the distribution of the vertices of the obstacles. In the *Voronoi Diagram* based clustering algorithm, cluster identification is done by considering (i) the Voronoi diagram of obstacle vertices and (ii) the intersection of obstacle edges with Voronoi edges. Finally, in the *Shortest Distance Approach*, the clustering of obstacles is implemented by computing the shortest distance between neighboring polygons directly.

## 4.2 Data Structures

The main data structures to model and implement the structure and distribution of obstacles was done by designing a set of C++ classes described next.

### CustomPoint

The class *CustomPoint* is used to model the vertices in the 2D. We have implemented this class by extending the *Polygon* class from *CGAL*. It describes the x- and y- coordinates of a vertex. The class interface diagram for *CustomPoint* is shown in the Table 4.1.

| $<< interface >>$<br>`CustomPoint` |
|---|
| - x : double<br>- y : double |
| + CustomPoint(double x, double y) |

Table 4.1: Class Interface Diagram of CustomPoint

### CustomLine

The class *CustomLine* describes any straight line segment in 2D plane connecting two given vertices. This class stores the two end vertices as an instance of the class *CGAL::Point*. It also stores the squared distance between these two vertices. We decided to store squared distance, rather than distance, to speed up execution time as square root is an expensive operation. The class diagram for this class is shown in the Table 4.2.

| |
|---|
| $<< interface >>$<br>`CustomLine` |
| - p : CustomPoint<br>- q : CustomPoint<br>- length : double |
| + CustomeLine(Point p, Point q)<br>+ CustomeLine(Point p, Point q, double length)<br>+ getP() : Point<br>+ getQ() : Point<br>+ getSquaredDistance() : double |

Table 4.2: Class Interface Diagram of CustomLine

### CustomPolygon

The class *CustomPolygon* represents a polygon in the 2D plane modeled as an obstacle. It has been implemented by extending the *CGAL::Polygon* class. It stores the vertices of the polygon in an anticlockwise direction as they occur along the boundary. In addition to storing the vertices, some additional helper functions are considered. For example, (i) the function *computeDistance(Polygon p)* returns the minimum distance between the candidate polygon object and argument polygon $p$, where the returned minimum distance is an instance of *CustomLine*; (ii) the function *doIntersect(Polygon p)* detects whether the candidate polygon intersects with the argument polygon $p$; and (iii) the function *hasVertex(double x, double y)* checks whether the candidate polygon has the vertex with the coordinates *(x, y)*. These helper functions have been frequently used in our implementations. The class interface diagram for *CustomPolygon* is shown in the Table 4.3.

### QList

This is a built-in *List* data structure from the *CGAL*. We have used this data structure to store the sequence of Vertices, Line Segments and Polygons wherever necessary.

These are the basic data structures used to model obstacles. Based on these data structures, we have applied algorithms like *Shortest Distance Approach*, *k-means Approach* and *Voronoi Diagram Approach*. Implementation of these approaches are explained in the Section 4.4, Section 4.5 and Section 4.6, respectively.

| |
|---|
| << *interface* >><br>`CustomPolygon` |
| - vertices : QList<CustomPoint><br>+ isSelected : bool |
| + CustomPolygon()<br>+ CustomeLine* computeDistance(CustomPolygon p) : CustomLine*<br>+ bool doIntersect(CustomPolygon p) : bool<br>+ CustomPoint* getVertices() : CustomPoint*<br>+ int hasVertex(double x, double y) : int |

Table 4.3: Class Interface Diagram of CustomPolygon

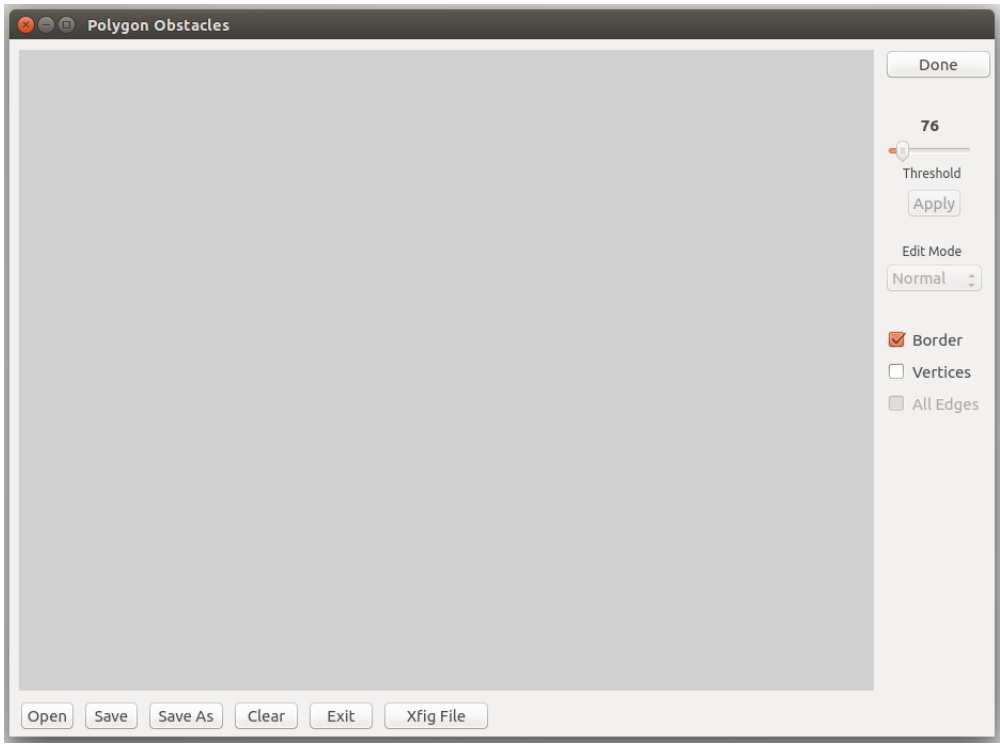


Figure 4.1: The Greeting Interface of the Program

## 4.3 Program Interface Description

As mentioned in the previous section, the GUI has been created using Qt. Figure 4.1 shows the main greeting interface of the program. The components like Buttons, Sliders, Drop Down Boxes and Check-boxes are the native components provided by the Qt Framework. The large gray rectangular area, which we call *Stage*,

is where the user can draw the polygons to model obstacles. This rectangular area is also a Qt component known as *QGLWidget*. However this is not a native component of Qt. It is built on the top of *GLWidget* which is a component of the *OpenGL* library.
The *Create* button is used for drawing polygons on the stage. Once the *Create* button is clicked, the program stays in *Create Mode*. In *Create Mode* the user can draw polygons on the *Stage*. Also, in this mode, the *Create* button changes to a *Done* button which can be clicked to exit the *Create Mode*. We can draw polygons by clicking on the *Stage*. A sequence of mouse clicks creates a series of vertices. When the user clicks back near the first vertex, it completes one polygon. One can repeat this process to draw the desired number of polygons. To exit the *Create Mode*, it is necessary to click the *Done* button.
The slider *Threshold* defines the threshold value denoted by $\delta$. This value will determine whether two or more polygons belong to the same group. This slider can select from the range of values between 10 and 500 pixels, inclusive.
The drop down menu *Edit Mode* defines the mode for constructed polygons. It defines five different modes. These are:

- Normal Mode
- Edit Mode
- Add Vertex Mode
- Delete Vertex Mode
- Move Polygon Mode

In *Normal Mode*, we cannot make any changes to the polygons. All we can do is highlight selected polygons. In *Edit Mode*, we can move the vertices of a selected polygon to change its shape. *Add Vertex Mode* and *Delete Vertex Mode*, as their names imply, are used for adding and deleting the vertices, respectively. Similarly, the *Move Polygon Mode* is used for moving the location of a polygon in the *Stage*.
In addition there are three check-boxes in the right panel. These are

- Border-D
- Vertices-D
- Edges-D

The *Border-D* check box is used for appearance/disappearance of polygons. While enabled, the *Border-D* check-box displays the boundaries of all polygons; disabling it hides them. *Vertices-D* is used for a similar function. However, in addition to hiding and showing vertices, there are also some background tasks going on with the display of vertices. Specifically there are two different background tasks executed at different modes, *k-means* and *Voronoi Diagram* modes. It can be used to toggle between i) display of generated obstacle

vertices and ii) display of only the obstacle boundary. In case of *k-means approach*, when generated vertices are displayed, a background process for computing clusters based on *k-means* algorithm is triggered. In the case of *Voronoi Diagram Approach*, a background process to compute the Voronoi diagram is executed. The *Edges-D* check-box is functional only with the *Voronoi Diagram Approach*. What it does is, upon enabling, it displays all the Voronoi Edges, while disabling it will hide all the Voronoi Edges that intersect with any one of the obstacle boundaries.

The bottom panel contains some buttons for I/O related tasks. *Open* button lets the user select a file from the disk in a pop-up window and read the polygons data saved in the file. *Save* button can be used to save the current configurations of the polygons so that it can be read later from the disk. *Save As* button is used for assigning a new file name. *Clear* button erases all the polygons data from the *stage*. *Exit* button will exit the program. The *XFig File* button can be used to export the current configuration into an *XFig* format file, so that it can be worked into a *XFig* program, when necessary.

## 4.4 Implementation of Shortest Distance Approach

This is the implementation of the algorithm described in the Section 3.2. The obstacles are modeled as convex polygons which are implemented as the class *CustomPolygon* by extending the features of *CGAL::Polygon_2*. This class is used to calculate the shortest distance between selected pair of polygons. To calculate the distance we have used *CGAL* function, *Polytope_distance::Coordinate_iterator*. The distance is taken into consideration if the line segment corresponding to the shortest separation does not intersect any other obstacle. Detection of the intersection of a line segment with a polygon is also done using the CGAL function *CGAL::do_intersect(T1, T2)*. The distances which are longer than the threshold value $\delta$ are also ignored. Once distances between desired pairs of obstacles are found, clustered obstacles are identified. For the obstacle distributions shown in Figure. 4.2, edge segments corresponding to shortest distances are drawn red in Figure 4.3.

Obstacles in a connected component (based on $\delta$-max distance) form a cluster. To implement the grouping of obstacles into a single cluster, we have used the Union-Find Data Structure [CLRS09]. As an input to the Union-Find, we sequentially feed the indices of polygons connected by the shortest distance. At the end Union-Find will have the connected components of the polygons which we treat as the separate clusters. Now each cluster can be treated as a single obstacle wrapped by a convex hull. To compute the convex hull, we have used the CGAL function *CGAL::convex_hull_2()*. This function requires input vertices. The input vertices are the vertices of all polygons in the cluster. Wrapped clusters are shown in Figure 4.4.

## 4.5 Implementation of k-means Approach

We have implemented the *k-means* approach by adopting the Algorithm 2.3 briefly described in Chapter 3. This algorithm describes the clustering process for the input vertices. It accepts a collection of vertices and

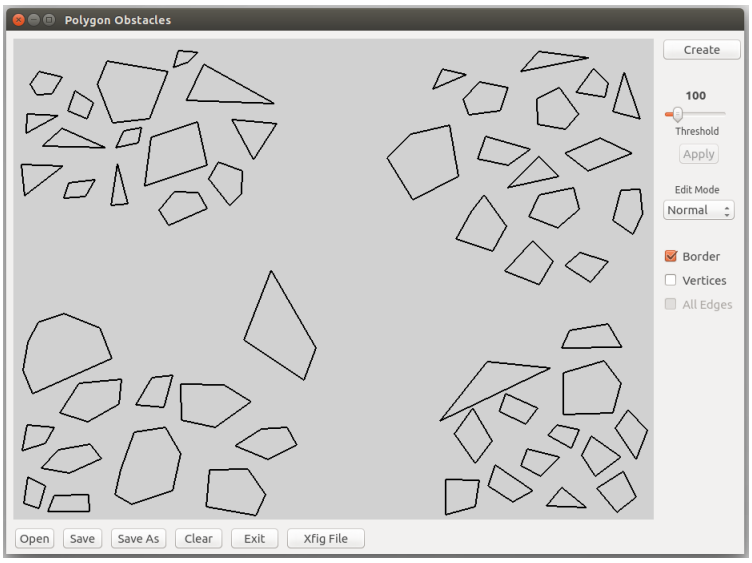


Figure 4.2: Obstacle Distribution

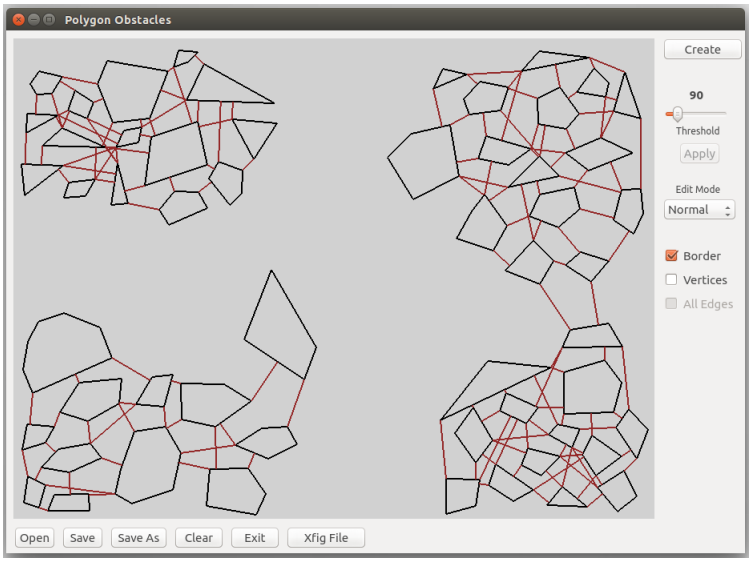


Figure 4.3: Obstacles Connected by Shortest Distances

clusters them based on the *Euclidean Distance*. To execute the *k-means* algorithm on selected obstacles, we need to convert them into points. Specifically, the conversion is done by using a grid mask as described in Chapter 3. This process is explained in detail in Section 3.4.1 and Algorithm 3.3. Once we have the grid points, we feed them to Algorithm 2.3 as input to obtain clusters.

For the obstacle distribution shown in Figure 4.2, the converted grid points are displayed in Figure 4.5 and the subsequent clustering is depicted in Figure 4.6.

The most basic version of *k-means* algorithm is used in our implementation. For better results improved

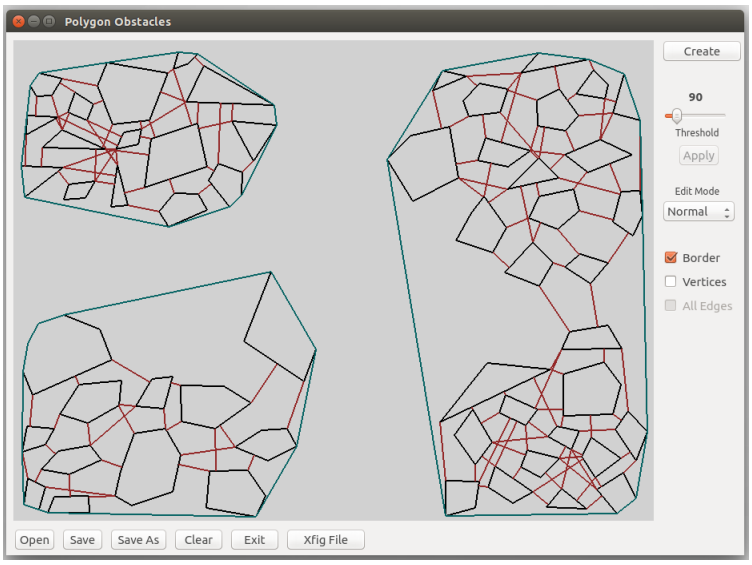


Figure 4.4: Obstacles Clustered by Convex Hull

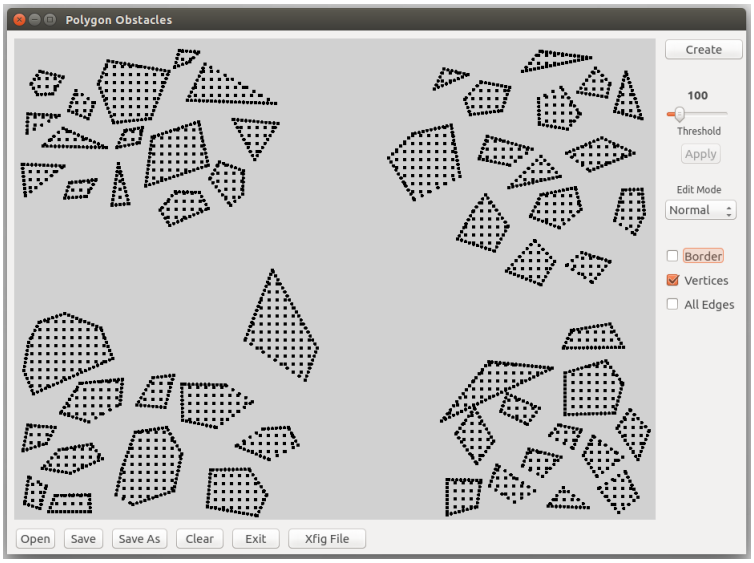


Figure 4.5: Grid Points of Obstacles

versions can be adopted.

## 4.6 Implementation of Voronoi Diagram Approach

We have implemented Voronoi diagrams with the help of CGAL. There are published algorithms for computing Voronoi Diagrams of polygons. The implementation of these algorithms are very complex and not available in the public domain. So we opted to approximate the Voronoi diagram of polygons by using the Voronoi diagram of selected points on the polygons.

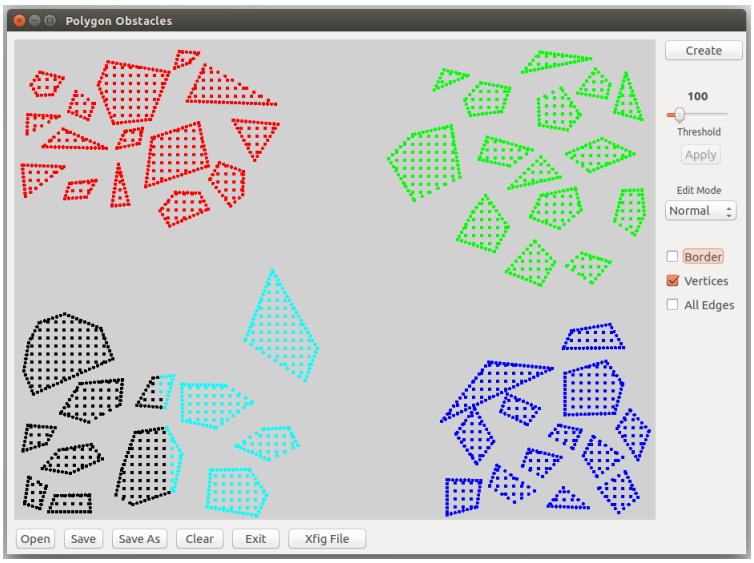


Figure 4.6: K-means Clustering of Grid Points of Obstacles

We have created a separate utility class VDUtil to handle the computation of Voronoi diagrams. In this class *construct(QList<PointToCluster> vertices, bool constructWithoutIntersectingEdges)* is the main utility function to compute Voronoi diagrams. It takes a list of vertices as the input, whose Voronoi diagram can then be computed. It also takes a Boolean variable constructWithoutIntersectingEdges. If it is true, it will compute two separate set of edges. One set will contain the Voronoi edges which intersect with any one of the input obstacles and the other contains the non-intersecting ones. This *construct()* function in turn invokes the helper function from *CGAL* which actually computes the Voronoi diagram. For the given obstacle distribution shown in Figure. 4.2, the Voronoi Diagram is shown in Figure 4.7.

The VDUtil class maintains two lists *i) voronoiLineSegments* and *ii) intersectingVoronoiLineSegments*. If the *construct()* method is computing the Voronoi diagram without identifying intersecting edges, then *voronoiLineSegments* stores all the Voronoi edges while *intersectingVoronoiLineSegments* remains empty. If that is not the case then *voronoiLineSegments* stores only the non-intersecting edges and *intersectingVoronoiLineSegments* stores all intersecting edges.
VDUtil class has functions to distinguish Voronoi edges that intersect with obstacles by using a helper function *bool doesIntersect(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4)*. This functions accepts the coordinates of four vertices representing the end points of two line segments. It determines whether the two line segments intersect or not based upon the calculation of triangle areas.
Figure 4.7 shows all the Voronoi edges, both the intersecting and non intersecting. In Figure 4.8, non-intersecting and intersecting (with obstacles) Voronoi edges are drawn differently for easier visual recognition. While the intersecting edges are drawn as dashed line segments, the non-intersecting ones are displayed as the solid lines. For further clarification, Figure 4.9 shows only the non-intersecting Voronoi edges.

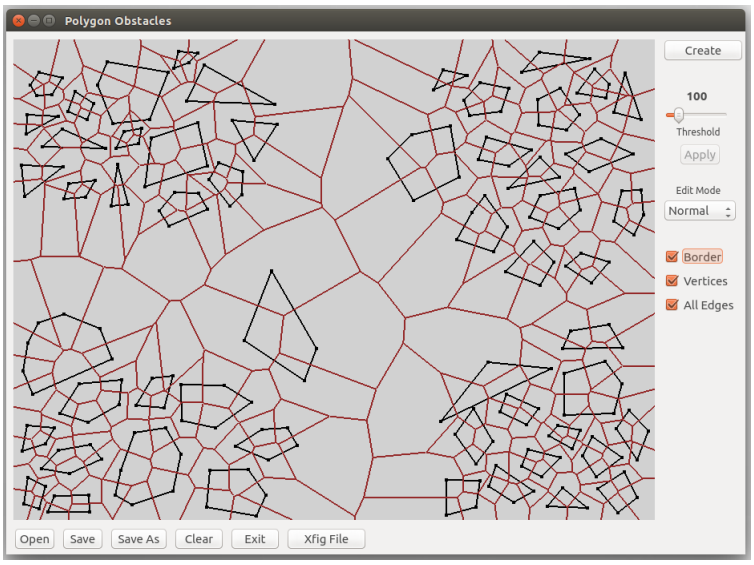


Figure 4.7: Voronoi Diagram

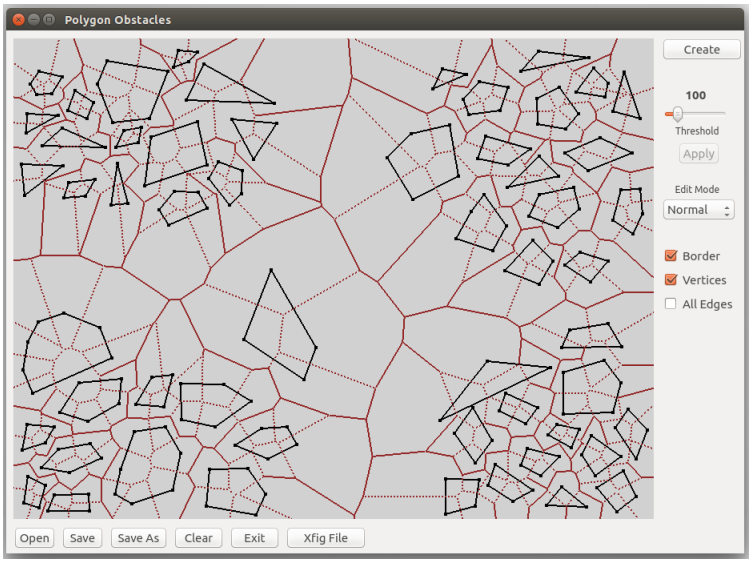


Figure 4.8: Voronoi Diagram showing distinct sets of intersecting and non-intersecting Voronoi Edges

As already explained, the Voronoi diagram is computed for vertices only. It has nothing to do with the polygon edges. Polygon edges are used only to identify the intersecting Voronoi edges. Figure 4.10 shows the Voronoi diagram of obstacle vertices without displaying obstacle edges.

## 4.7 Benefits After Clustering the Obstacles

So far we have discussed about how we can cluster the given collection of obstacles. The objective of obstacle clustering is to ease the path planning process in the presence of obstacles. If we start the path planning

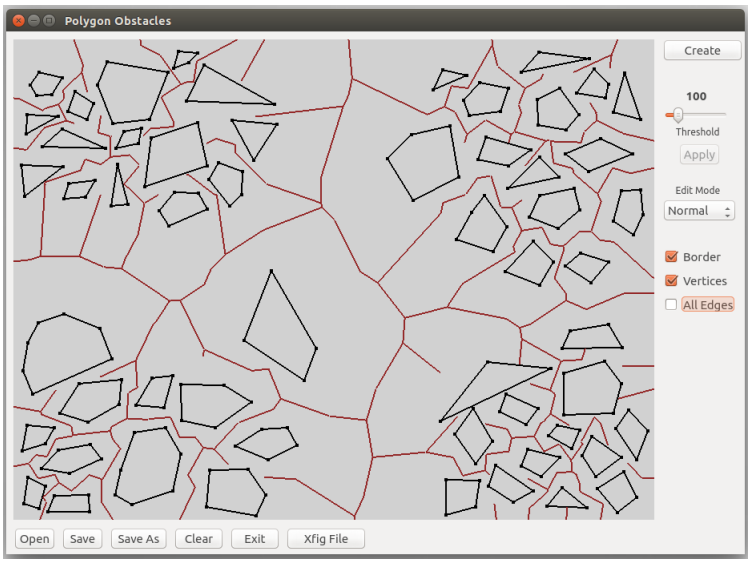


Figure 4.9: Voronoi Diagram showing only the non-intersecting Voronoi Edges

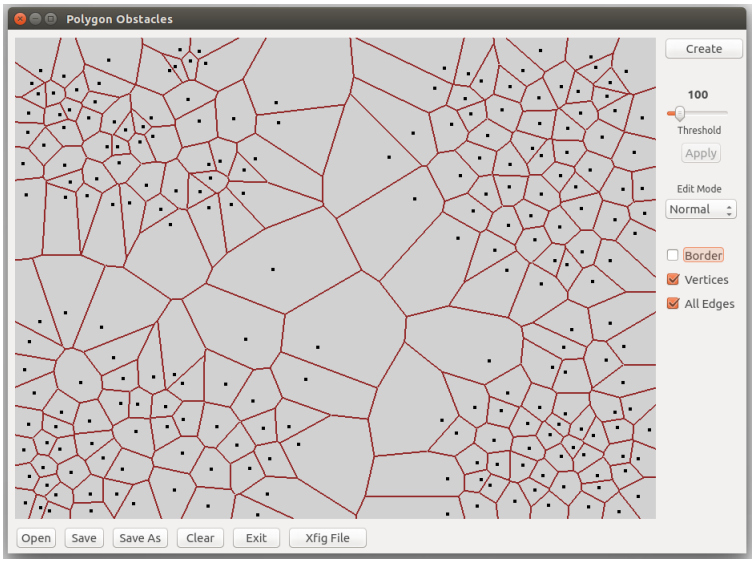


Figure 4.10: Voronoi Diagram with borders of obstacles hidden

process without clustering the obstacles, then we need to consider every single obstacles to consider a path. To consider a path, we will need to check for the threshold distance $\delta$ from each of the individual obstacles. This will lead to a very expensive computation. Our primary idea is to simplify the path planning process and to reduce the number of obstacles so that we need to check a fewer number of obstacles to consider a path. So the idea of clustering the obstacles is inspired by this objective.

Table 4.4 shows some results relevant to this. We can see that, after clustering of the obstacles, the number of vertices and edges are significantly reduced. For the given distribution of obstacles in Figure 3.1

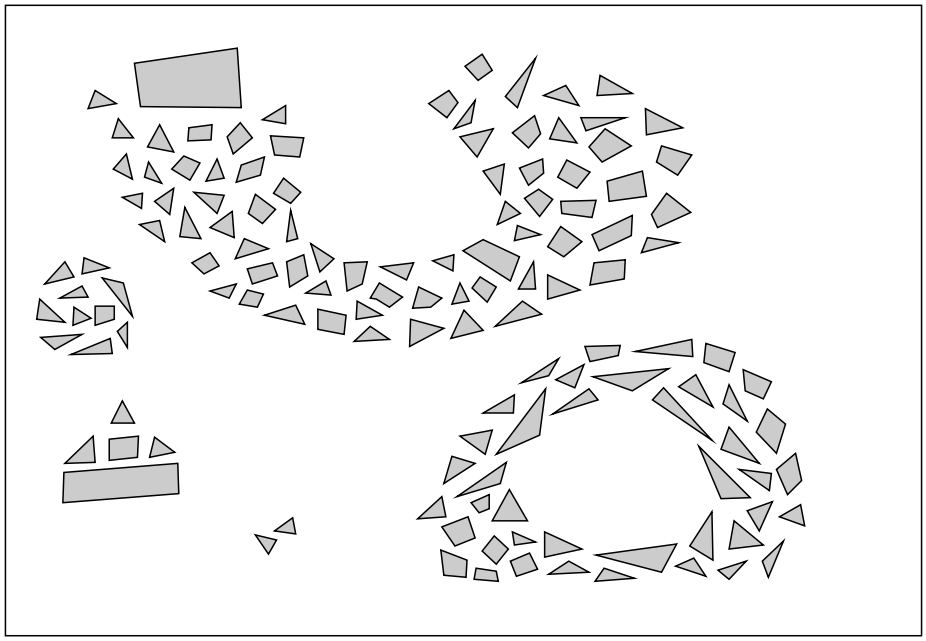

Figure 4.11: Obstacle Distribution

and Figure 3.15, the number has been reduced by a factor of about 10. Similarly for the distribution shown in Figure 4.11, the reduction factor is around 5. With such a reduction in number, the complexity of the computation of paths will be improved significantly.

Table 4.4: Improved Number of Vertices and Edges as a result of Obstacle Clustering

| | Figure 3.1 | | Figure 4.11 | | Figure 3.15 | |
|---|---|---|---|---|---|---|
| | Before Clustering | After Clustering | Before Clustering | After Clustering | Before Clustering | After Clustering |
| $\|E\|$ | 1026 | 142 | 571 | 97 | 863 | 113 |
| $\|V\|$ | 1026 | 142 | 571 | 97 | 863 | 113 |

# Chapter 5

# Conclusion and Discussion

The key idea that inspired us to write this thesis is that clustering polygonal obstacles can simplify the path planning problem.

We examined the possibility of extending the *k-means* algorithm to cluster polygonal obstacles. We presented a brief overview of one of the most cited methods for clustering point objects called *k-means algorithm*.

The first extension called *masking* is obtained by simply converting obstacles into points by using a *grid masking*. This method works to a certain extent, but produces poor results for certain point distributions where the regular *k-means* method fails. It would be interesting to explore further insight into removing shortcomings of the masking method.

We explained how the *Visibility Graph* induced by polygonal obstacles can be used to design a faster version of obstacle clustering. We showed that visibility edges can be examined locally to obtain closest obstacle pairs. This approach exploits faster Visibility Graph construction algorithms to obtain closest obstacles efficiently.

The direct separation method is rather 'brute force' as it computes all pairs of closest obstacles by exhaustive check and is very time consuming.

We presented how the *Voronoi diagram* induced by the vertices of the obstacles can be used to capture the boundaries of obstacle clusters. We exploited the clearance property of the *Voronoi diagram* to realize this. Our algorithm was developed by following the clearance corridors implied by Voronoi edges. Only those edges of the Voronoi diagram are used that correspond to a clearance of more than the predetermined value $\delta$. We did not implement the clustering algorithm based on the Voronoi diagram due to lack of a time. It would be interesting to investigate the performance of the proposed Voronoi-based obstacle clustering algorithms on several randomly generated obstacle distributions. Random generation of convex obstacles is itself an interesting problem worth further investigation.

In order to understand the effectiveness of an obstacle clustering algorithm, as a pre-processing step in path planning, it would be necessary to count the number of edges in obstacles before and after the application of clustering. We did obtain some results as reported in Chapter 4, Table 4.4. This investigation should be

extended on convex obstacle distributions obtained by random generation.

In this thesis, we have represented obstacles by convex polygons. In real life, an obstacle can also by non-convex. In such cases, the algorithms we have presented can still be usable, but with some modifications. For example one modification could be to convert the non-convex polygons into convex by computing the *convex hull* for each of the non-convex polygons. The lower bound for the computation of the convex hull is $\mathcal{O}(n \log n)$. So this conversion does not effect the complexity of the Voronoi approach which is the best we have. Convex polygons and their corresponding Convex Hull representation is shown in Figure 5.1.

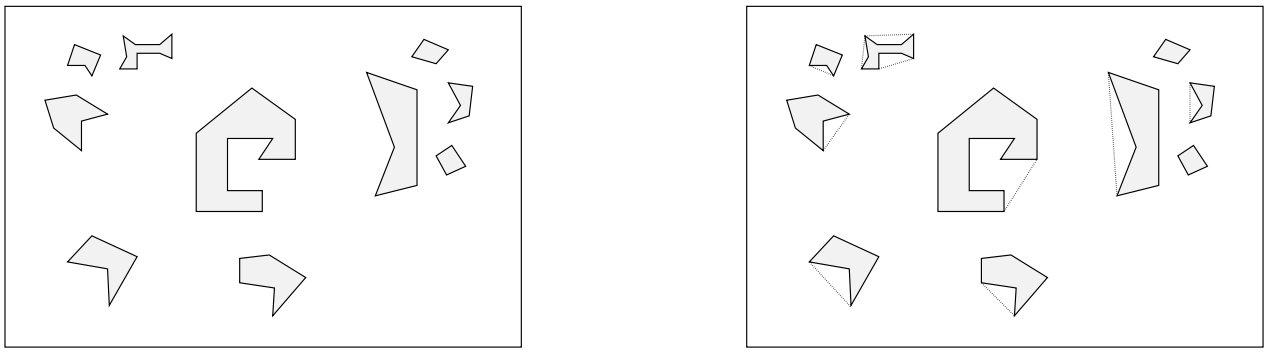

Figure 5.1: Conversion of Non-convex Polygons to Convex Polygons

Another extension of the proposed algorithm based on the Voronoi diagram would be to directly use the Voronoi diagram of polygons rather than those of its vertices.

# Bibliography


[ARS] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. *An Efficient K-Means Clustering Algorithm.* Available online at https://www.cs.utexas.edu/ kuipers/readings/Alsabti-hpdm-98.pdf.

[BKOS97] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkoft. *Computational Geometry: Algorithms and Applications.* Springer, 1997.

[CLRS09] Thomas H. Cormen, Charles E. Lieserson, Ronald L. Rivest, and Cliford Stein. *Introduction to Algorithms.* Cambridge, Massachussets.The MIT Press, 3 edition, 2009.

[GM91] Subir Kumar Ghosh and David M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Computation*, 20:888– 910, 1991.

[GO97] J.E. Goodman and J. O'Rourke. *Handbook of Discrete and Computational Geometry.* CRC Press, 1997.

[GRS] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. *An Efficient Clustering Algorithm for Large Databases.* Available online at http://www.cs.bu.edu/fac/gkollios/ada05/LectNotes/guha98cure.pdf.

[HS97] John Hershberger and Subhash Suri. An Optimal Algorithm for Euclidean Shortest Paths in the Plane. *SIAM Journal on Computing*, 28:2215 – 2256, 1997.

[HSY13] John Hershberger, Subash Suri, and Hakan Yildiz. A near optimal algorithm for shortest paths among curved obstacles in the plane. *SoCG Proceedings of the twenty-ninth annual symposium on Computational geometry*, pages 359 – 368, 2013.

[Jos11] Deepti Joshi. *Polygonal Spatial Clustering.* PhD thesis, University of Nebraska, May 2011.

[Kit] John Kitzinger. *The Visibility Graph Among Polygonal Obstacles: a Comparision of Algorithms.* University of New Mexico. Available online at: http://www.cs.unm.edu/ moore/tr/03-05/Kitzingerthesis.pdf.

[KMN+02] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:881– 892, 2002.

[Lee78] D.T. Lee. *Proximity and Reachability in the Plane.* PhD thesis, University of Illinois at Urbana-Champaign, 1978. A Doctoral Dissertation.

[Lly82] Stuart P. Llyoyd. Least squares quantization in pcm. *IEEE Transactions of Information Theory*, 28:129 – 137, 1982.

[O'R98] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1998.

[SU00] J.R. Sack and J. Urrutia. *Handbook of Computational Geometry.* North Holland, 2000.

# Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Sabbir Kumar Manandhar

Degrees:

Bachelor of Computer Engineering 2010
Tribhuvan University, Institute of Engineering, Pulchowk Campus, Nepal

Thesis Title: Efficient Algorithms for Clustering Polygonal Obstacles

Thesis Examination Committee:

Chairperson, Dr. Laxmi Gewali, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Committee Member, Dr. Justin Zhan, Ph.D.
Graduate Faculty Representative, Dr. Henry Selvaraj, Ph.D.