May 2016

Algorithms for Monotone Paths with Visibility Properties

Bikash Lama Bamjan
University of Nevada, Las Vegas

ALGORITHMS FOR MONOTONE PATHS WITH VISIBILITY PROPERTIES

By

Bikash Lama Bamjan

Bachelor of Computer Engineering
Tribhuvan University,
Kantipur Engineering College, Nepal
2009

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

Bikash Lama Bamjan

entitled

Algorithms for Monotone Paths with Visibility Properties

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

Fatma Nasoz, Ph.D.
Examination Committee Member

Justin Zhan, Ph.D.
Examination Committee Member

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

Abstract

Constructing collision-free paths in Euclidean space is a well-known problem in computational geometry having applications in many fields that include robotics, VLSI, and covert surveillance. In this thesis, we investigate the development of efficient algorithms for constructing a collision-free path that satisfies directional and visibility constraints. We present algorithms for constructing monotone collision-free paths that tend to maximize the visibility of the boundary of obstacles. We also present implementation of some monotone path planning algorithms in Java Programming Language.

Acknowledgements

"First of all, I would like to convey my sincere gratitude to my thesis advisor Dr. Laxmi Gewali for his valuable guidance and support on my thesis. The door to Dr. Gewali was always open whenever I ran into a trouble spot or had a question about my research or report writing and he always steered me towards the right direction. His guidance proved to be the utmost important for the completion of my thesis.

Furthermore, I would also like to thank Dr. Fatma Nasoz, Dr. Justin Zhan and Dr. Henry Selvaraj for being a part of my thesis committee and providing me some insightful comments and encouragement.

Moreover, my courteous appreciation goes to my parents, my beloved wife and my family members as they have always been an integral part supporting me in each and every aspect of my life.

Last but not least, I would like to thank my friends, seniors and juniors who have been the wonderful company and made my stay at UNLV a memorable journey."

Bikash Lama Bamjan

University of Nevada, Las Vegas

May 2016

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Planning collision-free paths in the presence of obstacles is an important problem in computational geometry having applications in many areas that include robotics, computer-aided manufacturing, and very large scale integration(VLSI) [ACM90]. It is often desired for collision-free paths to satisfy additional constraints such as the total length of the path, turn angles in the path, and visibility properties. For planning collision-free paths for aerial vehicles, it is necessary that the implied turn in the path should not be sharp. In some situations, it is necessary to impose clearance from the obstacle to accommodate safety. A collision-free path with enough clearance from obstacles would prevent collision even if there is some error in trajectory computation.

Geometric structures such as visibility graph [BKOS97], Voronoi diagram [BKOS97], and relative neighborhood graph [O'R98] have been used extensively for developing algorithms to solve path planning problems. It is known that the visibility graph induced by the collection of polygonal obstacles contains the shortest collision-free path [O'R98]. On the other hand for computing collision-free path with maximum clearance, Voronoi diagram have been used [BKOS97]. In fact, it is known that the path implied by following the edges of the Voronoi diagram does have maximum clearance [BKOS97].

In this thesis, we investigate the problem of developing efficient algorithms for computing collision-free paths in the presence of polygonal obstacles satisfying two properties to a certain extent: (i) the path should be monotone in the given direction, and (ii) the path should admit increased visibility in the sense that most obstacles boundary should be visible from the computed path.

The thesis is organized as follows. In Chapter 2, we present a brief review of path planning algorithms having visibility properties. In particular, we examine important results for computing watchman route inside polygons and in the presence of polygonal obstacles. In Chapter 3, we formulate a new variation of the path planning problem that satisfies both monotonicity and visibility constraints, which we call Visibility Aware Monotone Watchman Path (VAMWP) problem. An approach for finding heuristic solution for this problem, in the presence of polygonal obstacles is one of the main contributions of this thesis. In Chapter 4, we consider the implementation issues for both (i) monotone path planning problem, and (ii) VAMWP. The implementation is done in JAVA programming language. The prototype program supports friendly user interface so that a user can create obstacle environment interactively and obtain collision-free monotone path connecting given start point S and target point T. Finally, in Chapter 5, we discuss some possible extensions of the proposed algorithms and possible approaches for solving other variations of watchman path problem.

Chapter 2

Review and Preliminaries

In this chapter, we present a comprehensive review of algorithms for constructing the collision-free path in the presence of polygonal domains that satisfy certain visibility properties. In particular, we consider an overview of geometric algorithms for constructing (i) collision-free paths, (ii) collision-free paths satisfying visibility properties, and (iii) paths which are monotone in given direction.

2.1 Collision-free Paths

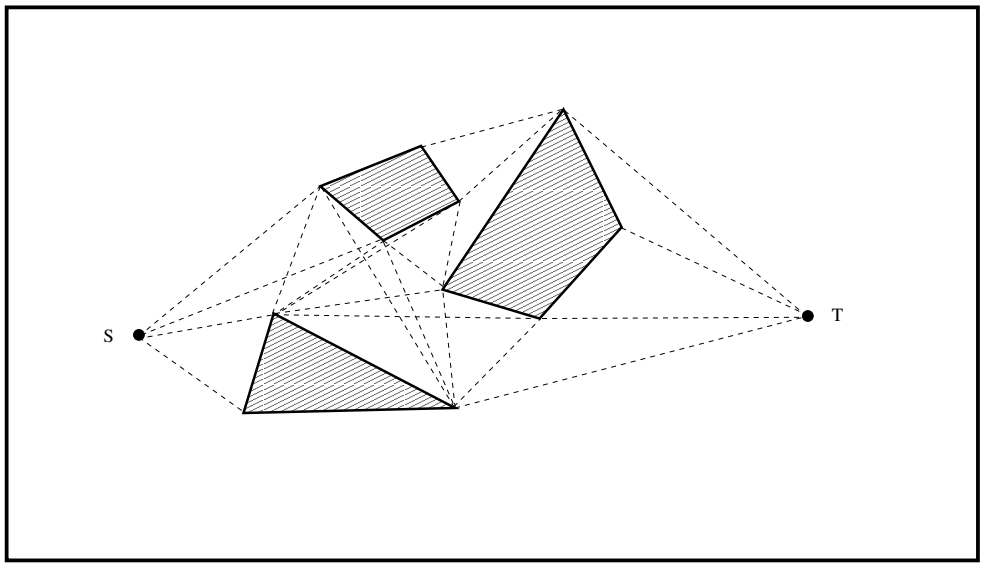


Figure 2.1: Illustrating Visibility Graph

Given a set of polygonal obstacles in the plane, start point S and target point T, a path connecting

S and T that does not intersect with an obstacle is called collision-free path. One of the widely used methods for constructing collision-free paths is based on the visibility graph induced by obstacles and S, T [O'R98]. The vertices of the visibility graph are the vertices of polygonal obstacles, including points S and T. Two vertices v_i, v_j of the visibility graph *VG(V,E)* are connected by an edge if the line segment connecting v_i and v_j does not intersect with obstacles. It is noted that the edges of the obstacles are also part of the visibility graph. Efficient algorithms of time complexity $O(n^2)$ have been reported in the literature [GM91], [CW15] for computing visibility graph.It has been proved that the shortest path connecting two given vertices is contained in the visibility graph. An example of visibility graph induced by a few convex obstacles is shown in Figure 2.1.

For motion planning of point objects, the normal visibility graph, as described above, is good enough. For planning collision-free path for robots with finite extension, a modified form of visibility graph is required [O'R98]. We can imagine the smallest circle of radius r that encloses the robot. Then for constructing collision-free paths for a finite robot, the obstacles are grown by r. It has been established that the collision-free path for a point in the presence of grown obstacles gives the collision-free path for a disk robot of radius r. To actually compute the shortest path, Dijkstra's shortest path algorithm can be applied to the visibility graph [O'R98], [LPW79].

2.2 Watchman Routes

Consider a simple polygon P with vertices v_0, v_1, ... v_{n-1}. Two points p_1 and p_2 inside P are said to be *visible* if the line segment connecting them does not intersect with the external region of P. A *watchman route* inside P is a closed path R such that any point inside P is visible from some point in R. Figure 2.2 illustrates two watchman routes R_1 and R_2 in the interior of the given polygon. It is noted that route R_2 is shorter than route R_1. In fact, it can be easily verified that R_1 is the shortest possible watchman route.

The problem of computing shortest watchman route was first introduced in [CN86], [LK08], [LK10], [GN98] which can be formally defined as follows:

Watchman Route Problem(WRP)

Given: A simple polygon P

Question: Find the shortest route R inside P such that any point inside P is visible from some

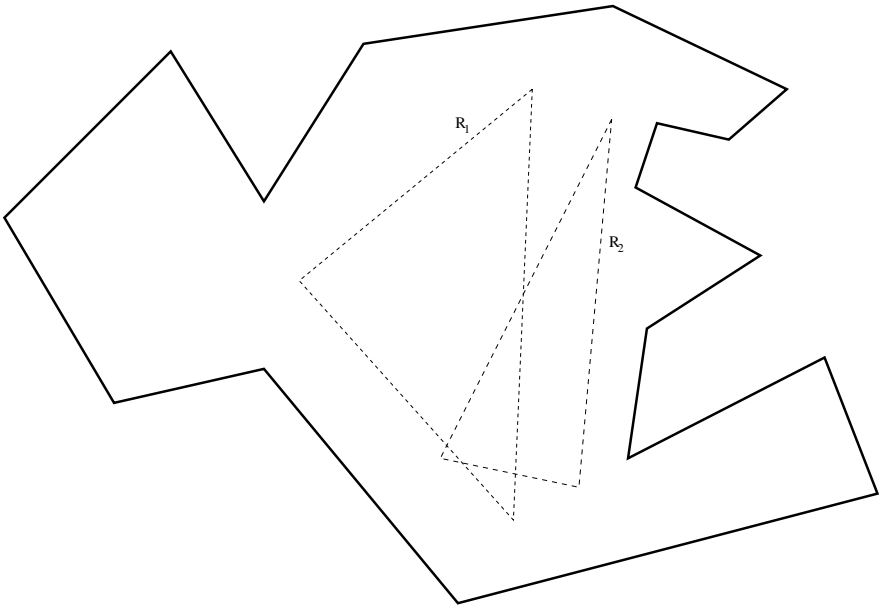


Figure 2.2: Illustrating Watchman Routes

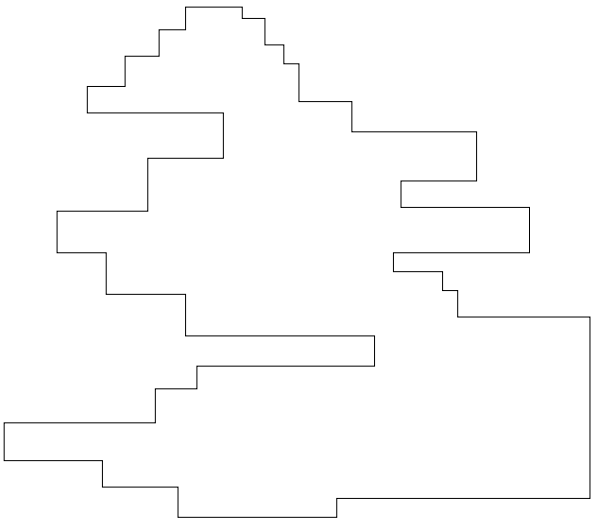
point on R.

The shortest watchman route problem inside a simple problem has been considered by so many researchers [CN86], [LK08], [LK10], [CN91]. The first polynomial-time algorithm for solving this problem inside a simple polygon was reported in [CN91].

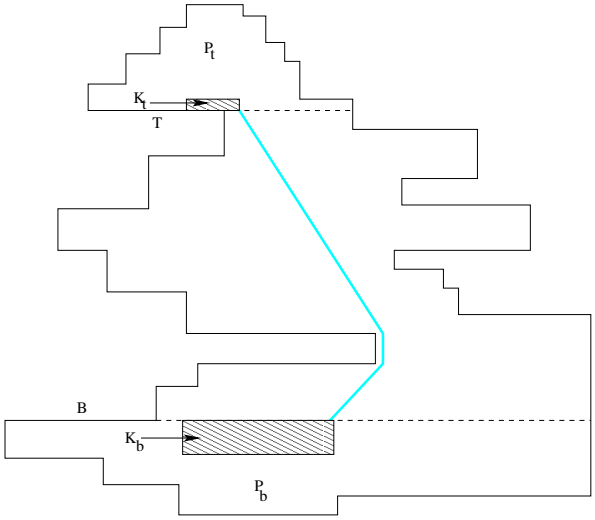
Watchman route problem in the exterior of a collection of polynomial obstacles has been considered. In this version of the problem, it is asked to construct a shortest closed path that lies outside all polygons and all points outside the polygon are visible from some point on the path. This problem is very hard. In fact, it is known that the problem of computing shortest watchman path in the exterior of a collection of polygonal obstacles is NP-hard [CN86].

2.3 Watchman Route inside Orthogonal Polygons

Efficient algorithms for computing shortest watchman route have been reported [CN86] when the simple polygon is orthogonal and monotone in a given direction. It is noted that a simple polygon is orthogonal if its sides are parallel to x or y-axis. A simple polygon is *monotone* with respect to a given line l if the boundary of the polygon consists of two chains each of which is monotone with respect to l. The definitions of a monotone chain are given in most textbooks on computational



(a) A Monotone Orthogonal Polygon



(b) A Monotone Orthogonal Polygon showing Kernels

Figure 2.3: Monotone Orthogonal Polygon

geometry [BKOS97], [O'R98]. Consider an orthogonal polygon that is monotone with respect to the y-axis as shown in Figure 2.3a. The shortest watchman route in a monotone orthogonal polygon is captured in terms of two *kernels* of the polygon. It is noted that *kernel* of a polygon P is the set of points in its interior from which all points inside P are visible.

As introduced in [CN88], we start with a few definitions. The edges of an orthogonal polygon can be distinguished into four kinds. Imagine traversing the boundary of the polygon in the counterclockwise direction. In doing so, if the interior of the polygon falls below a horizontal edge e then

it is a *top edge*. Similarly from a *bottom edge*, the interior of the polygon lies above the edge. The *left edge* and the *right edge* are defined similarly. Let T denotes the topmost *bottom edge* and B the bottom most *top edge*. The portion of the polygon above T is denoted by P_t and that below B by P_b.

It is easily observed that any watchman route must start from some point in P_t, visit some point P_b and complete the route. More specifically, the watchman route must start from the kernel K_t of P_t, visit the kernel K_b of P_b and return back. In Figure 2.3b, kernels K_t and K_b are drawn shaded. It is proved in [CN88] that the shortest path connecting two appropriately chosen points in K_t and K_b can be used to form watchman route. A formal sketch of the algorithm for computing shortest watchman route by adopting the approach given in [CN88] can be written as follows.

2.3.1 Algorithm for Watchman Route in Orthogonal Monotone Polygon

Input: A monotone orthogonal polygon P with vertices v_0, v_1, ... v_{n-1}

Output: Shortest route R from which all points inside P are visible

Step 1: Identify bottom most top edge B and top most bottom edge T.

Step 2: Construct top sub-polygon P_t and bottom sub-polygon P_b.

Step 3: Construct kernels K_t and K_b for P_t and P_b respectively.

Step 4: Find the shortest path R connecting K_t and K_b.

Step 5: Output R as the Watchman route.

2.3.2 Algorithm for Watchman Route in Rectilinear Polygon

It was found that Watchman Route can be determined in simple rectilinear polygons by partitioning the polygon into monotone orthogonal polygons based on paper Optimum Watchman Routes by Wei-Pang Chin and Simeon Ntafos (1986) [CN86]. The partitioning is done by introducing vertical cut edges along the peaks of the polygon. The peaks of the polygon with respect to the y-axis are the top (respectively, bottom) edges that are below (above) both of their adjacent edges. Given the partitioned monotone orthogonal polygons, we identify kernels in each partition shown as the shaded areas in Figure 2.4. Some partition may have a single kernel and some may have both top and bottom kernels. The Watchman Route is then the shortest path visiting the subset of these kernels. Some kernels are dominant over other kernels causing the skipping of non-dominant kernels i.e these kernels will not be included in the Watchman Route. In the figure, Kernel K_6 is dominant

over K_5 so K_5 will be skipped. Then the Watchman Route is determined using the algorithm given in the paper [CN86] along the top and bottom edges pair of these kernels.

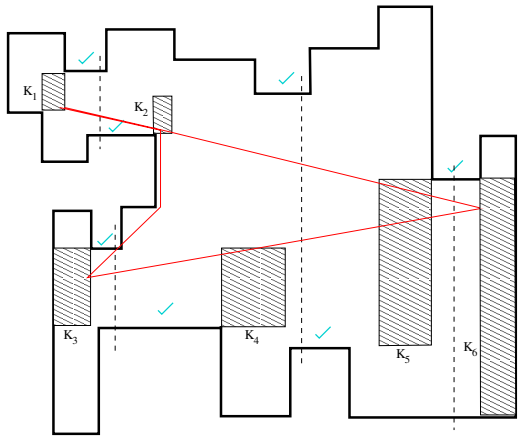


Figure 2.4: Watchman Route in Simple Rectilinear Polygon

2.4 Monotone Collision-free Paths

In some application, it is required to find collision-free path that progresses only in the forward direction i.e. the path never backs up. Such a path can be conceptualized in terms of monotonicity in a given direction. Consider a direction $\hat{d}$ along the x-axis. A path $\hat{P_m}$ consisting of line segments $\hat{P_m} = <s_1, s_2, ..., s_n>$ is called monotone along $\hat{d}$ if the projections of s_i, $1 \leq i \leq n$ along x-axis do not overlap. In Figure 2.5, two paths $\hat{P_m}$ and $\hat{Q_m}$ are shown where $\hat{P_m}$ is monotone along $\hat{d}$ and $\hat{Q_m}$ is not.

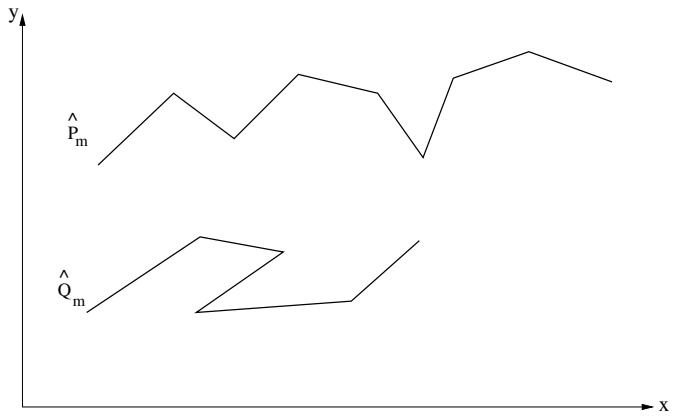
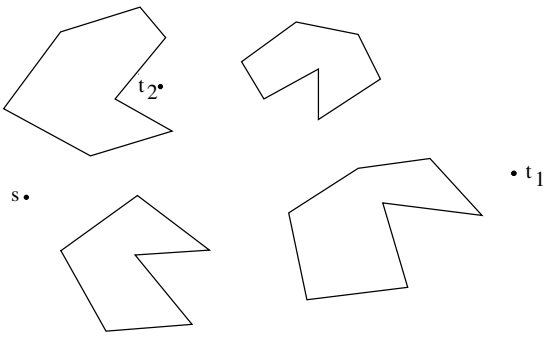


Figure 2.5: Illustrating Path $\hat{P_m}$ that is monotone along x-axis

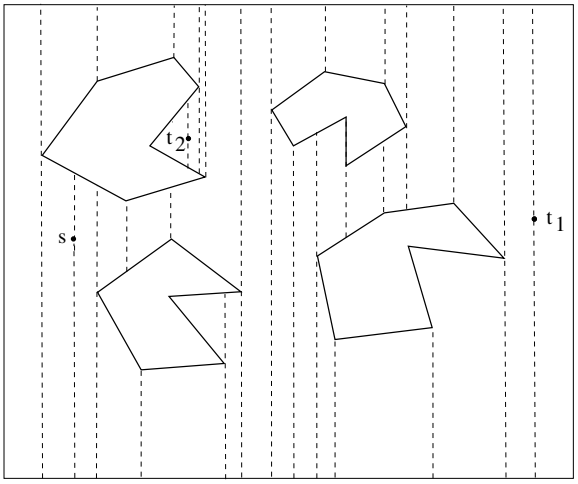
It is found that collision-free monotone path between a source point and a destination point in the presence of polygonal obstacles can be determined [ACM90]. Consider a number of obstacles as shown in Figure 2.6a. Let s be the source point and t_1 be the destination point.

The algorithmic ingredient given in [ACM90] can be briefly described as follows. For simplicity, we assume the obstacles are bounded by a rectangular box. To find the shortest path, first extend the vertical lines from each vertex and given source and destination points in both directions to meet an obstacle or the boundary. The resulting diagram forms a graph called Vertical Adjacency Graph (VAG) consisting vertical projections as shown in Figure 2.6b. The edges of VAG will be all the vertical lines, horizontal projections on x-axis and the edges of the polygonal obstacles. The length of an edge e_i of VAG is equal to its projected length along x-axis. By the concept of projections, the projected length of an edge of the polygon parallel to x-axis is equal to it's own length while vertical edge parallel to y-axis is equal to zero. All other edges' projections length will have shorter length than it's own length.

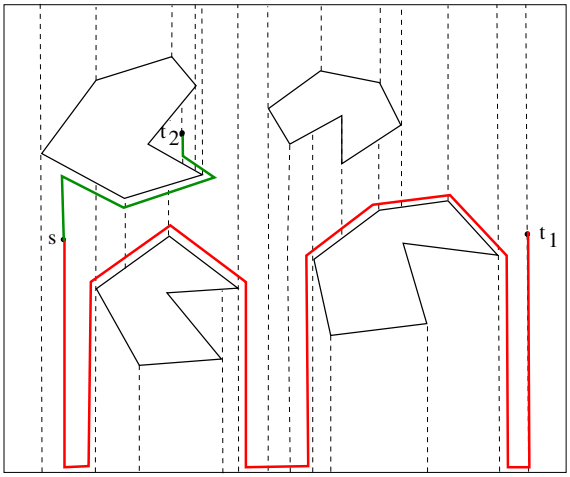
The shortest path between a source point and a destination point can be found by applying Dijkstra's algorithm [CLRS09]. After finding the shortest path, we can then test if the path is monotone or not. A path is monotone if the length of the shortest path connecting s to t in VAG is equal to the difference between x-coordinates of s and t [ACM90]. In Figure 2.6c, the path shown in red is monotone to x-axis while path shown in blue is not monotone. In [ACM90], it is shown that such a monotone path can be found in *O(nE)* time, where E is the size of the visibility graph defined by the n vertices of the obstacles. It is also shown in [ACM90] that if all obstacles are convex then there always exist a monotone path between any two given points.



a) Input Obstacles



b) Vertical Adjacency Graph



c) Generated Monotone Paths

Figure 2.6: A number of obstacles

Chapter 3

Visibility Properties of Monotone Paths

In this chapter, we present the main contributions of the thesis. We present algorithms for designing monotone paths that tend to maximize the region visible from the path.

3.1 Preliminaries and Problem Definition

We first start with a few definitions. The set of points reachable by a monotone path starting from a given start point S is enclosed by Upper Boundary Chain (UBC) and Lower Boundary Chain (LBC). *UBC* induced by start point s in the presence of the polygonal obstacles can be defined iteratively by a sequence of vertical rays starting from s. We can imagine a ray r_s starting from s upward. Ray r_s will hit either an obstacle O_{i1} or the upper boundary of enclosing box. If r_s hits O_{i1} at a hit point h_i, then we follow the boundary of O_{i1} in the forward direction until we encounter the rightmost vertex of O_{i1} at v_{i1}. From v_{i1}, we can shoot a ray r_{i1} up to meet either the upper boundary of enclosing box or obstacle O_{i2}. If ray r_{i1} hits O_{i2} at h_{i2}, then we again follow the boundary of O_{i2} forward to reach the rightmost vertex v_{i2}. We continue like this alternating between ray segment and polygon boundary to construct a path which we call *UBC*. Similarly, Lower Boundary Chain (*LBC*) is defined. Further detail on the construction of UBC and LBC are in Chapter 4. This kind of boundary construction is usually referred to as *waterfall model* [ACM90]. We refer to the region bounded by *UBC* and *LBC* as the *core region*. The region outside the core region is called *forbidden region*. Figure 3.1 illustrates these definitions.

Our interest here is to examine the visibility properties of a monotone path connecting start point S and target point T, both located in the core region. The first problem we introduce concerns on determining whether a given S-T-monotone path is a watchman path for the core region. We can recall from Chapter 2 that a watchman path g_1 is such that every point on the free region is visible from some point on g_1. The problem can be formally defined as follows.

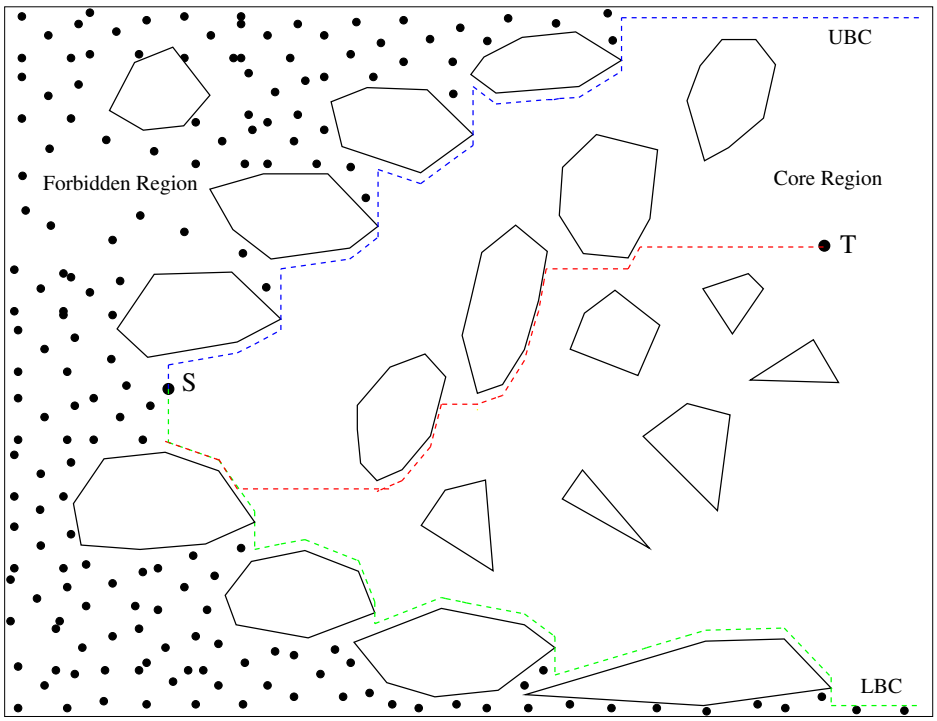


Figure 3.1: Defining Core Region and Forbidden Region

Monotone Watchman Path Detection Problem (MWPDP)

Given: A monotone path g_1 connecting start point S and a target point T in the presence of 2D convex polygonal obstacles. The obstacles are enclosed in a rectangular box.

Question: Is g_1 a watchman path for the core region induced by obstacles and points S, T?

Figure 3.2 illustrates an example of watchman path. It can be verified that all free region inside core region is visible to some point in the indicated monotone path.

To determine whether a given path is a watchman path or not, we could construct visibility polygons

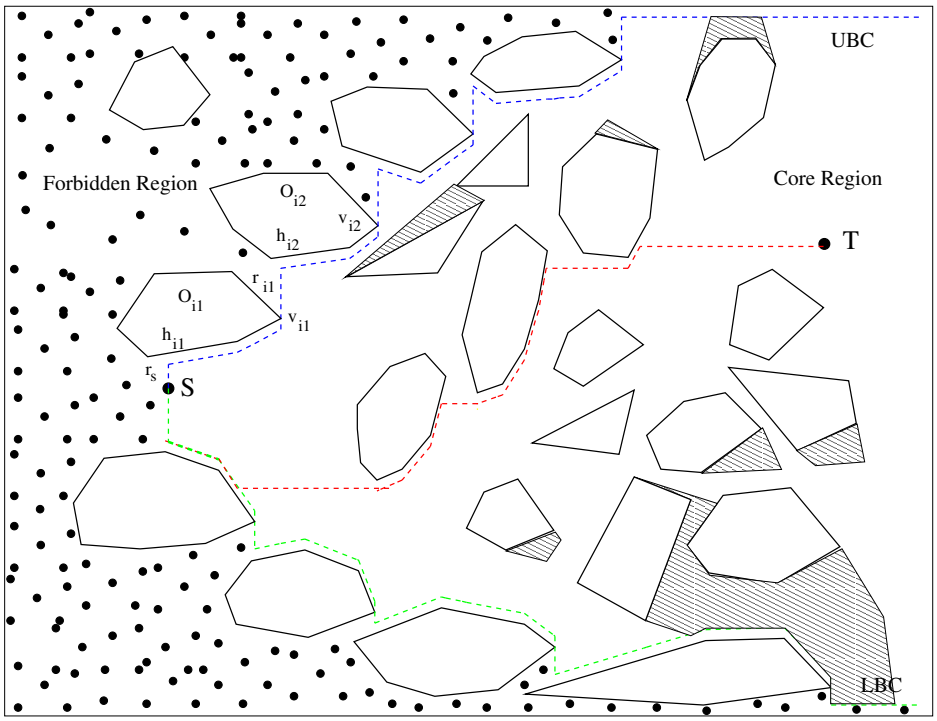


Figure 3.2: A Monotone Watchman Path

from selected points on the route. If the union of corresponding visibility polygons indeed covers the boundary of all obstacles inside the core region, then the path is a watchman path. It may be noted that the visibility polygon from a point q is the set of points visible from q. Figure 3.3 illustrates an example of visibility polygon.

For each edge e_i of an obstacle, we can define **support edge** of e_i denoted as *sup(e_i)* obtained by extending e_i in both directions to meet another obstacle or the bounding box.

Definition 3.1 *An obstacle edge e_i is called **shadow edge** with respect to a path g_1 if (a) sup(e_i) does not intersect with g_1 and (b) both g_1 and obstacle $O(e_i)$ are on the same side of sup(e_i). In Figure 3.3, e_1 and e_2 are shadow edges. On the other hand, e_3 and e_4 are not shadow edges. The presence of a shadow edge is the witness that path g_1 is not a watchman path. This is stated in the following lemma.*

Lemma 3.1 *The occurrence of shadow edge implies that the path g_1 is not a watchman path.*

An algorithm for marking shadow edges can be described in a straightforward manner. We construct

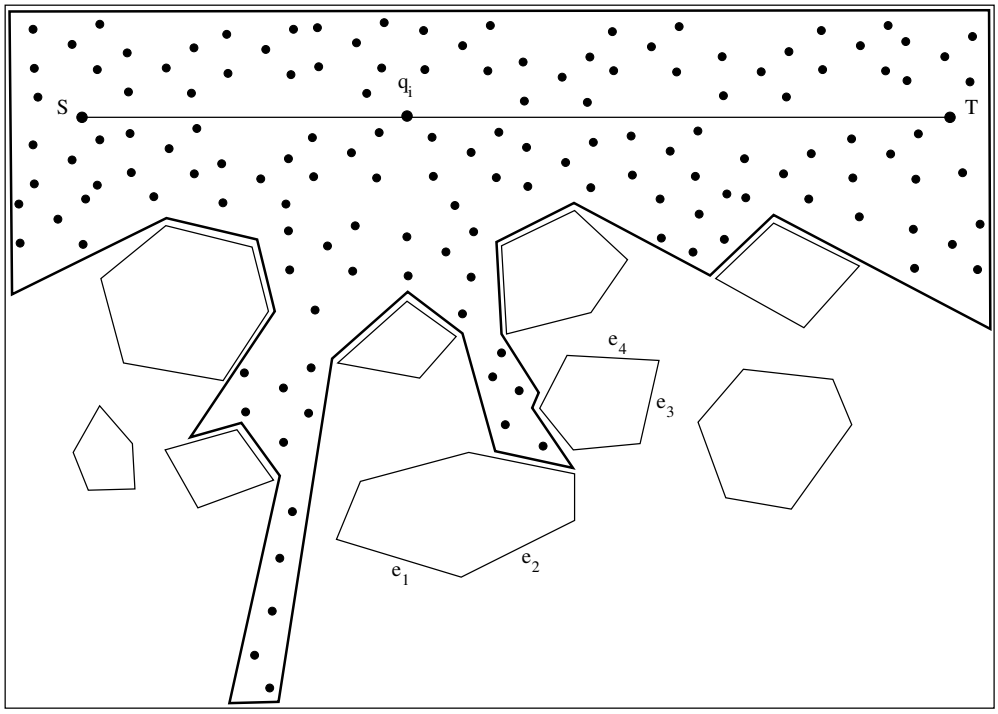


Figure 3.3: Illustrating Visibility Polygon

sup(e_i) for all edges of obstacles. For each *sup(e_i)*, we can check the conditions of Definition 3.1 by using left-turn/right-turn check [O'R98].

3.2 Algorithm Development

To compute the area visible from monotone path g_1, one approach would be to compute visibility polygons from finite set of points, say m, located on path g_1. The union of these visibility polygons will give the area

visible from the path g_1. For this purpose, we consider (i) all supporting line segments of obstacle edges and (ii) all supporting line segments of **connecting edges**. Note that connecting edges are the edges connecting two vertices of different obstacles that do not intersect with other obstacles.The set of supporting line segments through obstacle edges and connecting edges partition obstacle edges and edges of boundary box into sub-edges. Figure 3.4 illustrates the arrangement of supporting line segments and supporting connecting segments. Such an arrangement induces **interior points** u_1, u_2, ... , u_k on the path g_1. Our approach for computing the region visible

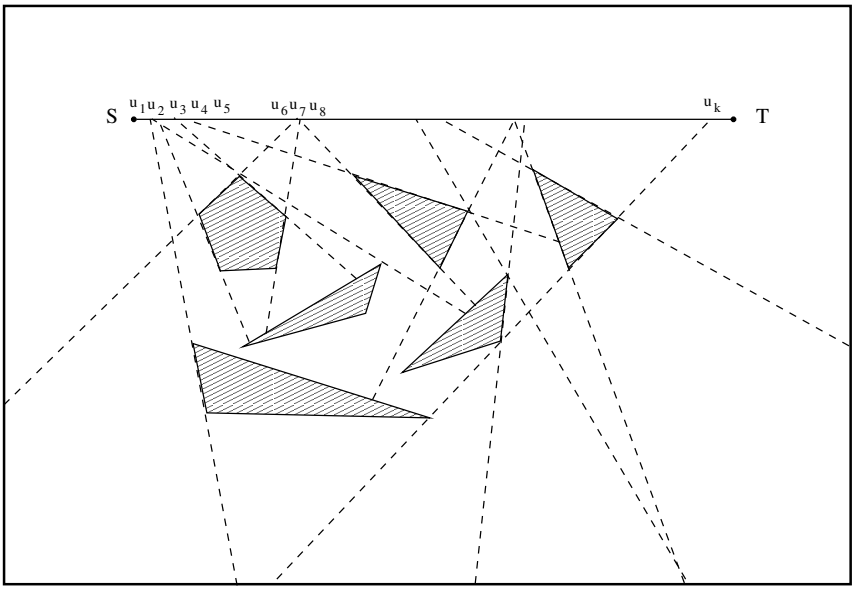


Figure 3.4: Arrangement of Supporting Segments and Obstacle Edges

from path g_1 is to compute visibility polygons from S, T, and the interior points. Specifically, visibility polygons are computed from each interior points, and points S and T and combine them to obtain the entire visible region from the path. The algorithm is formally sketched as Visible Region Computation (Algorithm 3.1).

Algorithm 3.1: Visible Region Computation

Data: i) Set of convex obstacles Q_1, Q_2, ..., Q_n

ii) Bounding Box B

iii) Path g_1

Result: The parts of the obstacle boundary visible from g_1

1 Compute the partitioning of path g_1, obstacle edges, and bounding of B.

2 Let u_1, u_2, ..., u_k be the interior points on path g_1

3 Compute visibility polygons from interior points u_i's, S and T. Sub-edges of obstacles lying on visibility polygons are marked visible.

4 Scan boundary of obstacles and output those that are marked visible.

Lemma 3.2 *Let u_i, u_{i+1} be two consecutive interior points on the S-T-Monotone path as illustrated in Figure 3.5. Let VisP(u_i) denote the visibility polygon from point u_i. Then VisP(u_i) U VisP(u_{i+1}) contains VisP(q), where q is any point in the segment (u_i, u_{i+1}).*

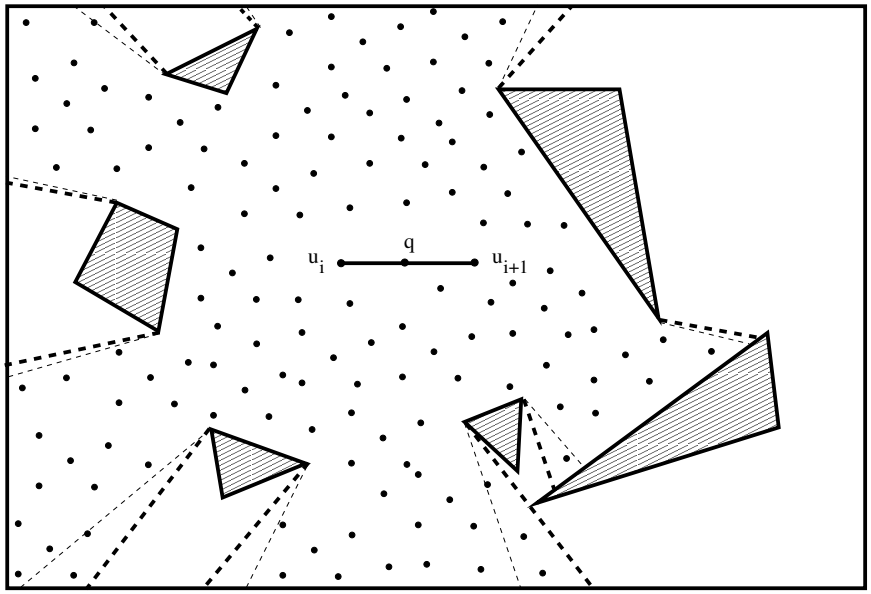


Figure 3.5: Illustrating the proof of Lemma 3.2

Proof: The structure of the visible area *VisP(u_i, u_{i+1})* from the line segment *(u_i, u_{i+1})* is a visibility polygon whose boundary consists of alternate sequence of obstacle edges and visibility edges as shown in Figure 3.5. The visibility edges are drawn as dashed line segments.

Specifically, *VisP(u_i, u_{i+1})* consists of k funnels f_1, f_2, ... , f_k arranged angularly around segment *(u_i, u_{i+1})*. The area visible from q is a visibility polygon *VisP(q)* whose structure also consists of funnels f_1', f_2', ... , f_k' arranged angularly about q. Now observe that f_i' is properly contained in f_i. The visible areas other than the funnels are identical for both interior point q and segment *(u_i, u_{i+1})*. □

3.3 Heuristics and Approximations

The problem of computing the shortest watchman path in the presence of convex obstacle is known to be NP-Hard [CN86]. So, it is interesting to seek for the development of approximation algorithms that finds a route or path from which most portion of obstacles boundaries is visible from the path. We now proceed to deform the monotone *S-T-Path* g_1 lying in the core-region to increase the visibility. For example, in Figure 3.6, the area not visible from the *S-T-Monotone path* g_1 in the core region, is shown shaded. (Now onward, we use the term *S-T-Monotone path* to indicate the

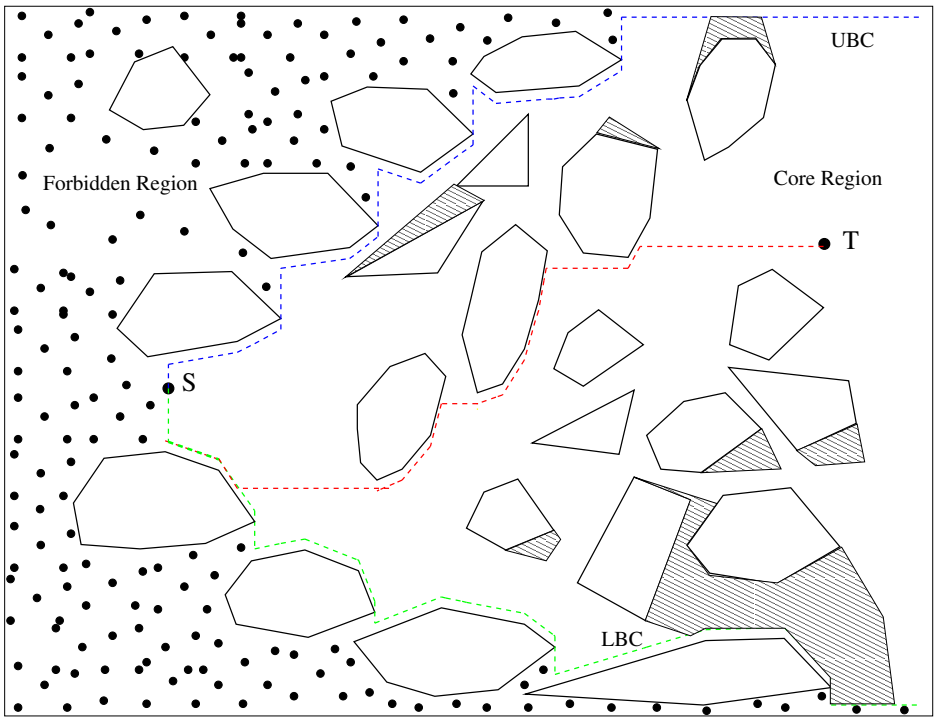


Figure 3.6: Shaded Areas not visible from S-T-Monotone Path

monotone path formed by using Waterfall-Model described in Chapter 2).

Funnel Heuristic: Take a straight line segment s_1 from the *S-T-Monotone path* g_1. We replace s_1 by a *funnel path* consisting of two concave chains as shown in Figure 3.7, where three funnel paths are depicted, drawn by thick edges.

Constructing Funnel Extensions: Consider a ray originating from a point q_i, on the S-T-Monotone path g_1 and extending either up or down to meet a hit point h_i. Let s_1 be a line segment on g_1 that contains q_i. Let l_1 be the shortest path connecting left endpoint of s_1 to h_1. Similarly, r_1 is the shortest path connecting right endpoint of s_1 to h_1. The chains l_1 and r_1 form a funnel with s_1 as its *lid*. A funnel can be constructed for any point on path p_1. Figure 3.7 shows three funnels on a S-T-Monotone path g_1.

Strategies for Locating Funnel Extensions: Now we describe strategies for locating funnel extensions on S-T-Monotone path g_1 so that the total length of the path is not very long and the visibility from the path is also increased.

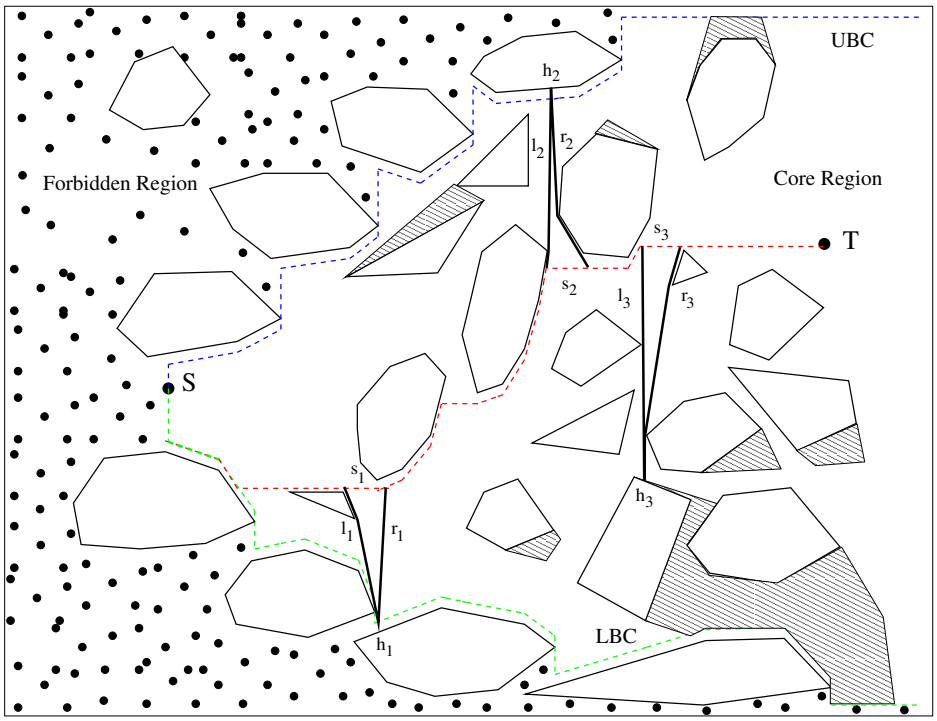


Figure 3.7: Construction of Funnel Path

We need to construct funnel extensions only at a few selected positions so that the length of the path is not too large. The positioned extensions should possibly increase visible area. With this objective, the S-T-Monotone path is partitioned into several line segments called *basic-segments*. Each basic segment is evaluated for positioning funnel extensions.

The S-T-Monotone path g_1 is partitioned by using the *vertical chords* of vertical adjacency graph (VAG) introduced in Chapter 2. The intersection of vertical chords with path g_1 are the *steiner points* on the path. Steiner points and original vertices of g_1 partition the path g_1 into basic segments. Figure 3.8 shows the steiner points induced by vertical chords of VAG, which are shown as filled dots. The connected free-space above a basic segment bounded by two consecutive vertical chords is referred to as *up-column*. Similarly, the free-space below basic segment between consecutive vertical chords is called *down-column*.

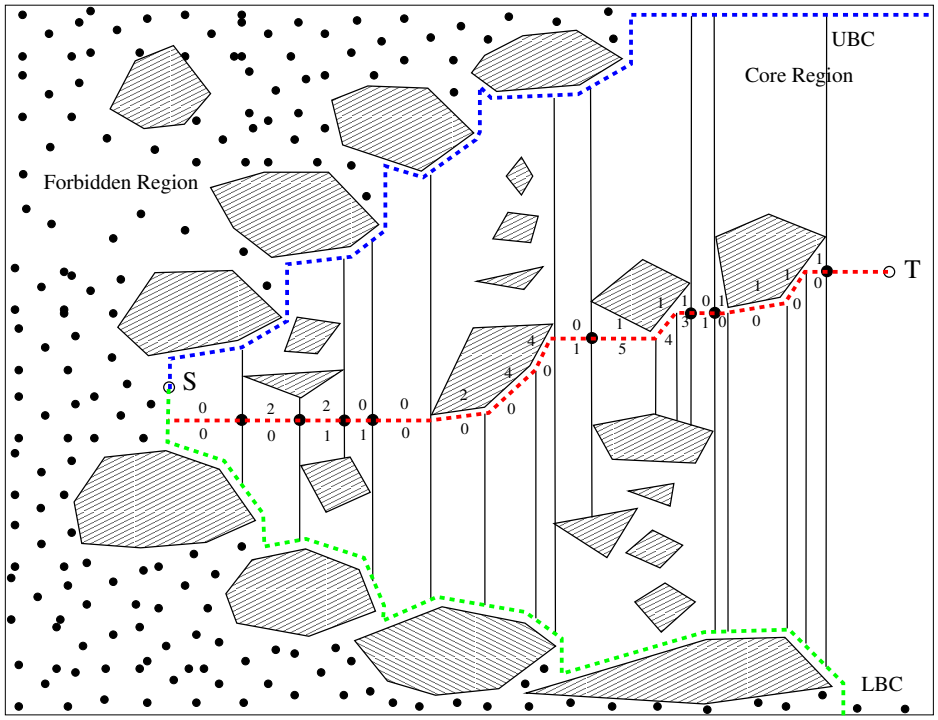


Figure 3.8: Illustrating Basic Segments, Upper/Lower Layer Values

For each basic-segment, we define two properties: (i) *upper-layer value*, (ii) *lower-layer value*. The number of obstacles intersected by a ray originating at a point on a basic segment b_i and extending vertically upward up to the boundary of the core region is the upper-layer value for b_i. Lower-layer value for b_i is defined similarly. In Figure 3.8, upper-layer and lower-layer values are shown above and below selected basic segments. The criteria for placing funnel extensions above a basic segment b_i can be listed as:

- *Criteria 1:* The height of up-column should be large.
- *Criteria 2:* The upper-layer value of b_i should be low and the upper-layer value of adjacent basic segments (b_{i-1} or b_{i+1}) should be high.

The criteria for placing funnel extensions below a basic segment are defined analogously.

Chapter 4

Implementation

In this chapter, we describe the implementation detail of the algorithms proposed in Chapter 3. In particular, we present (i) obstacle modeling, (ii) data structure for representing the vertical adjacency graph induced by input obstacles and the positioning of the start point S and target point T, (iii) overview of the prototype program developed in the Java Programming Language, and (iv) the results of experimental investigation.

4.1 Obstacle Modeling

Obstacles are modeled by convex polygons and they are assumed to be enclosed in a rectangular bounding box. The coordinates of the vertices of a polygon are stored as they occur when the boundary of the obstacle is traversed in counterclockwise order. Each convex obstacle has two distinguished vertices *leftmost vertex* and *rightmost vertex*. The vertex with the smallest x-coordinate is the leftmost vertex and the one with the largest x-coordinate is the rightmost vertex. We can imagine a line segment called *skeleton-segment* that connects the leftmost vertex to rightmost vertex. Due to convexity, the skeleton-segment lies completely inside the obstacle. Each obstacle's boundary can be viewed to consist of two disjoint chains: *top-chain* and *bottom-chain*. The top-chain lies to the left of directed skeleton-segment and the bottom chain lies to the right. As described in Chapter 2, the vertical adjacency graph is formed by constructing vertical line segments (called *vertical chords*) from each leftmost and rightmost vertices. Each vertical chord extends both up and down from leftmost/rightmost vertex until they hit another obstacle or the edge of the rectangular bounding box. These definitions and example modeling are illustrated in Figure 4.1.

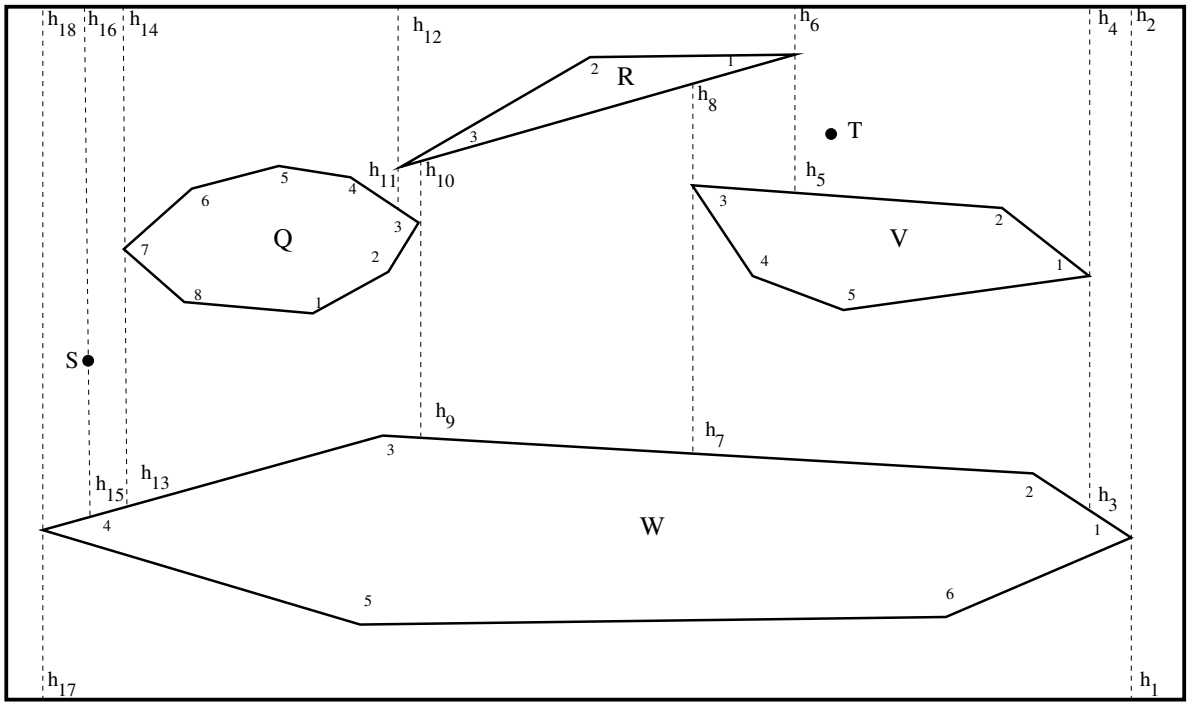


Figure 4.1: Obstacle Modeling

4.2 Data Structures

In this section, we describe the data structures used for modeling the vertical planar graph and the obstacles. Firstly, we present how the vertical planar graph is stored and provide the algorithms for the construction of upper, lower and back paths. In the next subsection, we explain the actual implementation of the data structures in our Java Project.

4.2.1 Data Structure for storing Vertical Planar Graph

Generalized Vertex
Coordinates
Previous Vertex
Next Vertex
Top-Hit
Bottom-Hit

Table 4.1: Generalized vertex

The graph induced by obstacles, start point S, target point T and the vertical chords from leftmost and rightmost vertices induce a planar graph called Vertical Adjacency Graph (VAG).

Rather than explicitly constructing VAG, we model it implicitly by assigning *top-hit* and *bottom-hit* vertices for all leftmost and rightmost vertices of obstacles. Each vertex is thus considered to have the following fields.

The previous vertex of a vertex v is the vertex occurring before v when the boundary of the obstacle is traversed in counterclockwise direction. Similarly, the next vertex is the vertex that occurs after v. The *top-hit* vertex is the first hit-point when a ray is extended up from a leftmost or rightmost vertex. Similarly, the *bottom-hit* vertex is the first hit-point with an obstacle or boundary edge when the ray originates from rightmost(or leftmost) obstacle vertex extends downwards. For each hit-vertex, there will be either a bottom-hit vertex or top-hit vertex depending upon its position on the obstacle boundary. We can illustrate this with some selected vertices as shown in Table 4.2 where v_3 is the leftmost vertex of obstacle V in Table 4.2. We use the convention of naming obstacles with uppercase letters and their vertices with the corresponding lowercase letter.

Vertices	**Top-Hit**	**Bottom-Hit**	**Previous**	**Next**
v_3	h_8	h_7	h_3	v_4
h_8	nil	v_3	h_{10}	r_1
S	h_{16}	h_{15}	nil	nil
h_7	v_3	nil	w_3	h_9
h_8	nil	v_3	h_{10}	r_1

Table 4.2: Fields of Specific Vertex

Some of the regions can not be reached by a monotone path starting from source point S. Such a region can be identified by constructing a *upper bounding chain*. The region near the top portion of the bounding box that can not be reached by a monotone path is shown in Figure 4.2 filled with dots. The region on the left side of the upper bounding path indicated in Figure 4.2, is a portion of such a region which we call *forbidden region*.

Similarly, a *lower bounding chain* can be constructed starting from S that progresses bottom right is shown in Figure 4.2. Specifically, <S, h_1, h_2, h_3, h_4 > is upper bounding chain and <S, h_5, h_6, h_7, h_8, h_9> is lower bounding chain. Upper bounding chain can be constructed by alternately identifying upper hit-points and rightmost vertices of obstacles starting from start point S. A vertical ray starting from start point S with the first hit on obstacle at hit-point h_1. We can then progress

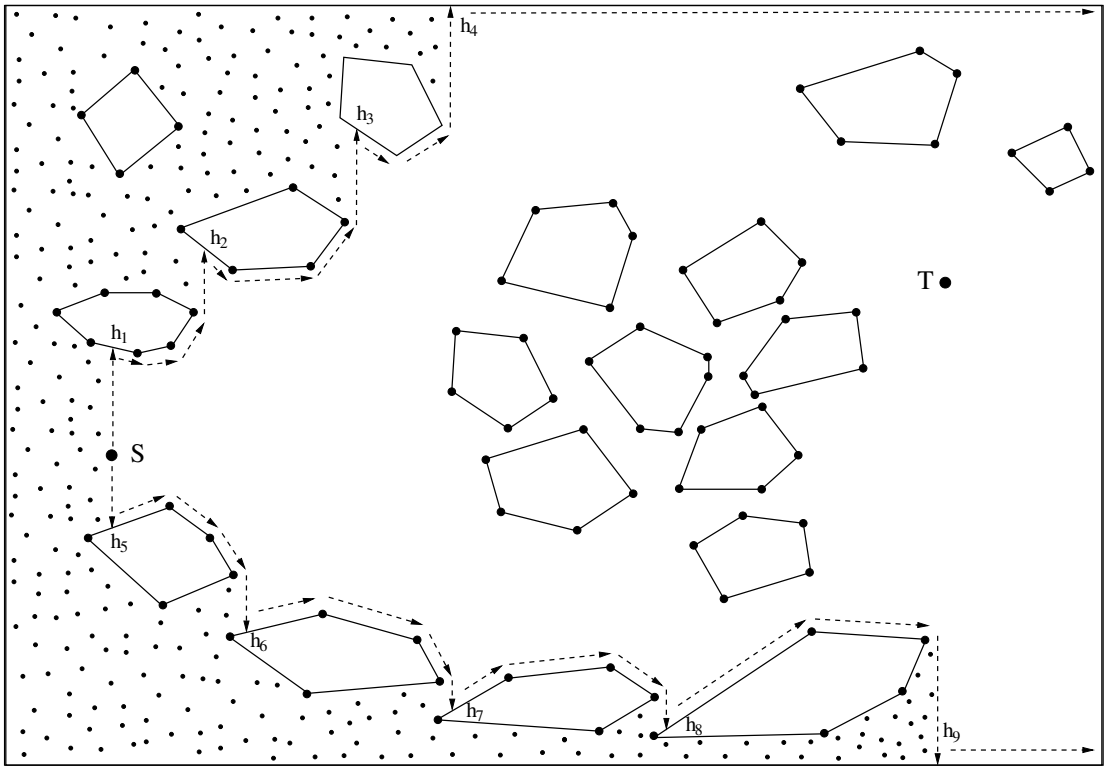


Figure 4.2: Illustrating Forbidden Region

forward to right starting at h_1 along the boundary in the counterclockwise direction to identify rightmost vertex r_1. From this rightmost vertex r_1, we can shoot a ray vertically up to identify the second hit-point h_2. From hit-point h_2, we can progress forward right to discover the second rightmost point r_2. This process of identifying hit-point and rightmost point alternately is repeated until the boundary of the bounding box is encountered. A formal sketch of the algorithm for computing upper bounding chain is sketched as shown in UpperChain Algorithm (Algorithm 4.1). This algorithm takes collection of obstacles and start point as input and outputs the sequence of vertices describing the upper bounding chain.

Algorithm 4.1: UpperChain Algorithm

Data: i) startPoint S, ii) ObstacleSet

Result: *Path*

```
1  Path = φ;
2  currentPoint = startPoint;
3  Insert currentPoint into Path;
4  currentPoint = getHitPointUp(currentPoint);
5  Insert currentPoint into Path;
6  while (currentPoint is not on the boundary of Bounding Box) do
7      currentPoint = getRightMostVertex(currentPoint);
8      Insert currentPoint into Path;
9      currentPoint = getHitPointUp(currentPoint);
10     Insert currentPoint into Path;
11 // Now currentPoint is on the top of bounding box
12 Insert top-right corner point of bounding box into Path
13 Output Path
```

Algorithm 4.2: LowerChain Algorithm

Data: i) startPoint S, ii) ObstacleSet

Result: *Path*

```
1  Path = φ;
2  currentPoint = startPoint;
3  Insert currentPoint into Path;
4  currentPoint = getHitPointDown(currentPoint);
5  Insert currentPoint into Path;
6  while (currentPoint is not on the boundary of Bounding Box) do
7      currentPoint = getRightMostVertex(currentPoint);
8      Insert currentPoint into Path;
9      currentPoint = getHitPointDown(currentPoint);
10     Insert currentPoint into Path;
11 // Now currentPoint is on the bottom of bounding box
12 Insert bottom-right corner point of bounding box into Path
13 Output Path
```

After constructing the upper and lower bounding chains, the next step is to construct a path from the target point T to source point S called *back chain* which will ultimately determine the final path from source S to target T. This back chain path can be constructed in similar fashion as upper and lower chains but the direction of the path will move towards left. Before describing how the back chain is constructed, we need to define certain properties of the convex obstacle. In each obstacle, the segment joining the leftmost and the rightmost vertices divides the obstacle's boundary into two parts. The boundary of the obstacle which lies in the upper part of this segment is termed as an *upper chain* and the other one is referred to as a *lower chain*.

We can now describe the construction of the back chain. Starting from target point T, a horizontal ray is emanated until it hits an obstacle with first hit point referred as *left hit-point*. After identifying the left hit point, selection of next point in the path depends upon whether the left hit point lies in the upper chain or lower chain of the obstacle.The left-hit point will be either on the right-chain i.e. the chain from bottom most vertex to top-most vertex, counterclockwise. If the left-hit point is above the right most vertex, then the traversal is done in the counterclockwise direction along

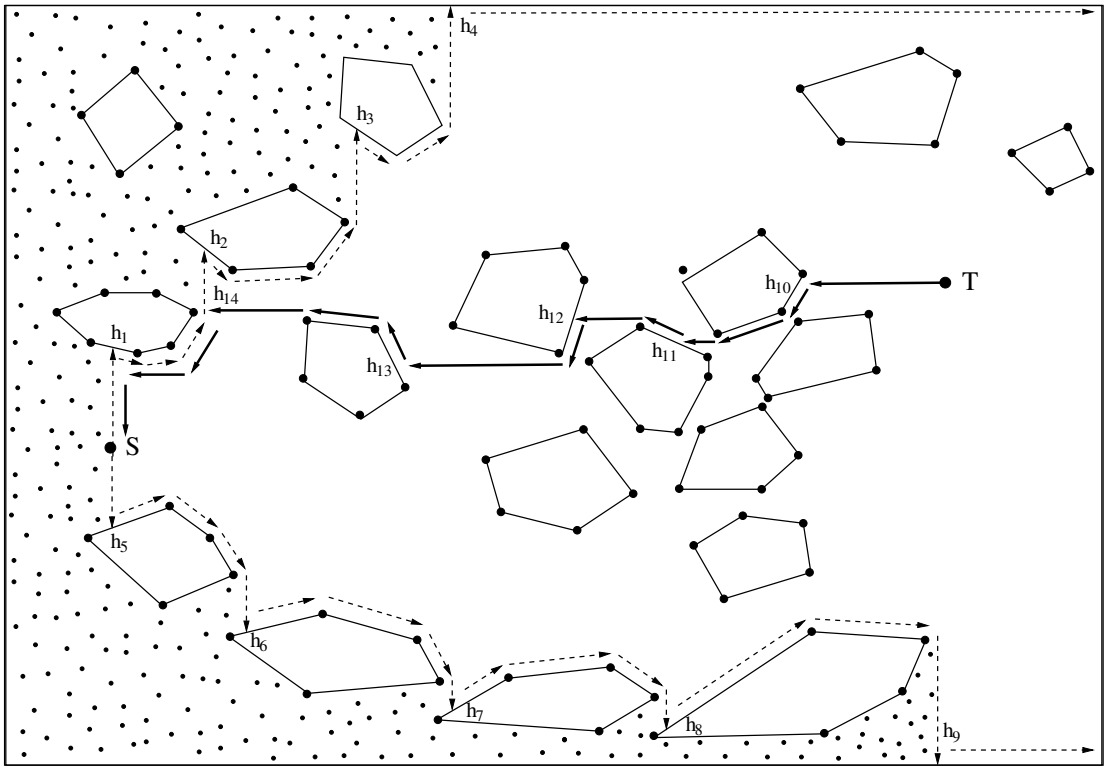


Figure 4.3: Tracing of Back Chain Path

the boundary to reach the top most vertex. On the other hand, if the left-hit point is below the rightmost vertex, then the traversal is done clockwise to reach the bottom most vertex. This process of constructing the left-hit point and traversing the obstacle (clockwise or counterclockwise) is repeated until the left-hit point falls on the vertical edge of the bounding box. A formal sketch of the algorithm for constructing backward path is listed as BackChain Algorithm (Algorithm 4.3).

Algorithm 4.3: BackChain Algorithm

Data: i) targetPoint T, ii) ObstacleSet

Result: *Path*

```
1  Path = φ;
2  currentPoint = targetPoint;
3  Insert currentPoint into Path;
4  currentPoint = getHitPointLeft(currentPoint);
5  Insert currentPoint into Path;
6  while (Path does not intersect vertical edge of Boundary Box) do
7      if currentPoint is member of TopChain
       currentPoint = getTopMostVertex(currentPoint);
8      Insert currentPoint into Path;
9      currentPoint = getHitPointLeft(currentPoint);
10     Insert currentPoint into Path;
11     else currentPoint = getBottomMostVertex(currentPoint);
12     Insert currentPoint into Path;
13     currentPoint = getHitPointLeft(currentPoint);
14     Insert currentPoint into Path;
15 // Now currentPoint intersects Paths from UpperChain or LowerChain
16 Compute intersection between Path and the union of UpperChain and LowerChain and
   insert the appropriate position into Path
17 Output Path
```

4.2.2 Data Structures in Java implementation

The main data structures used for modeling of obstacles and constructing the required paths are implemented using the following Java classes:

G_Node

The vertices of the obstacles are represented as Class *G_Node*. Table 4.3 shows the Class Interface Diagram of this class. Each vertex of the obstacle has x and y coordinates and it's type and kind. A vertex can have type as *'L'* for left, *'R'* for right, *'B'* for bottom, *'T'* for top, *'D'* for outer

box boundary vertex or *'U'* as default. Similarly, the kind of the vertex can be *'S'* for a *Steiner* vertex,*'O'* for an obstacle vertex, *'H'* for a hit vertex and *'I'* for an isolated vertex. As discussed earlier in the previous section, all of the vertices in the obstacle have their next and previous vertices. Similarly, the hit vertices can be classified as up-hit, down-hit and left-hit vertices. The leftmost and rightmost vertices of the obstacle have up and down-hit vertices and the topmost and bottom-most vertices of the obstacle have left-hit vertex. The next, previous, up-hit, down-hit and left-hit vertices are also represented as *G_Node* class. To access these vertices, class *G_Node* implements various set and get methods as shown in the Table 4.3.

MyJPanel

Class *MyJPanel* is the main implemented class where all the hit points are calculated and the upper, lower and back chain paths are constructed. Table 4.5 shows the Class Interface Diagram for *MyJpanel*. This class defines source and target points as a type of *G_Node* class. Initially, each obstacle is represented as vector of vertices of type *Point* and all obstacles are stored in a vector. After computation of next and previous vertices and the hit points, this representation of the obstacles is converted as the vector of vectors of type *G_Node*. Similarly, the outer big bounding box; and upper, lower and back chain paths are represented as vector of vertices of type *G_Node*. To compute the hit vertices and construct upper, lower and back chain paths, this class implements methods as *processHitPointsFromAllVertices()*, *processHitPointsLeftFromAllVertices()*, *constructUpperBoundary()*, *constructLowerBoundary()*, *constructBackwardPath()* etc. This class extends the class *JPanel* of Java that supports the Swing component architecture.

JGUI4

Class *JGUI4* mainly implements the graphical user interface and the reading and saving of the *dcel* file containing the plotting of the obstacles and the paths. The class contains the data structures of types *JButton*, *JCheckBox*, *JRadioButton* and *JPanel* as shown in Table 4.5 which are derived from class *JFrame* of Java that also supports the Swing component architecture. The methods like *setUpMenuBar()*, *updatePolyPanel*, *repaint()*, *saveToFile()*, *readFromFile* etc are implemented in this class.

4.3 GUI and Interfaces

4.3.1 GUI Description

The main graphical interface of the implementation is developed by using Java Swing class. As shown in Figure 4.4, there are five panels present in the main fame. The top panel contains the menu bar and the middle one three sub-panels: *center panel*, *west panel*, and *east panel*. The *center panel* is the main area to display graphics. The *east panel* contains checkboxes, buttons, one text field and a text area to display coordinates of objects drawn in the *center panel* as shown in Figure 4.5.

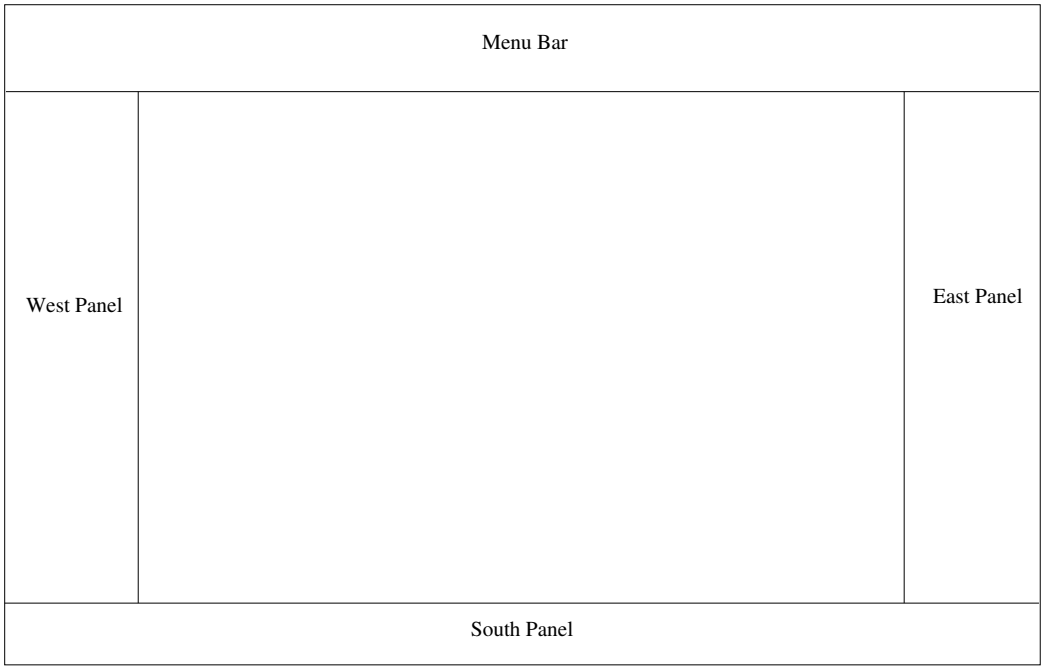


Figure 4.4: Layout of main user interface

On the *right panel*, the checkboxes are used to select operations such as *draw polygons*, *split segment*, *delete vertices*, *edit vertices* and *move polygons*. The *right panel* also contains seven buttons *Move Left*, *Move Right*, *Move Down*, *Move Up*, *Clear Canvas*, *Update Canvas*, and *Others* which are used for movement of the polygons as well as perform operations for clearing and updating the canvas. The text field below the buttons is used to provide the obstacle number to be drawn which usually starts from 0 for our convention. The lowermost part of the *right panel* has a text area which is used to show the number of obstacles constructed, the number of vertices in each obstacle, and

their corresponding coordinates.

The south panel on the bottom of the interface consists of two buttons: *AdjacencyGraph* and *ComputePaths*. The *Adjacency Graph* button can be used to construct adjacency graph implied by obstacles and the *Create Paths* button executes the display of *LowerChain*, *UpperChain*, and *BackChain*. There is a *west panel* on the left side which consists of choice buttons. These buttons can be used to select the default color to be used in the center panel. All of the panels are implemented by extending the *JPanel* class.

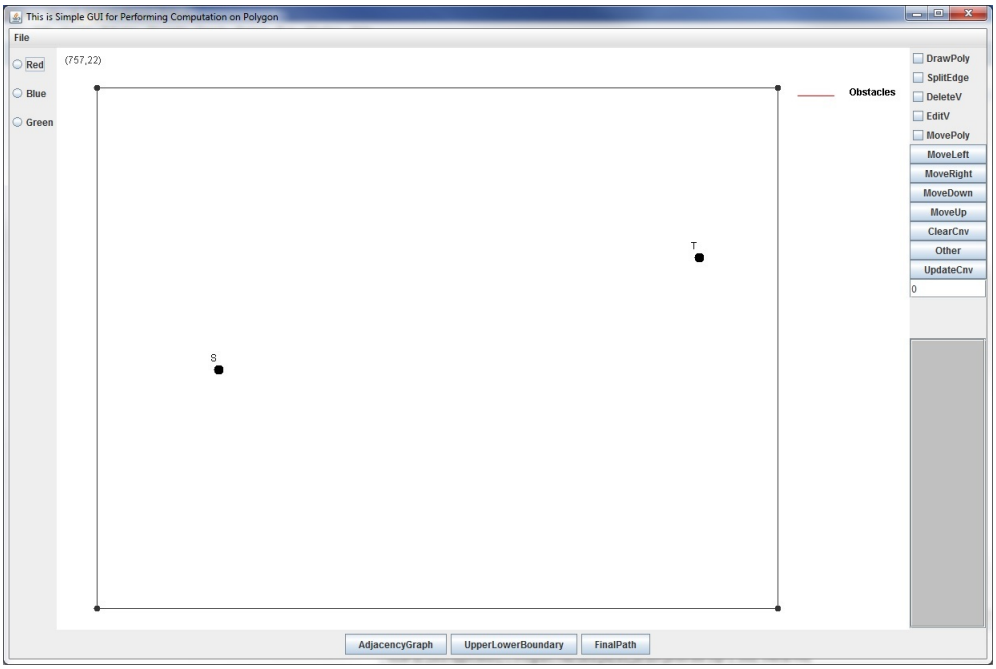


Figure 4.5: Main Graphical Interface

4.3.2 Interface Description

Figure 4.5 shows a snap-shot of the actual interface of the program. The file menu on the top panel allows users to (i) read an existing dcel file and (ii) save the dcel file to disk. A brief description of the functionalities of the file menu items is provided in Table 4.6. There are multiple check boxes and buttons in the GUI as mentioned earlier. Their corresponding functionalities are given in Table 4.7.

4.3.3 Snaps of Selected Interface States

As mentioned earlier, the obstacles can either be drawn directly on the canvas or load the dcel file containing previously drawn obstacles for applying various operations. In this section, we will show the snapshots of the output interfaces showing various functionalities of our java implementation of the project loading a dcel file.

As shown in Figure 4.6, a dcel file can be opened from the file menu and the obstacles with corresponding vertices (shown as black points) can be loaded.

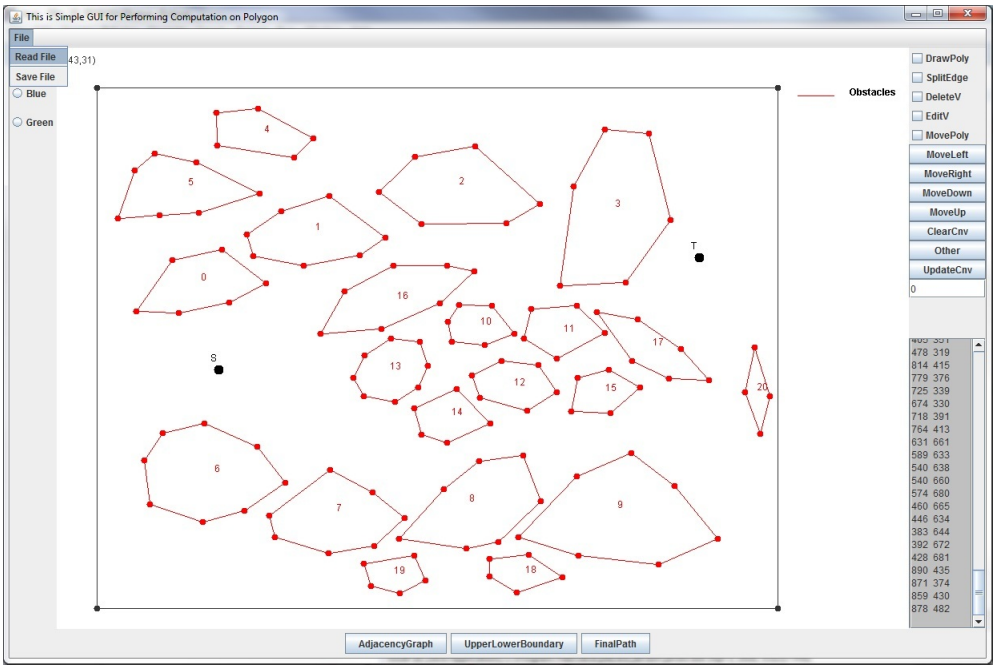


Figure 4.6: Loading input obstacle dcel file

By clicking the button *AdjacencyGraph* in the south panel, Vertical Adjacency Graph can be drawn. For avoiding the complexity in the canvas, we have only shown the vertices from which vertical graphs are generated as depicted in Figure 4.7.

In Figure 4.8, the construction of Upper and Lower Chain paths is shown which is generated by clicking the button *UpperLowerBoundary* in the south panel. The blue path in the upper section represents the Upper Chain path and the black path in the lower section represents the Lower

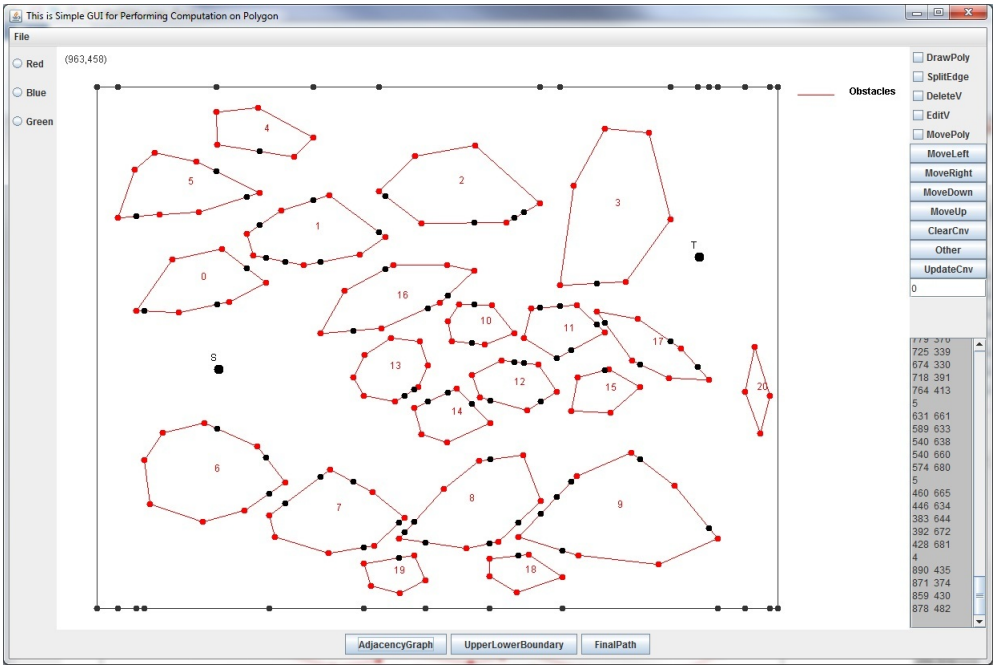


Figure 4.7: Showing Adjacency Graph Vertices

Chain path.

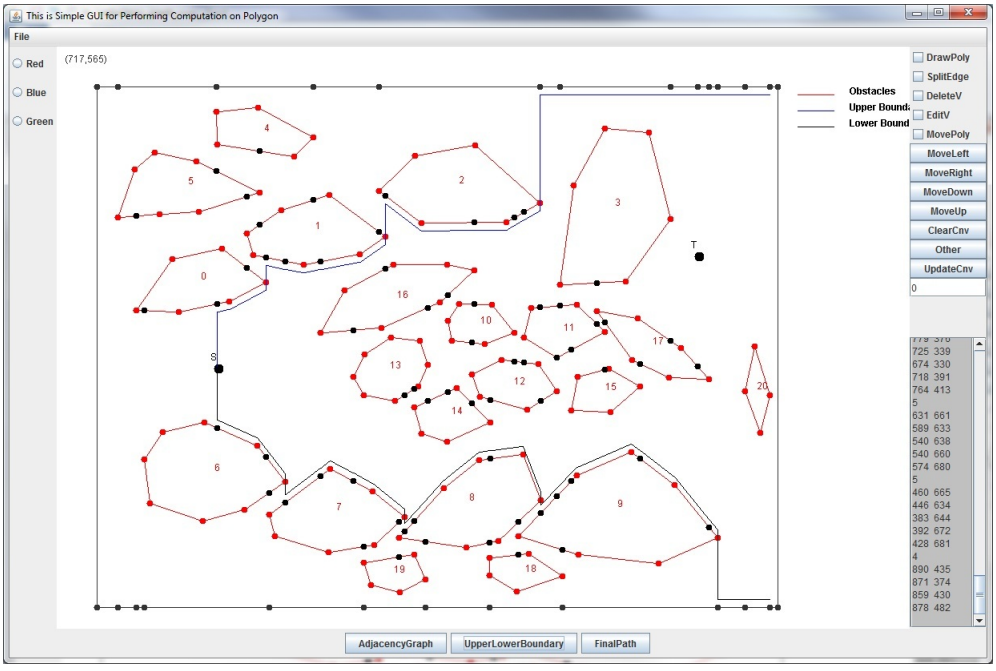


Figure 4.8: Drawing Upper and Lower Chain Paths

Figure 4.9 shows the construction of the back chain path from the target T to the source S as well as the final path from the source S to the target T. Both paths construction is generated by clicking the button *FinalPath*. The green path in the middle section depicts the back chain path and the orange path represents the final path.

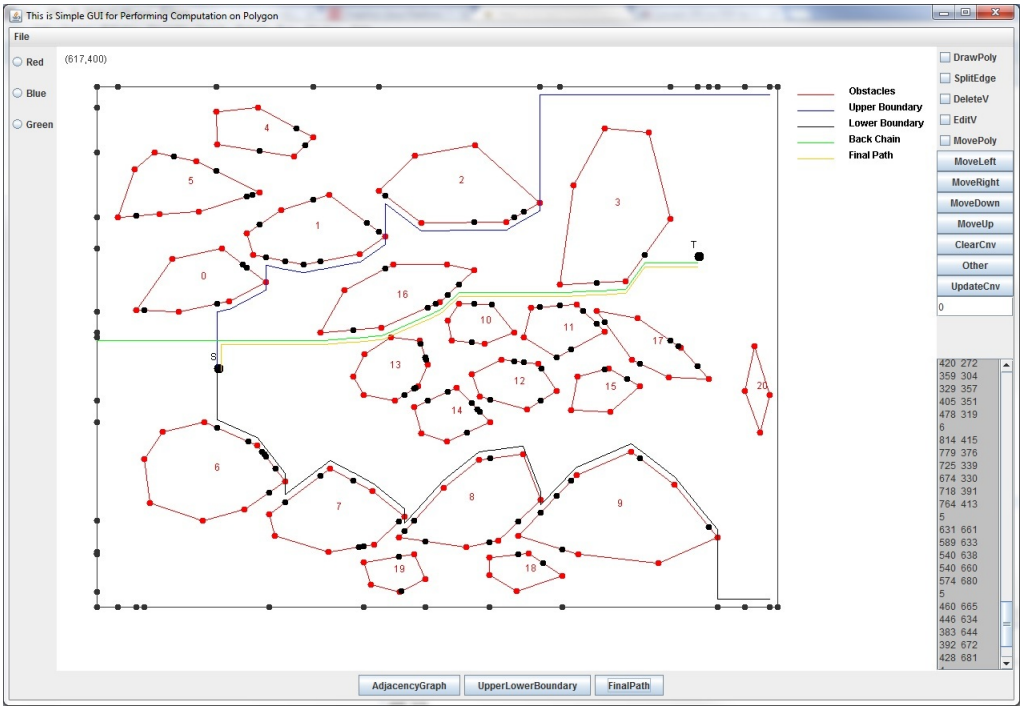


Figure 4.9: Final path from Source to Target and the Back Chain Path

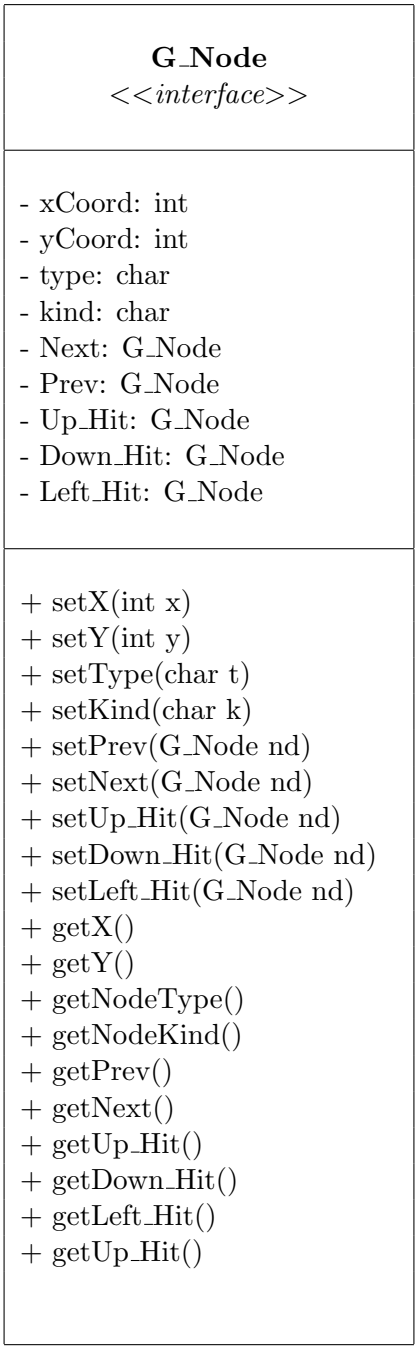


Table 4.3: Class Interface Diagram for G_Node

MyJPanel *<<interface>>*
- Source_Point: G_Node - Target_Point: G_Node - QQ: Vector<Vector <Point>> - BBB: Vector<Point> - GG: Vector<Vector <G_Node>> - G_BBB: Vector<G_Node> - U_B: Vector<G_Node> - L_B: Vector<G_Node> - B_P: Vector<G_Node>
+ processHitPointsFromAllVertices() + processHitPointsLeftFromAllVertices() + constructUpperBoundary() + constructLowerBoundary() + constructBackwardPath()

Table 4.4: Class Interface Diagram for MyJPanel

JGUI4 *<<interface>>*
- bt1 to bt10: JButton - cb1 to cb5: JCheckBox - rb1 to rb3: JRadioButton - eastPanel: JPanel - southPanel: JPanel - checkBoxButtonPanel: JPanel - rButtonPanel: JPanel
+ setUpMenuBar() + saveToFile() + readFromFile() + updatePolyPanel() + repaint()

Table 4.5: Class Interface Diagram for MyJPanel

S.N.	Menu Item	Functionalities
1	Read File	Open pop-up window to allow the user to select pre-saved file
2	Save File	Open pop-up window to allow the user to save file

Table 4.6: File Menu Items Description

S.N.	Menu Item	Functionalities
1	Draw Poly	Allow users to draw obstacles on the center panel
2	Split Edge	Allow users to split obstacle edge
3	Delete Vertex	Allow users to delete unwanted vertex
4	Edit Vertex	Allow users to edit or move vertex
5	Move Poly	Allow users to move individual obstacles

Table 4.7: Checkbox Items Description

Chapter 5

Conclusion and Discussion

As mentioned in Chapter 2, the problem of computing shortest Watchman path is NP-hard. This fact inspired us to look for heuristics for constructing Watchman path that satisfies additional requirements. The first requirement is that the path should be monotone. The second requirement is that the path should admit increased visibility.

We presented a critical review of important algorithmic results reported in literature to compute various versions of Watchman route problem. In particular, we examined how a shortest watchman route is computed in linear time in a rectilinear polygon.

We also examined the construction of collision-free paths in the presence of polygonal obstacles that are monotone in the given direction. As a main contribution, we formulated a problem for computing watchman path that is monotone in a given direction. We showed how visibility polygon for selected points on the path can be used to determine the overall visibility for the complete watchman path. To increase visibility from the path, we proposed to deform the path by adding funnel structures on selected position on the path. The deformed path indeed increases the visibility and remains monotone as before.

We also implemented algorithms that compute a collision-free monotone path from a source point S to a target point T in presence of polygonal convex obstacles. The monotone path is constructed by following the vertical adjacency graph left to right and hit points. We characterized region which we call *forbidden region* which can not be reached by monotone path starting from the fixed source vertex S. Our implementation allows the construction of monotone path by using the Waterfall

model.

The funnel extension heuristic we presented can be further improved by performing the following modifications. Instead of placing the funnels based on two criteria: (i) height of columns and (ii) values of layers, we could compute visibility from discrete set of points on the funnels as described in Section 3.3. Specifically, if the visibility from the selected points on a candidate funnel indeed covers additional area not visible from the initial S-T-monotone path, then that funnel could be selected for the deformation of the path. This approach looks computationally expensive but would generate better quality path.

Another avenue for further research would be to look for algorithm for generating shorter S-T-monotone path. The S-T-monotone path generated by using Waterfall model is not the shortest one. Making detours in the path generated by using the Waterfall model could shorten the length of the path. The turn angle implied by S-T-monotone path could have sharp turn upto $90°$. For many robotics application, this may not be acceptble. It would be interesting to look for designing S-T-monotone path that is constrained to have turn angle no more than given value θ.

Bibliography

[ACM90] E.M. Arkin, Robern Connelly, and J.S.B. Mitchell. "On Monotone Paths Among Obstacles with Application to Planning Assemblies". *Proceedings of the Fifth Annual Symposium on Computational Geometry*, pages 334–343, 1990.

[BKOS97] M. De Berg, M. Van Kreveld, M. Overmays, and O. Schwarzkopf. *"Computational Geometry: ALgorithms and Applications"*. Springer, 1997.

[CLRS09] T.H. Coreman, C.E. Leiserson, R.L. Rivest, and C. Stein. *"Introduction to Algorithm"*. The MIT Press, third edition, 2009.

[CN86] Wei-Pang Chin and Simeon Ntafos. "Optimum Watchman Routes". *Proceedings of the First Annual Symposium on Computational Geometry*, pages 24–33, 1986.

[CN88] W. Chin and S. Ntafos. "Optimum Watchman Routes". *Information Processing Letters*, 28(1):39–44, 1988.

[CN91] W. Chin and S. Ntafos. "Shortest Watchman Routes in Simple Polygons". *Discrete and Computational Geometry*, pages 9–39, 1991.

[CW15] D. Z. Chen and Haitoo Wang. "A New Algorithm for Computing Visibility Graphs of Polygonal Obstacles in the Plane". *Journal of Computational Geometry*, pages 316–345, 2015.

[GM91] S. B. Ghosh and D. M. Mount. "An Output Sensitive Algorithm for Computing Visibility Graphs". *Siam Journal of Computational Geometry*, 30(5):888–910, 1991.

[GN98] L. Gewali and S. Ntafos. "Watchman Routes in the Presence of a Pair of Convex Polygon". 105:123–149, 1998.

[LK08] F. Lee and R. Klette. *"An Approximate Algorithm for Solving Watchman Route Problem", Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[LK10] F. Lee and R. Klette. "Watchman Route in Simple Polygon with a Rubberband Algorithm". *Proceedings of the* 20^{th} *Canadian Conference on Computational Geometry*, pages 1–4, 2010.

[LPW79] T. Lozano-Perez and M.A. Wesley. "An Algorithm for Planning Collision-free Paths among Polyhedral Obstacles". *Communication of ACM*, pages 560–570, 1979.

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

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Bikash Lama Bamjan

Degrees:

Bachelor of Computer Engineering 2009

Tribhuvan University, Kantipur Engineering College, Nepal

Thesis Title: Algorithms for Monotone Paths with Visibility Properties

Thesis Examination Committee:

Chairperson, Dr. Laxmi Gewali, Ph.D.

Committee Member, Dr. Fatma Nasoz, Ph.D.

Committee Member, Dr. Justin Zhan, Ph.D.

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