

**Modern University for technology**

**and Information Faculty of Computers & Artificial intelligence Department of Information system**

**Shortest path planning in polygonal world for an Autonomous Mobile Robot**

Graduation project 2024

**Submitted by:**

Peter Hany Hosny Shafek

Mahmoud Ahmed Mohammed Tawfik

Hamed Mohamed Hamed

Ahmed Emad Saied Khllaf

Wesam Soltan Ahmed Soltan

Ahmed Tarek Saber Amin

Saleh Abdulhakim Saleh

**Supervised by:**

**Prof. Dr. Mahmoud Wahdan**

**TA. Osama Hassan**

**TA. Hazem Omar**

**ACKNOWLEDGMENT**

**Our greatest in deep and gratitude to Modern University for Technology and Information (M.T.I) for the great role it played in forming our character and personalities, providing us with all means necessary for our educational development and providing us as well with a wonderful environment to learn in throughout our educational years.**

**This would not have been possible unless Prof. Dr. Olfat Kamel, President of the**

**M.T.I University has established such a learning environment, providing the most effective facilities for our job to be done.**

**It is a pleasure to thank those who maneuvered this Possible, Prof. Dr. Mohamed Taher ElMayah Dean of the faculty of Computers and Information & Artificial intelligence for his unlimited help, support, encouragement and guidance.**

**A Special thanks to our head of the Information System Department Prof. Dr. Hafez Abd El-Wahab are Honored and got the pleasure of working with Prof. Dr. Mahmod Wahdan , and our teaching assistants TA. Osama Hassan , TA. Hazem Omar and TA. Ahmed Huzzain.**

**As well as the Department staff, Prof. Dr. Mohamed Answer Assal and Dr. Sayed Oraby for their effort along our university studies.**

**A Special thanks to our head of the Computer Science Department Prof. Dr. Hanafy Isamil, as well as Prof. Dr. Hesham El Deeb for their effort along our university studies.**

**Special thanks to Prof. Dr. Emain Taha, head of Basic Science Department. As well as the department staff, Prof. Dr. Elsayed Bakkar for their effort teaching us the basic for our computer science degree, Prof. Dr.Rasha Mohammad, Prof.Dr.Marwa Refaey and Prof.Dr.Rania Ahmed Lecture in Basic science Department.**

**Special thanks to our families who always supported us during our academic life, and we could not do it without them with us.**

# **Abstract**

This paper describes an algorithm to solve the shortest path planning for a point robot in a polygonal world. This is one of the most fundamental problems in robotics. In this paper, actually we discuss the most dangerous path considering the path length being the most important factor in robot path planning. This paper addresses the problem of shortest path through free space of a two dimensional, world model with convex polygonal obstacles from a start point to a goal point using the modified Dijkstra’s algorithm. In order to find the globally shortest path, first find the locally shortest path in each path class. Next, find the globally shortest path among those locally shortest paths. The approach divides the problem into four subproblems:

1. finding all the visible tangents from start point
2. finding all the visible tangents to goal point
3. finding all the visible common tangents among polygons
4. calling modified Dijkstra’s algorithm

*TABLE OF CONTENTS*

[**Abstract** 3](#_Toc170503246)

[**Table of Figures** 6](#_Toc170503247)

1. [INTRODUCTION 8](#_Toc170503248)

[1.1 OBJECTIVES 9](#_Toc170503249)

[1.2 BACKGROUND 9](#_Toc170503250)

[1.3 PROBLEM STATEMENT 11](#_Toc170503251)

[1.3.1 Definitions 11](#_Toc170503252)

[1.4 THESIS ORGANIZATION 14](#_Toc170503253)

2. [Project Planning 16](#_Toc170503254)

[2.1. Introduction 17](#_Toc170503255)

[2.2. Project Task Description 17](#_Toc170503256)

[2.3. Project Management 19](#_Toc170503257)

[2.4. Software Tools 22](#_Toc170503258)

[2.4.1 Project Subsystems 23](#_Toc170503259)

[2.4.2 Gantt Chart 23](#_Toc170503260)

[2.4.3. Network Diagrams 24](#_Toc170503261)

3. [POLYGONS 26](#_Toc170503262)

[3.1 Polygons and their Orientation 27](#_Toc170503263)

[3.2 Polygonal World 32](#_Toc170503264)

[3.3 Polygonal World and Path Class 34](#_Toc170503265)

[3.3.1 Directed v-edge 35](#_Toc170503266)

[3.3.2 Canonical Paths and Directed v-edges Sequences 38](#_Toc170503267)

[3.3.3 Connectivity Graph 42](#_Toc170503268)

[3.3.4 Path Class Representation 43](#_Toc170503269)

[3.4 Finding the Best Path Class 46](#_Toc170503270)

[3.5 Image of Point on Convex Polygon 46](#_Toc170503271)

[3.5.1 Visibility from Point to Polygon 48](#_Toc170503272)

[3.5.2 Type of an Image from a Point to a Convex Polygon 51](#_Toc170503273)

[3.5.3 The Image Type Algorithm 54](#_Toc170503274)

[3.5.4 Finding an Image on a Non-convex Polygon 57](#_Toc170503275)

[3.6 Path Classes and Sub-Polygons 59](#_Toc170503276)

[3.7 Advantages of Path Class Representation Using Directed V-Edges Sequences 63](#_Toc170503277)

4. [SHORTEST PATH PLANNING ALGORITHM 65](#_Toc170503278)

[4.1 CANONICAL PATHS AND TANGENT SEUENCES 66](#_Toc170503279)

[Interpreting Tangent Sequences into Paths and Visibility 67](#_Toc170503280)

[4.2 TANGENTS ON POLYGONS 71](#_Toc170503281)

[4.2.1 Tangents from point to polygon 72](#_Toc170503282)

[4.2.2 Tangents from Polygon to Point 75](#_Toc170503283)

[4.2.3 Common Tangents Between Two Polygons 76](#_Toc170503284)

[4.3 VISIBILITY 78](#_Toc170503285)

[4.4 SHORTEST PATH FINDING AGORITHM 81](#_Toc170503286)

[4.4.1 Comparison between Two Paths 87](#_Toc170503287)

5. [CONCLUSION 89](#_Toc170503288)

6. [System Design 91](#_Toc170503289)

[APPENDIX A. SOURCE CODE 95](#_Toc170503290)

[APPENDIX B. USER GIUDE 108](#_Toc170503291)

[REFREN‎CES 116](#_Toc170503292)

# **Table of Figures**

Figure 1.1 Canonical path…………………………………………......13

Figure 3.1 Interior and exterior angle of a simple polygon……………27

Figure 3.2 Convex and concave simple polygons………………...…...25

Figure 3.3 Cross product of vectors…………………………...………26

Figure 3.4 Intersecting segments ...........................................................28

Figure 3.5 Interior and exterior of a simple polygon .………...….….. 30

Figure 3.6 Polygonal world……………………………...……………... 30

Figure 3.7 Generalized Voronoi diagram of polygonal world (1) ...…. 31

Figure 3.8 Generalized Voronoi diagram of polygonal world (2) …..... 32

Figure 3.9 Defining directed v-edge for (ccw polygons) ................... 33

Figure 3.10 Defining directed v-edges for (cw & ccw) ………......... 34

Figure 3.11 Paths and canonical paths…………………………...……34

Figure 3.12 Interpretation of canonical path as directed v-edges sequence……………………………………………………………..36

Figure 3.13 Directed v-edges sequence [γ1]…………………………....37

Figure 3.14 Directed v-edges sequence [γ2].........................………...... 38

Figure 3.15 Basic connectivity graph of a polygonal world (1)……39

Figure 3.16 Basic connectivity graph of a polygonal world (2)……….40

Figure 3.17 Polygonal world (1)………………………………………40

Figure 3.18 Augmented connectivity graph of a polygonal world…….41

Figure 3.19 Polygonal world (II)………………………………………42

Figure 3.20 Augmented connectivity graph of a polygonal world (II)..42

Figure 3.21 Image of Point on convex polygon……………………….43

Figure 3.23 Image on object…………………………………………...44

Figure 3.25 Images on world…………………………………………..44

Figure 3.25 Visibility from point p to convex polygon B……………..45

Figure 3.26 Classifications of vertex vi of polygon B with respect to a segment …………………………………………………………. 46

Figure 3.27 Visibility from point p to convex polygon B……………..47

Figure 3.28 Images of point P lies on an edge of convex polygon B….49

Figure 3.29 Images of point P lies on vertex vi of convex polygon B...50

Figure 3.29 Image type block diagram………………………………...51

Figure 3.30 Correctness of image type algorithm..……………………53

Figure 3.31 Image of a point P on cw concave polygon B…………….54

Figure 3.32 Image of a point P on ccw concave polygon B…………...55

Figure 3.33 Problem1: initial orientation of a vehicle is different from the direction of a motion.………………………………………………57

Figure 3.34 Problem2: voronoi diagram of polygonal world consisting of two polygons (ccw polygon inside cw polygon boundary)………...58

Figure 3.35 solution of problem1: voronoi diagram of a sub polygonal world…………………………………………………………………...58

Figure 3.36 Basic connectivity graph of a sub polygonal world……....59

Figure 3.37 Augmented connectivity graph a sub polygonal world…..59

Figure 3.38 Solution of problem2: world and augmented connectivity graph………………………………………………………………….. 60

Figure 4.1 Examples into paths and visibility…………………………64

Figure 4.2 Tangent sequence A-B-C+ and its canonical path…..………65

Figure 4.3 Tangents………….…..….…………………………………69

Figure 4.4 Tangents from point to polygon…………………………....70

Figure 4.5 Tangents from polygon to point……………………………72

Figure 4.6 Tangents between two polygons…………………………...73

Figure 4.7 Visibility…………………………………………………...75

Figure 4.8 All tangents from s ………………………………………..76

Figure 4.9 All visible tangents………………………………………...78

Figure 4.9 All visible tangents………………………………………...80

Figure 4.10 Comparison of paths……………………………………...85

CHAPTER.1

# INTRODUCTION

* 1. OBJECTIVES

The primary objective of the work in this thesis was to develop an application program implementing the theoretical work being done by Professor Yutaka Kanayama, U. S. Naval Postgraduate School, Monterey California, in searching for shortest and safest paths among obstacle common tangents. Obstacle common tangents are one form of representing path equivalence classes, as well as providing a representation of locally shortest paths. The implementing this work equates to determining minimum cost paths for a robot moving in an environment that imposes certain geometric constraints. The main objectives of this thesis are the following: 1) Develop a two-dimensional world model capable of representing convex and concave polygons. 2) Construct a world tangent visibility graph providing symbolic representation of all common tangent situations. 3) Implement a searching method for the tangent vidibility graph using heuristic control to locate shortest path,,

## 1.2 BACKGROUND

There are two goals for planning autonomous vehicle navigation planning: shortest path and safe path. These goals are often in conflict. We will discuss the most dangerous path considering the path length being the most important factor in robot path planning.

The problem of planning the path of a robot around obstacles has attracted quite a bit of attention over the past several years. The two main types of path planning algorithm are graph searching and potential field. The graph searching techniques also fall into two main areas, those that determine a safe path by tracing around the obstacles and those that the free space, which is the complement of the space occupied by the obstacles.

The visibility graph [1] was an early technique for tracing around obstacles. By using the robot's configuration space, the problem of planning the path of the robot was reduced to planing the path of a point representing the robot .The visibility graph shows which vertices of the obstacles are visible from other vertices. A safe path is generated by piecing together several of these safe path segments. This algorithm is typical of the tracing based techniques.

Three typical free space searching techniques are: the generalized cones [2], the voronoi diagram [3], and the subdivision of the free space [4]. In the first technique, the free space is represented by generalized cones whose sides are the face for the obstacles. A safe path is produced by piecing together the center lines of the cones the voronoi diagram in two dimensions is a collection of curves that are equi-distant from two or more obstacles. A safe path is found by moving along these curves. In three dimensions, the curves become surfaces that are equidistant from two or more obstacles. In the subdivision algorithms, the space is subdivided until a region is either completely filled with an obstacle or the region is empty. A safe path is then determined by moving from one free region to another to the goal. The A\* algorithm [5] is frequently employed for searching the subdivided free space.

Artificial potential field algorithm [6] uses repulsive potential fields around the obstacles to force away the robot, which is subjected this potential, and an attractive potential field around the goal to attract the robot. Another potential field algorithm [7] begins with a trial path through the space and uses repulsive potential fields around the obstacles to modify the path so that it moves toward the free space until a safe path is found. The difference to note here is that the potential field effects the entire path simultaneously as opposed to the current robot position. This was shown to be less sensitive to local minima in the potential field than other potential field based algorithms.

The most effective of the free space algorithms, in terms path length optimization and thoroughness, is the A\* technique [5]. However, it has a tendency to be slow planner due to the amount of space that is searched. Other artificial features are that is easily extended to higher dimensions and obstacles are not restricted to nice, geometric shapes If there was away to increase the speed for this algorithm, it could be ideal for real-time path planning.

An unrelated technique that is quite fast is the vector based method [8]. In this technique, a trial vector is drawn from the starting position (s) to the goal (g). If the vector across an obstacle, a new intermediate goal (IG) is created and the previous goal is pushed on to a stack. The intermediate goal is obtained by drawing a new vector from the centroid (c) of the crossed obstacle through the midpoint (m) of the vector segment that crossed the obstacle. This vector is continued until it is clear of the obstacle plus a small extension for margin of safety. This point is used as an intermediate goal. The planning procedure is then repeated until there is a safe path to an intermediate goal. At this time, the starting point is moved to the intermediate goal is popped from the stack. This procedure continues until the final goal is reached, thus yielding a safe path from the original starting point this brief survey is by no means complete, but it does show some of the typical algorithms. Techniques have also been presented for avoiding moving obstacles [9] and obstacles with uncertainty [10].

## 1.3 PROBLEM STATEMENT

### 1.3.1 Definitions

This subsection defines a list of terms and concepts used throughout this project.

Let ℜ denote the set of real numbers. The environment for this task of this project is a two-dimensional plane ℜ2 on which a global Cartesian coordinate system is defined.

Let *B*i, … ., *B*n be fixed objects (convex polygons) distributed in ℜ2. These *B*i’ s are called *obstacles*.

A *world* *W* is a set of *n* convex polygonal obstacles,

*W* = {*B*0, *B*i, … ., *B*n}, *n* > 0

where *B*0 is the outermost polygonal boundary,  *B*i, … ., *B*n are polygonal obstacles inside the boundary, and no pair of polygons intersects or touches.

The *free space* free(*W*) is the inside of *B*0 minus the union of the *n* polygons contained in *B*0. In other words, the free space is the complement of the union of all polygons in *W.*

We consider path *f* to be directed curve with natural direction from *f*(0) to *f*(1). A *path* *f* in *W* is a continous function

*f* : [0, 1] → free(*W*)

with *f*(0) ≠ *f*(1). The two points *f*(0) and *f*(1) are called its *endpoints*, and the path *joins* them. If they are distinct, we usually denote *f*(0) as a start *S* and *f*(1) as a goal *G*.

* + 1. **Problem Description**

The purpose of this project is to find an algorithm to solve the shortest path planning problem for a point robot in a polygonal world. This is one of the most fundamental problems in robotics. In this project, actually we discuss the most dangerous path considering the path length being the most important factor in robot path planning.

In this project, a world *W* which consists of *n* convex polygons, and two points *S* and *G* in free (*W*) are given. (Notice that *W* contains no CW polygons.) We want to find the shortest path in free (*W*) U *W* connecting the two points (see Figure 1).

In order to find the (globally) shortest path, we first find the locally shortest path in each path class. Next we will find the globally shortest path among those locally shortest paths. Only “locally shortest paths” in path classes. A locally shortest path is called “canonical path” (see Figure 1). By this concept, we will obtain an efficient algorithm for this problem.

**Canonical Paths:**

* **Locally Shortest Path:** A path within a specific "path class" (explained below) that is the shortest path for that particular class.
* **Path Class:** A categorization of possible paths based on how they interact with the obstacles. Different path classes might involve the robot going around obstacles entirely, partially entering some obstacles, or completely traversing specific obstacles (depending on the project's specific definition).
* **Canonical Path:** By finding the locally shortest path in each path class and then comparing them, the algorithm aims to identify the **globally shortest path** that connects S and G.

**Overall Approach:**

1. **Classify Possible Paths:** Divide potential paths into different categories based on their interaction with obstacles (path classes).
2. **Find Locally Shortest Paths:** Within each path class, identify the path with the minimum length that connects S and G.
3. **Compare Locally Shortest Paths:** Compare the lengths of the locally shortest paths from each class.
4. **Identify Globally Shortest Path:** The path with the minimum length among all the locally shortest paths is considered the globally shortest path for the robot to travel from S to G.

**Benefits of Canonical Paths:**

* This approach might lead to an **efficient algorithm** for finding the shortest path by focusing on specific path categories instead of exploring all possible paths blindly.

**Additional Notes:**

* The specific details of path classification and how the algorithm efficiently searches within each class are likely explained in the full project description.
* This project focuses on theoretical concepts rather than a physically realistic scenario where the robot can't pass through obstacles.

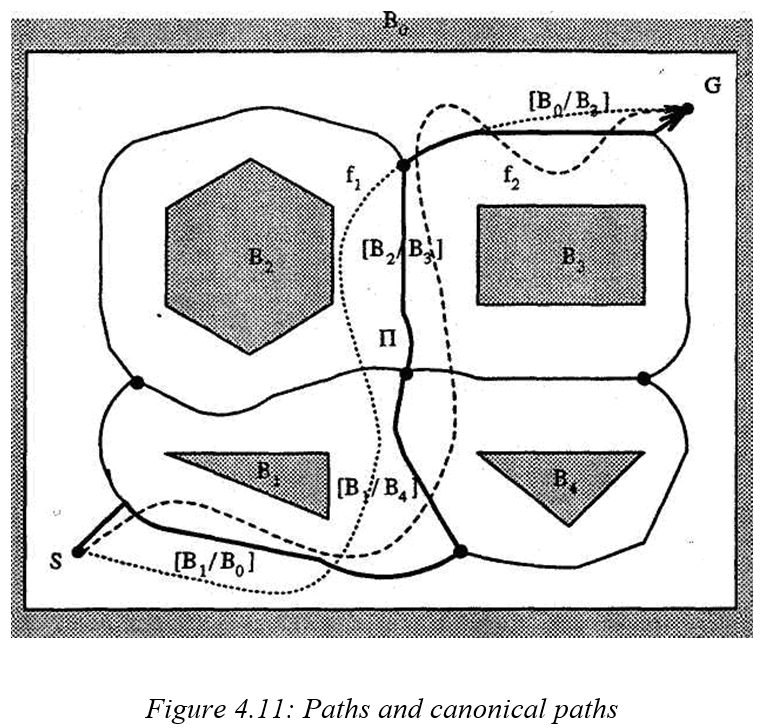


Figure 1.1 canonical path

## 1.4 THESIS ORGANIZATION

The concepts which are introduced and applied in this thesis deal with finding globally shortest paths through a two-dimensional. orthogonal world, represented as distinct convex and concave polygons. The first three chapters of this work detail much of the as yet unpublished work of Professor Yutaka Kanayama on searching for shortest and safest paths. I The mathematical foundations for path planning using obstacle common tangents is presented in considerable detail in Chapter I1, beginning with the simple geometric relationships between points, polygons and lines in the two-dimensional plane. The construction and identification of obstacle common tangents is explained, and we introduce the convex sUbpolygon; a very simple and elegant method we use to represent and process concave polygons, allowing all obstacles to be processed in a standardized symbolic manner. Chapter III addresses the efficient visibility testing between two points, interpretation of tangent sequences, and links paths along xommon tangents to path equivalence classes and locally shortest paths. We conclude the chapter by outlining the necessary tools for developing efficient searching algorithms using pre-processed obstacle tangent visibility graphs. Chapter IV completes the theoretical background by describing the process of building constant clearance paths along obstacle common tangents, yeilding shortest safest paths for robot vehicles. In Chapter V. the implementation of optimal path searching among obstacle common tangents is detailed. A working application is presented for constructing obstacle tangent visibility graphs. as well as path finding through the tangent visibility graph using heuristic control of it modified Dijkstra search. A detailed account of this implementation in Common Lisp is given, as well as the graphical screen displays showing the shortest path searching process. Chapter VI addresses the contributions of this work, and outlines the areas where additional research effort is needed

CHAPTER.2

# Project Planning

.

**2. Project Plan**

## 2.1. Introduction

This chapter is a discipline for saying how to complete a project within a certain timeframe, usually with defined stages, and with designated resources.

## 2.2. Project Task Description

The project includes the following tasks:

* Defining Problem
* Gathering requirements
* System Implementation
* Testing
* Documentation
* Demo
* Final presentation

The description of the above tasks is written below **:**

* **Defining Problem :**

In this task, you will articulate the specific challenge of finding the shortest path for an autonomous robot navigating a polygonal world. This includes:

* Describing the environment, such as the types of polygons and their configurations.
* Outlining the primary goal : guiding the robot to navigate efficiently from a start point to a goal point within the polygonal space.
* **Gathering requirements**

This task involves collecting detailed information on the needs and constraints of the path planning system. Key activities include:

1. Defining functional requirements, such as the ability to handle various polygonal shapes, static obstacles, and different start and end points.
2. Specifying non-functional requirements like computational efficiency
3. Consulting with stakeholders, such as engineers and end-users, to gather input on system needs.
4. Reviewing existing algorithms and techniques for path planning to identify best practices and potential improvements (such as dijextra algorithm ,A\* algorithm ,bellman ford algorithm )

* **System Implementation**

in this phase, you will design and develop the path planning system for the robot. This involves:

* Selecting appropriate algorithms, such as Dijkstra's or visibility graph methods, tailored to a polygonal world.
* Designing the system architecture, including modules for environment representation, path computation
* Implementing the system in a suitable programming environment.
* **Testing**

Testing ensures the path planning system functions correctly and efficiently. This includes:

* Developing test cases to cover various scenarios, including different polygonal environments and obstacle configurations.
* Performing unit tests to validate individual components, such as obstacle detection and path calculation.
* Conducting integration tests to ensure the system works as a whole.
* Collecting performance metrics to assess the system's efficiency.
* **Documentation**

Comprehensive documentation is crucial for understanding and maintaining the path planning system. This includes:

* Creating user manuals detailing how to operate the robot and use the path planning system.
* Documenting the system design, algorithms used, and implementation details.
* Recording test plans, cases, and results to provide a thorough testing history.
* Ensuring all documentation is clear and accessible to both technical and non-technical stakeholders.
* **Demo**

The demo showcases the capabilities of the path planning system to stakeholders. This involves:

* Preparing a presentation highlighting the key features and functionalities of the system.
* Using different scenarios to showcase the system's versatility .
* Gathering feedback for further refinements and improvements.
* **Final presentation**

The final presentation summarizes the entire project, emphasizing key achievements and outcomes. This includes:

1. Presenting the problem definition and its importance in the context of autonomous robots.
2. Summarizing the requirements gathered and how they were addressed.
3. Explaining the system implementation, including design choices and algorithms used.
4. Sharing testing procedures, results, and performance metrics.
5. Discussing challenges faced and how they were overcome.
6. Providing recommendations for future work and potential enhancements to the system.

## 2.3. Project Management

*Table 2-1 Project Participants*

|  |  |
| --- | --- |
| Student No. | Student Name. |
| 1 | Peter Hany Hosny |
| 2 | Mahmoud Ahmed Mohamed Tawfiq |
| 3 | Hamed Mohamed Hamed Abdel Hamed |
| 4 | Ahmed Emad Saeed Khallaf |
| 5 | Wesam Sultan Ahmed Sultan |
| 6 | Ahmed Tariq Saber Amin Ali |
| 7 | Saleh Abdulhakim Saleh Babakri |

*Table 2-2 Project Tasks*

|  |  |  |
| --- | --- | --- |
| Task No. | Task  Name | date |
| 1 | Defining Problem | October 1 – October 5 , 2023 |
| 2 | Gathering requirements | October 6 – October 12 , 2023 |
| 3    4  5  6 | System Analysis & Design   * Path planning problem, the most important classical methods * Polygons and theory of a free space * Canonical path and tangent * Shortest path finding algorithm | October 13 – October 26 , 2023  October 27 – December 21 , 2023  December 22 – February 15 , 2024  February 16 – April 11, 2024 |
| 7 | System Presentation | April 26 – May 2 , 2024 |
| 8 | System Testing | May 3 - May 9 , 2024 |
| 9 | System Documentation | May 10 - May 16 , 2024 |
| 10 | System Demo | May 17 - May 23 , 2024 |

*Table 2-3 Project Task Assignment*

|  |  |  |
| --- | --- | --- |
| Task NO . | Duration | Student NO. |
| 1 | 1 | All Student |
| 2 | 1 | All Student |
| 3 | 2 | All Student |
| 4 | 8 | All Student |
| 5 | 8 | All Student |
| 6 | 8 | All Student |
| 7 | 1 | All Student |
| 8 | 2 | All Student |
| 9 | 1 | All Student |

A network diagram for the project showing estimated times for each activity and the earliest and latest expected completion time for each activity.

A diagram of a network

Description automatically generated

*Figure 2-1 Critical Path*

*Table 2-4 Activity Slack Management*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task | Preceding Task | Expected Duration | Early Finish | Late Finish | Slack Time | Critical Path |
| 1 |  | 1 | 1 | 1 | - | Yes |
| 2 | 1 | 1 | 2 | 2 | - | Yes |
| 3 | 2,1 | 2 | 4 | 4 | - | Yes |
| 4 | 3,2 | 8 | 12 | 12 | - | Yes |
| 5 | 4,3 | 8 | 20 | 20 | - | Yes |
| 6 | 5 | 8 | 28 | 28 | - | Yes |
| 7 | 3,4,5,6 | 1 | 29 | 29 | - | Yes |
| 8 | 3,4,5,6 | 2 | 30 | 30 | - | Yes |
| 9 | 8 | 1 | 31 | 31 | - | Yes |

## 2.4. Software Tools

Microsoft Project is a powerful project planning software developed by Microsoft. It helps users create project schedules, define tasks, allocate resources, and track progress. With features like Gantt charts, resource management, and collaboration tools, it supports efficient project management and ensures successful project completion. The task made by (Microsoft Project) Microsoft Project can:

• Draw the action plan and represent it on both the Network Diagram and the Gantt chart. • Allocate and organize resources for each activity.

• Follow-up progress of the project.

• Project budget management, and workload analysis.

• The possibility of programming work - as in all programs Microsoft Office package

**A green and white sign with arrows

Description automatically generated**

### 2.4.1 Project Subsystems

A screenshot of a calendar

Description automatically generatedGroup of interconnected and interactive parts that perform an important job or task as a component of a larger system. A subsystem, while a system, is also wholly contained within a larger system.

*Figure 2-2 Project Sub System*

### 2.4.2 Gantt Chart

A Gantt chart commonly used in project management is one of the most popular and useful ways of showing activities (tasks or events) displayed against time. On the left of the chart is a list of the activities and along the top is a suitable time scale. Each activity is represented by a bar; the position and length of the bar reflect the start date, duration, and end date of the activity.

A screenshot of a computer

Description automatically generated

*Figure 2-3 Gant Chart*

### 2.4.3. Network Diagrams

A Network Diagram is a visual representation of a project’s schedule. Wellknown complements to network diagrams. A network diagram in project management is useful for planning and tracking the project from beginning to finish. It represents a project’s critical path as well as the scope for the project

A screenshot of a computer screen

Description automatically generated

*Figure 2-4 Network Diagram*

A screenshot of a computer

Description automatically generated

*Figure 2-5 Network Diagram*

CHAPTER.3

# POLYGONS

## 3.1 Polygons and their Orientation

Polygons, Vertices, Edges, and Navigation in a Simple Polygon

This text defines key concepts related to polygons and introduces functions for navigating between their vertices:

**Polygons:**

* A polygon (denoted by B) is a closed shape formed by connecting a sequence of n points (v₁ to vₙ) in the plane, where n ≥ 3.
* **Vertices (vi):** The points that define the corners of the polygon.
* **Edges (vi vi+1):** Line segments connecting consecutive vertices.

**Polygon Representation:**

* B = {v₁,..., vₙ} (Equation II.1): A polygon is defined as a set of vertices in a specific order. The order matters because changing it can result in a different shape.

**Types of Polygons (based on n):**

* Triangle (n = 3)
* Quadrilateral (n = 4)
* n-gon (n sides)

**Simple Polygons:**

* A simple polygon is one whose path doesn't intersect itself. This means:
  + No consecutive edges are on the same line (i.e., any three consecutive vertices are not colinear).
  + No two edges intersect except at their common vertex (consecutive edges meet at a corner).
* Figure 2a shows simple quadrilaterals, while Figure 2b (not provided) is a non-simple one (likely with intersecting edges).

**Next Vertex Function (j(vi)):**

* This function determines the next vertex in the sequence B after a given vertex vi.
* j(vi) = vi+1 if 1 ≤ i ≤ n-1 (i.e., for all vertices except the last one).
* j(vₙ) = v₁ (the last vertex connects back to the first).
* In simpler terms, j(vi) gives you the vertex that comes after vi when following the polygon's boundary in a counter-clockwise direction (assuming we follow a standard convention).
* Example: In Figure 2a (not provided), j(v₁) is v₂ and j(vₙ) is v₁.

**Previous Vertex Function (j⁻¹(vi)):**

* This function (inverse of j) gives the vertex that comes before vi in the sequence B.
* Proposition II.1 states that j is a bijection (one-to-one correspondence), so its inverse exists.
* The meaning of j⁻¹(vi) is the "previous vertex" of vi.
* Example: In Figure 2a (not provided), j⁻¹(v₁) is v₄ (assuming counter-clockwise traversal).

**Interior Angles (bi):**

* The angle formed inside the polygon at vertex vi.
* In any n-gon, the sum of all interior angles equals 180(n - 2) degrees.
* Example: A triangle's interior angles add up to 180 degrees.

**Exterior Angles (di):**

* The angle formed outside the polygon at vertex vi, formed by extending one edge and the next edge.
* Represented by di (see Figure 3, not provided).

**Direction (γ(vi, j(vi))):**

* This represents the direction from vertex vi to its next vertex j(vi).
* It likely refers to the angle or orientation in which you move from vi to j(vi) while following the polygon's boundary.

This explanation clarifies the concepts of vertices, edges, simple polygons, and how to navigate between them using the j and j⁻¹ functions. The introduction of angles and direction provides a foundation for further analysis of polygon properties and potential calculations involving rotations or movements within the polygon.

ψ(*v*2, *v*3)

*v*3

*v*4

*v*1  δ2

β2  *v*2 ψ(*v*1, *v*2)

Figure 3.1 Interior and exterior angle of a simple polygon

*Definition*: Given two distinct points, *p*1 ≡ (*x*1, *y*1) and *p*2 ≡ (*x*2, *y*2). We define a direction function ψ(*p*1, *p*2) as

ψ(*p*1, *p*2) ≡ *atan*2(*y*2 – *y*1, *x*2 – *x*1) (II.3)

The *exterior* angle, δI, at *v*i is the angle between one side and the extension of the adjacent side related to *v*i [11] (see Figure 3).

δi = Φ(ψ(*v*i, ϕ( *v*i)) - ψ(ϕ -1(*v*i), *v*i )) (II.4)

*Definition*: A vertex *v*i on a simple polygon is said to be a *convex* vertex if δi > 0. If δi <0, a vertex *v*i is said to be *concave* vertex.

For example, in Figure 4, in part (a), the vertex *v*2 is convex because δ2 > 0. In part (b), the vertex *v*3 is concave because δ2 < 0.

*Definition*: A simple polygon is a *convex* polygon if all of its vertices are convex (Figure 4(a)), otherwise it is *nonconvex* polygon (Figure 4(b)).

*v*4 *v*7

*v*9

*v*8

*v*3  *v*10 *v*6

*v*5 *v*4

*δ*3 *δ*4

*v*3

*δ*2

*v*1 *v*2 v1 v2 v5

(a) Convex Simple Polygon (b) Concave Simple Polygon

Figure 3.2 Convex and concave simple polygons

Now, we will define three important predicate, *ccw* (counterclockwise), *cw* (clockwise) and *col* (colinear). Consider vector u = (*x*1, *y*1)T and v = (*x*2, *y*2)T, shown in Figure 5(a). The *cross product* u x v can be interpreted as the signed area of the parallelogram formed by the points (0, 0), *u*, *v*, and *u* + *v* = (*x*1 + *x*2, *y*1 + *y*2). An equivalent, but more useful, definition gives the cross product as the determinant of a matrix.

*y* u + v

v ccw u

(0, 0) cw

u

(0, 0) *x*

(a) (b)

Figure 3.3 Cross product of vectors

*x*1  *x*2

u x v =

*y*1  *y*2

= *x*1 *y*2 - *x*2 *y*1

= - v x u (II.5)

If u x v is positive, then u is clockwise from v with respect to the origin (0, 0); if this cross product is negative, then u is counterclockwise from v. Figure 5(b) shows the clockwise and counterclockwise regions relative to a vector u. A boundary condition arises if the cross product is zero; in this case, the vectors are collinear, pointing in either the same or opposite directions.

The above discussion is very useful for all results related to the area of the polygon.

The *area* of a polygon whose vertices *v*i have coordinates (*x*i, *y*i), for 1 ≤ *i* ≥ *n*, is the “signed” value of

*Area*(*B*) = ½ (*x*1 *y*2 - *x*2 *y*1) + …. + ½ (*x*n-1 yn - *x*n yn-1) + ½ (*x*n y1 - *x*1 yn)

*n*

= ½ Σ (*x*i yi+1 - *x*i+1 yi)

i=1

In particular, for a triangle *B* = {*v*1, *v*2, *v*3} = {(*x*1, *y*1), (*x*2, *y*2), (*x*3, *y*3)}, the “signed” area is defined as

1 1 1

Δ = ½ *x*1 *x*2 *x*3 (II.6)

*Y*1  *y*2  *y*3

= ½ ((x2 – x1) (*y*3 – *y*1) – (*x*3 – *x*1) (*y*2 – *y*1))

**Proposition III.1** *For any triangle B,*

1. If Δ > 0, *B is ccw and area of B is equal to* Δ.
2. If Δ < 0, *B is cw and area of B is equal to* Δ .
3. If Δ = 0, *B is col and area of B is* = 0.

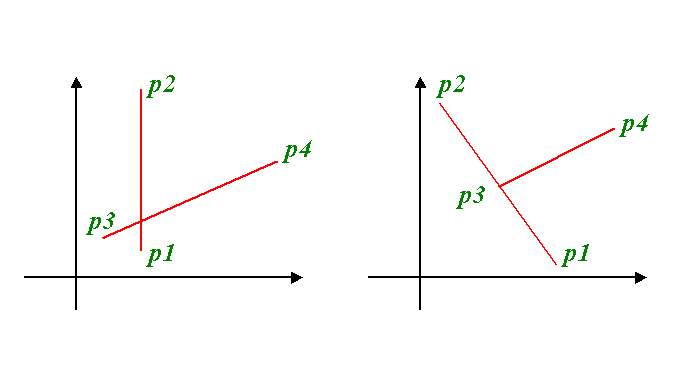
*Definition*: A convex polygon is a polygon whose ordered list of vertices produces a counterclockwise (ccw) boundary loop. An nonconvex polygon is a polygon whose ordered list of vertices produces a clockwise (cw) directed boundary loop.

Figure 3.4 Intersecting Segments

Let us show how the orientation concept is useful in solving geometrical and robotics problems by giving an example. Any two distinct points in a plane define a *segment*. A *closed segment* includes the endpoints, but an *open segment* does not. Two segments are said to *intersect* if they share only one point (thus, if they are sharing a part or all of the segments, we do not claim they are intersecting). The problem of testing whether two (open or closed) segments are intersecting is answered by the orientation function (Figure 6)

**Proposition II.2** *A necessary and sufficient condition for two open segments p*1*p*2 *and p*3*p*4*, to intersect each other is*

[*O* (*p*1, *p*2, *p*3)= - *O* (*p*1, *p*2, *p*4) ≠ 0 ]∧[*O (p*3*, p*4*, p*1)= - *O* (*p*3, *p*4, *p*2)≠0 ] (II.7)

**Proposition II.3** *A necessary and sufficient condition for two closed segments p*1*p*2 *and p*3*p*4*, to intersect each other is*

[*O* (*p*1, *p*2, *p*3)≠  *O* (*p*1, *p*2, *p*4) ]∧[*O (p*3*, p*4*, p*1)≠ *O* (*p*3, *p*4, *p*2) ] (II.8)

The computational costs of these tests are constant and inexpensive. If two open segments intersect, so do the closed segments. However, its converse is not true.

## 3.2 Polygonal World

**Simple Polygon and Plane Partitioning:**

* A simple polygon, as defined earlier, doesn't have any self-intersections and separates the plane into two distinct areas.
* This separation is based on the Jordan Curve Theorem (not explicitly mentioned here but implied).

**Regions of a Simple Polygon (Figure 7):**

* **Interior (int(B)):** The set of points **enclosed** by the polygon. This is the finite area "inside" the polygon's boundary.
* **Boundary (B):** The set of points that form the actual **edges** of the polygon.
* **Exterior (free(B)):** The set of points surrounding the polygon. This is the infinite area "outside" the polygon.

**Classification of Simple Polygons (ccw vs. cw):**

* The text introduces a classification system for simple polygons based on how their "free side" (exterior) is defined:
  + **Counter-clockwise (ccw) polygon:**
    - The exterior (free(B)) of the polygon is the area to the **right** of the boundary when following the polygon's vertices in a counter-clockwise direction.
  + **Clockwise (cw) polygon:**
    - The exterior (free(B)) of the polygon is the area to the **left** of the boundary when following the polygon's vertices in a clockwise direction.

**Key Points:**

* This classification system is crucial for various geometric calculations and algorithms that interact with polygons. It helps determine which side of the polygon is considered "outside" based on the direction you traverse its boundary.

**Additional Notes:**

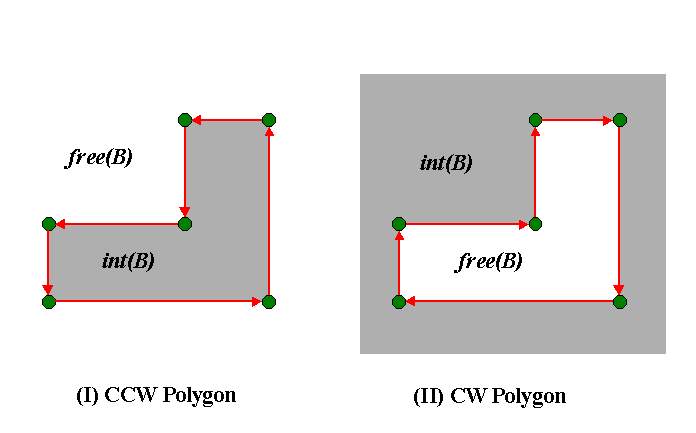
* The text doesn't explicitly mention, but the "left" and "right" definitions for exterior might be based on a specific convention where the polygon is drawn on a plane with a positive x-axis to the right and a positive y-axis upwards.
* In some contexts, the "interior" might also be referred to as the "hole" of the polygon.

Figure 3.5 Interior and exterior of a simple polygon

## 3.3 Polygonal World and Path Class

**B1**

**B2**

**B0**

*Figure 3.6: Polygonal world*

Consider a world W which consists of a finite number of polygons, i.e.,

W= {B0,B1,….,Bn}, n > 0

Where *Bo* is a *cw* polygon, and *ccw* polygons *B1, ……*, *Bn* are considered to be obstacles for the robot (see Figure 4.6).

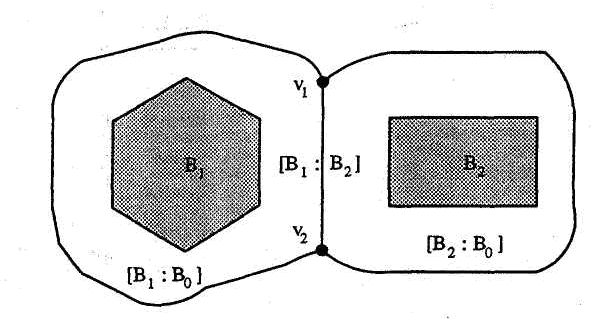
For a point *p*  free (W), the distance *d(p,Bi)* from *p* to a polygon *Bi* is defined in Eq5.2. The Voronoi region *V(Bi)* of a polygon Bi in W is defined as

V(Bi) = {*p* *free*(W) | (for all j) [(i ≠ j ^ 1≤ *j≤ n*) 🡪[*d*(*p,*Bi) < *d(p,*Bj)]]} (4.l)

For instance, Eq.4.l means that any point within free (W) has its image on the two polygons. The Voronoi diagram of world W consisting of three polygons is shown in (figure 4.7), and the Voronoi boundaries of W consists of line segments and parabolic arcs. Note that the intersection point of three or more Voronoi boundary segments is called a   
*v-node.* A Voronoi boundary segment(s) between two v-nodes is called a *v-edge.* For example, there are two v-nodes and three v-edges as shown in Figure 4.7.

Each undirected v-edge £ is the boundary of two Voronoi regions, *V{Bi)* and *V(Bj).* We denote an undirected v-edge £ by£ ***= [Bi* : *Bj]****,*

**B0**



*Figure 3.7 : Generalized Voronoi diagram of polygonal world (1)*

Where [Bi : Bj] and [Bj : Bi] are considered the same. For example, in Figure 4.7, the undirected v-edge between the two v-nodes v1 and v2 is £ = [B1 : B2] or  
 £ = [B2 : B1]. Also, there are three undirected v-edges [B1 : B0], [B1 : B2], and   
[B2 : B0]. Another example is shown in (figure 4.8). In which, a world W consists of five polygons B0, B1, B2, B3 and B4. There are five v-nodes and eight undirected   
v-edges [B1 : Bo], [B1 : B2], [B2 : B0],[B2: B3], [B1 : B4], [B4 : Bo],[B3 : B4], and [B3 : B0].

### **3.3.1 Directed v-edge**

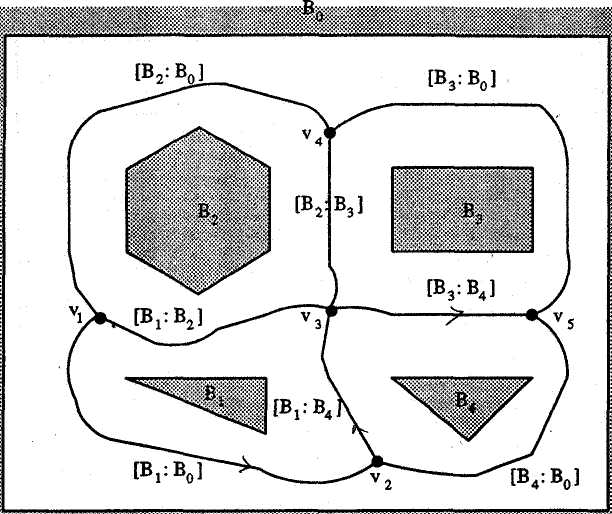
Each undirected v-edge is the boundary of two Voronoi regions, *V(Bi)* and *V(Bj).* In this case,

[Bi: Bj] ≠ [Bj: Bi].

Now, we consider the *directed* v-edge. Once the directed v-edge is given, the concepts of left and right images take on meaning. This will aid in using the world data to capture the spatial relationship between the objects in the world. We have two types of directed boundaries:

* Directed boundaries of two polygons that are the same *(ccw):*

There are two opposite directions on an undirected v-edge [Bi : Bj]. One direction goes ccw with Bi and cw with Bj. The other direction goes cw with Bi and ccw with Bj (Figure 4.9).



*Figure 3.8: Generalized Voronoi diagram of polygonal world (2)*

* Directed boundaries of two polygons that are different *(ccw* and *cw):*

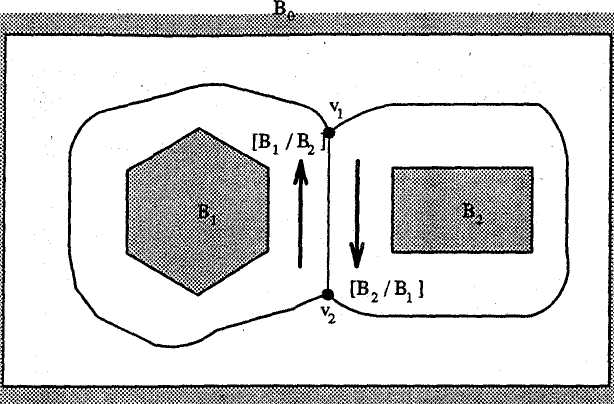
There are two opposite directions on an undirected v-edge *[Bi : Bj].* One direction goes *ccw* with *Bi* and *cw* with *Bj.* The other direction goes *cw* with *Bi* and *ccw* with *Bj* (Figure 4.10).

Now, we denote *directed* v-edge *ξ* by

*ξ =* [Bi / Bj],

where *Bi* and *Bj* refer to the left and right polygons respectively. It is clear that [Bi/Bj] and [Bj/Bi]are not the same.

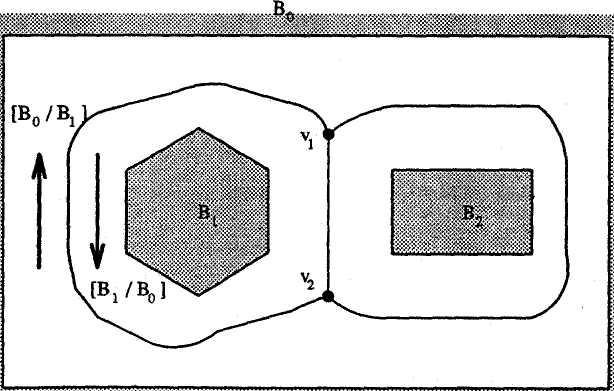
**NOTE : From the above we see that we don't have to be exhausted by the costly calculation of the Voronoi Diagram for our polygonal world, but it is enough to represent the free(W) in the form of v-edges (right and left obstacles) and v-nodes.**



*Figure 3.9 : Defining directed v-edge for the same directed boundaries (ccw polygons)*

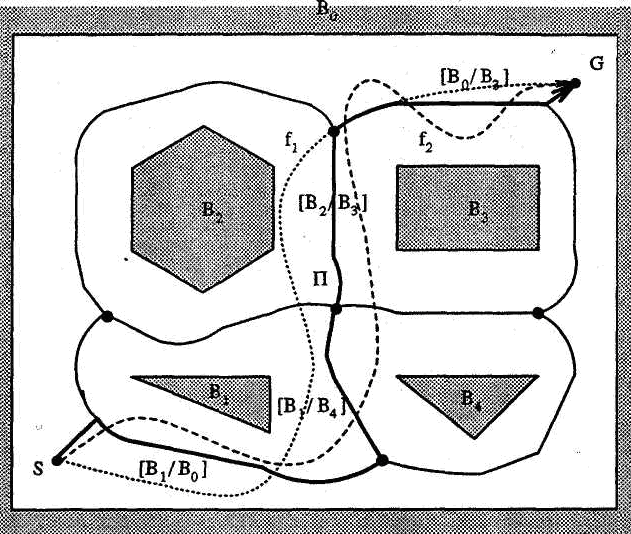
Although the assignment of left and right is arbitrary, it is fixed for all times once set. For consistency in this work, left and right polygons will be the first and second terms in directed v-edges, respectively. The following is the result of the previous discussion of directed v-edge.

**Lemma 4.1** *In a polygonal world W, where* W *is encircled by an outermost cw polygonal boundary and has n (n* ≥ *1) ccw polygonal obstacles inside the boundary, a directed v-edge always consists of two different polygons.*



*Figure 3.10: Defining directed v-edge for different directed boundaries (cw and ccw)*

### **3.3.2 Canonical Paths and Directed v-edges Sequences**

**

*Figure 3.11: Paths and canonical paths*

A robot can work only in the free space, free(W). A *path f* in a world W is a continuous function

*F* : [0,1] 🡪 free(W) (4.2)

Consider the problem of finding a path from a start configuration, *S,* to a goal configuration, *G* in a polygonal world W (Figure 4.11), where *ccw* polygons *B1,B2 , B3 and B4* are considered as obstacles for robot in this world and a world has one *cw* polygon *B0 .* It is desired to connect the start configuration, S, to the goal configuration, *G,* using a continuous, smooth path. There are infinitely many distinct paths connecting *S* and *G.* However, actually, we need to compare only paths which satisfy a special property.

Definition: A path Π is called a *canonical path* if there exists a sequence of directed

v-edges such that : Π = ss ξ1 . . . . ξk sg k ≥ 1 (4.3)

Where

1. the right hand side of Eq. 4.3 is the concatenation of *k* + 2 sub paths,
2. the sub path ss is the shortest path from *S* to ξ1,
3. ξ1 …. *ξk* is the sequence of directed v-edges, and
4. the sub path *sg* is the shortest path from *ξk* to *G.*

For example, in figure 4.12,  *Π = ss[B1/Bo][B1/B4][B2/B3][B0 /B3 )sg.*

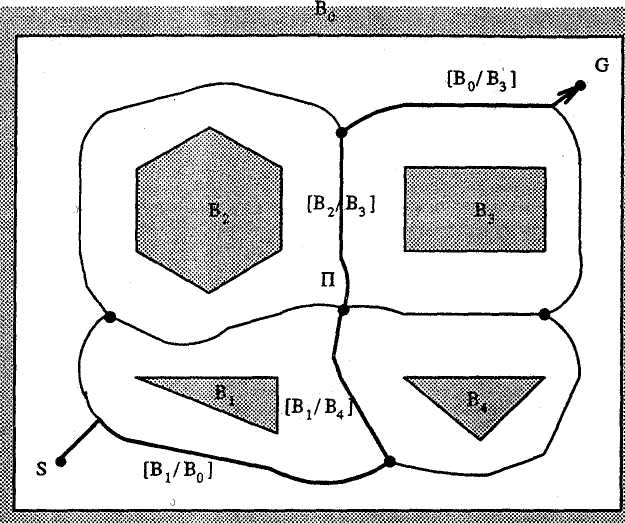
The following is the result of the previous discussion of the canonical path.

**Lemma 4.2** ; *For a given W, S, and G, a canonical path* ∏ *is the only one among all the paths in a homotopy class which satisfies the following conditions:*

* the sub path connecting *S* to first directed v-edge is the shortest one,
* sequential pieces from one directed v-edge to the next, and
* the sub path connecting the last directed v-edge to *G* is the shortest one.

**Proposition 4.1:** *For a given* W*, S, and G, for paths f*1 *and f2 in a homotopy class, if f1* 🡪 ∏1 and *f2* 🡪 ∏2  *then*  ∏1≡ ∏2 *,*

***Proof****.* Assume that the hypothesis is true. Since *f1* and ∏1 are homotopic, there is a continuous function Hwhich transforms *f1* into ∏1*.* Also, there is a continuous functionHwhich transforms *f2*  into ∏2 By Lemma 4.2, there is only one canonical path ∏ among all paths in a homotopy class. It follows that ∏1≡ ∏2 *,*



*Figure 3.12: Interpretation of canonical path as directed v-edges sequence*

**Definition**: A *directed v-edges sequence* γ is a finite sequence of directed v-edges such that no subsequence of *[Bi/Bj] [Bj/Bi]* is a part of it.

By definition, if ∏ is a canonical path, then ∏ = *ss* γ *sg* (Figure 2.12), where

γ is ξ1 .......ξk.

Several examples of directed v-edges sequences are illustrated in Figures 2.13 and 2.14. For example, the directed v-edges sequences for the above figures are as follows:

γ1 = *[B1 /B0*]*[B1 /B4][B2/B3][B0/B3*] *(Figure* 2.13)

γ2 = *[B1 /B0*]*[B4 /B0][B3 /B0] (Figure* 2.14)

**Proposition** **4.2**: *In a homotopy class, for all paths f1 and f2, γ1* = γ*2 is the necessary and sufficient condition to make f1* = *f2.*

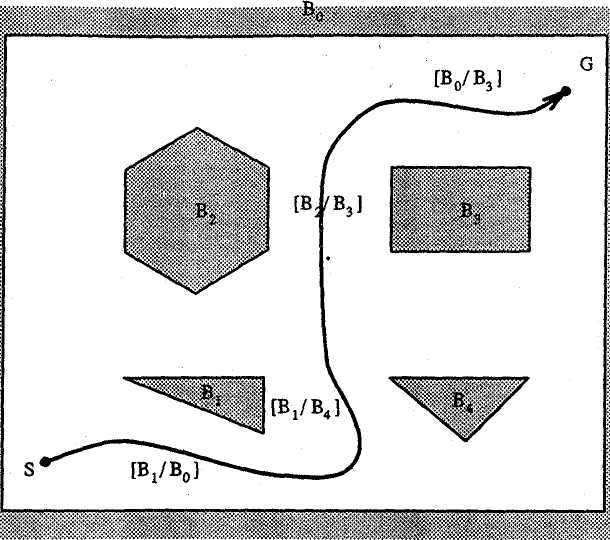
***Proof****.*

First prove the sufficiency. Assume γ1= γ2. If γ1= γ2, each path has a sequence of the same directed v-edges. Furthermore, in a homotopy class, both paths have the same left and right polygons. Each path is a concatenation of pieces. These pieces connect the start configuration to the first directed v-edge in γ. the sequential pieces from one directed v-edge to the next, and the last directed v-edge to the goal configuration. We can easily construct *H* to transform *f1* into *f2* piece by piece without running over any obstacles. The transformation, *H,* is the composition of the sequences of the transformations shown. Hence, the paths are homotopic.

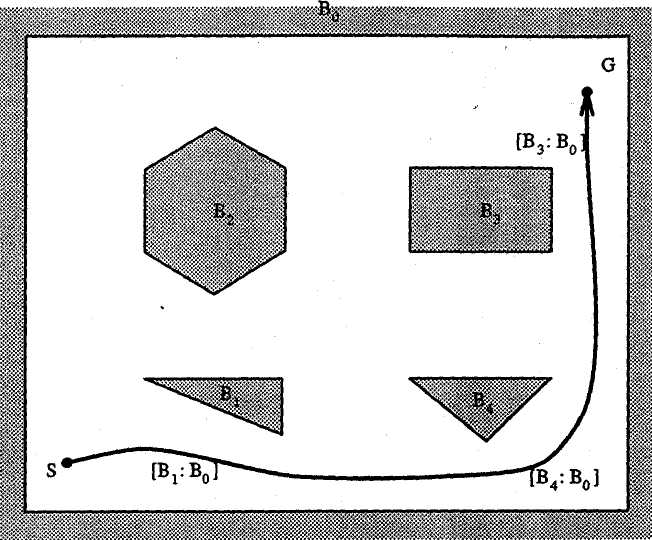
To prove the necessity, assume *f1 ≈ f2* We are given a path *f1.* Consider  
a directed v-edges sequence γ1 of *f*1. Since *f*1 and *f2* are homotopic, there is a  
continuous function *H* which transforms *f\* into *f2,.* Since *H(s, t)* is a continuous  
function, each directed v-edge ξ, which has left and right polygons, continuously  
concatenates with the next ξ over *s* as *t* moves when transforming *f1* into *f2.* However,  
there is no way in which *f*2 can eliminate, insert or repeat any ξ other than in the  
monotopic sequence of *f1. H(s, t)* can neither destroy existing nor create any new ξ,  
because *H(s,t)* € free(W) and *H(s,t)* is continuous. Therefore γ 1 = γ 2.

From above, we can conclude that:

* A directed v-edges sequence γ is unique for paths which are not homotopic.
* A directed v-edges sequence γ is a symbolic representation.



*Figure 3.13 : Directed v-edges sequence* [γ1]

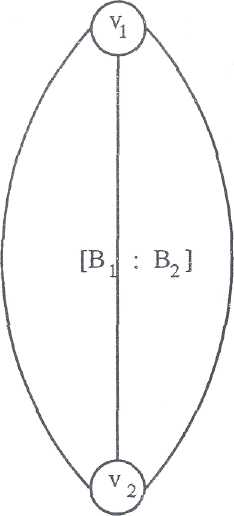


*Figure 3.14 : Directed v-edges sequence* [γ2]

### **3.3.3 Connectivity Graph**

We make the following observations about the world in Figure 4.7. Three Voronoi boundary segments intersect in one node (v-node). There is one line segment between two   
v-nodes (v-edge). Each v-node operates in both directions, and no v-node has a v-edge to itself.

***Definition****:* A *basic connectivity graph G* = (V, *E)* consists of V, a non-empty set of   
v-nodes, and E, a set of unordered pairs of distinct elements of *V* called undirected v-edges. Consequently this figure can be modeled using a basic connectivity graph, consisting of vertices which represent v-nodes, and undirected edges, which represent undirected v-edges, where each edge connects two distinct vertices.



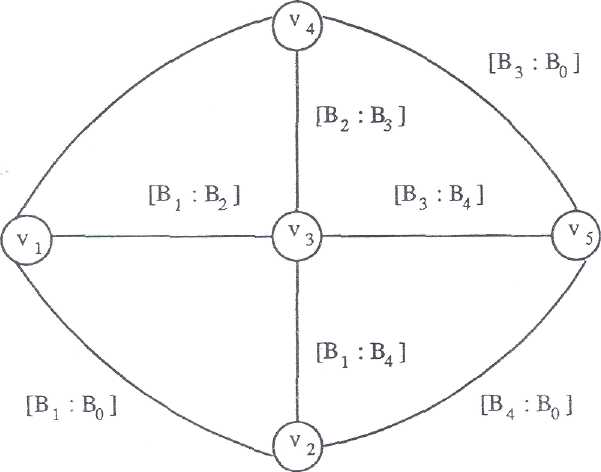
**[B2 : B0]**

*Figure 3.15 Basic connectivity graph of a polygonal world (1)*

[B1 : B0]

The basic connectivity graphs generated by the world in figures 2.7 and 2.8 are shown in figures 2.15 and 2.16.

Now we will explain how to represent a path class (see subsection 3.4).



****

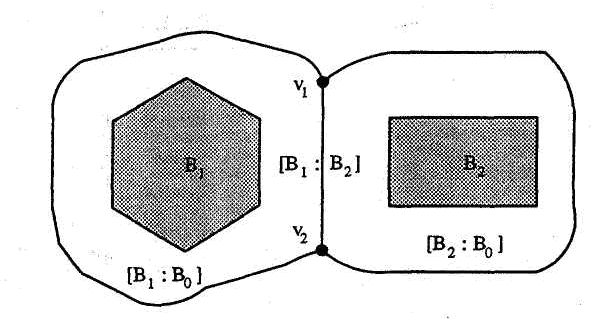
*Figure 3.16 : Basic connectivity graph of a polygonal world (2)*

### **3.3.4 Path Class Representation**

**B0**

**G**

**S**



*Figure 3.17 : Polygonal world (1)*

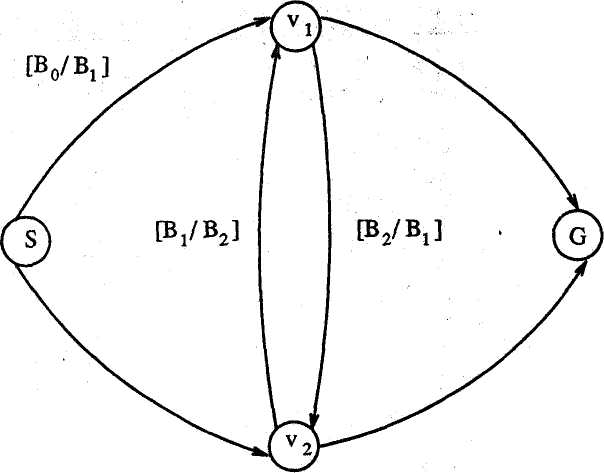
Consider the problem of finding a path from a start configuration *S,* to a goal configuration *G,* in a polygonal world W (Figure 3.17). It is desired to connect the start configuration S, to the goal configuration G, using a continuous, smooth path. In Figure 3.17, there are four different path classes. Consider the problem of how to symbolically represent each path class. A method based on directed v-edges is presented. Given start and goal configurations, we add two new nodes, S and G, to the basic connectivity graph to obtain an augmented connectivity graph. The augmented connectivity graph generated by the world in Figure 3.17 is shown in Figure 3.18. In which, there are four different path classes. In its most general form, a path classes, П, is symbolically represented by a sequence of directed v-edges. For instance, four different path classes in Figure 3.18 are represented by:

π 1  = [B0 / B1] [B0/B2]

π2 = [B0/B1] [B2/B1] [B2/B0]

π3 = [B1/B0] [B1/B2] [B0/B2]

π4 = [B1/B0] [B2/B0]



**[B1 / B0]**

**[B0 / B2]**

**[B2 / B0]**

*Figure 3.18 : Augmented connectivity graph of a polygonal world*

Another example is shown in Figure 3.16. The augmented connectivity graph generated by the world in Figure 3.19 is shown in Figure 3.20. In which, there are twelve different path classes which connect *S* with *G:*

*π1 = [B0/B1] [B0/B2] [B0/B3]*

*π2 = [B0/B1] [B0/B2] [B3/B2] [B3/B4] [B3/B0]*

*π3 = [B0/B1] [B0/B2] [B3/B2] [B4/B1] [B4/B0] [B3/B0]*

*π4 = [B0/B1] [B2/B1] [B2/B3] [B0/B3]*

*π5 = [B0/B1] [B2/B1] [B3/B4] [B3/B0]*

*π6 = [B0/B1] [B2/B1] [B4/B1] [B4/B0] [B3/B0]*

*π7 = [B1/B0] [B1/B4] [B1/B2] [B0/B2] [B0/B3]*

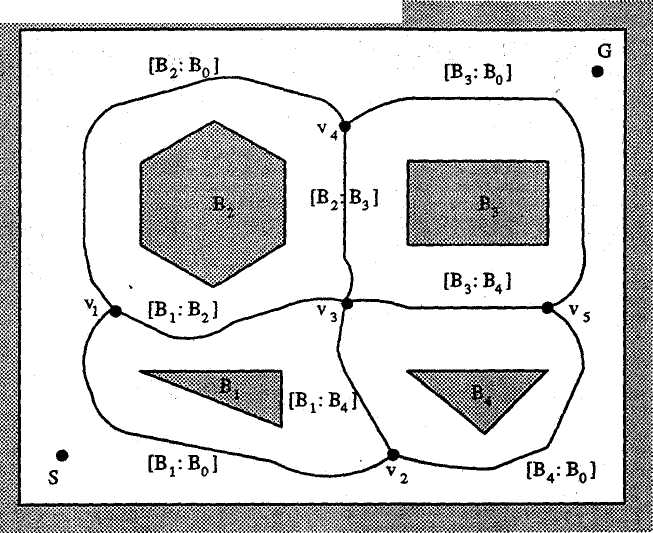
*π8 = [B1/B0] [B1/B4] [B2/B3] [B0/B3]*

*π9 = [B1/B0] [B1/B4] [B3/B4] [B3/B0]*

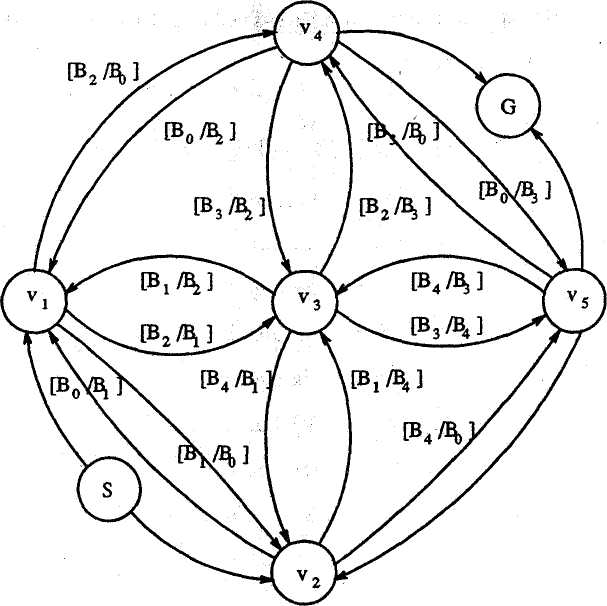
*π10 = [B1/B0] [B4/B0] [B3/B0]*

*π11 = [B1/B0] [B4/B0] [B4/B3] [B2/B3] [B0/B3]*

*π12 = [B1/B0] [B4/B0] [B4/B3] [B1/B2] [B0/B2] [B0/B3]*



*Figure 3.19: Polygonal world (II)*



*Figure 3.20 : Augmented connectivity graph of a polygonal world (*

## 3.4 Finding the Best Path Class

In this section we outline how to find the best path class. Finding the best path class from start configuration S, to goal configuration *G,* in the world is transformed into the minimum cost path finding problem from S to *G* in the augmented connectivity graph. The augmented connectivity graph uses a weighted edge whose value depends upon the mission-based cost function associated with the v-edge. For instance, a cost for the edge is defined as the energy (or time) for the vehicle to make a motion from one v-node *Vi* to another v-node v*j.* This cost not only reflects the distance, but the turns needed to make the motion. It may also reflect the safety (i.e., if the region is narrow, the cost is high). Then by applying any of the graph searching techniques (discussed in appendix B), e.g. Dijkstra's algorithm can be perfectly applied to this global motion planning problem. The best path class in terms of a sequence of directed v-edges is hence obtained. The computation time is

*O((n* + *m)log n)* using a priority queue, where *n (* is v-nodes (vertices) in a graph *)* and *m(is the number of* v-edge *in the graph)* in the augmented connected graph respectively.

Once the path class is found, it is passed to the next stage of the Motion planning process, Local Motion Planning which ensures that the vehicle will follow the path class to reach the goal. The choice of the mission type ultimately affects which steering function (which is out of our scope in this work) is used to guide the vehicle. Simulation results for the proposed approach are given in chapter 6.

## 3.5 Image of Point on Convex Polygon

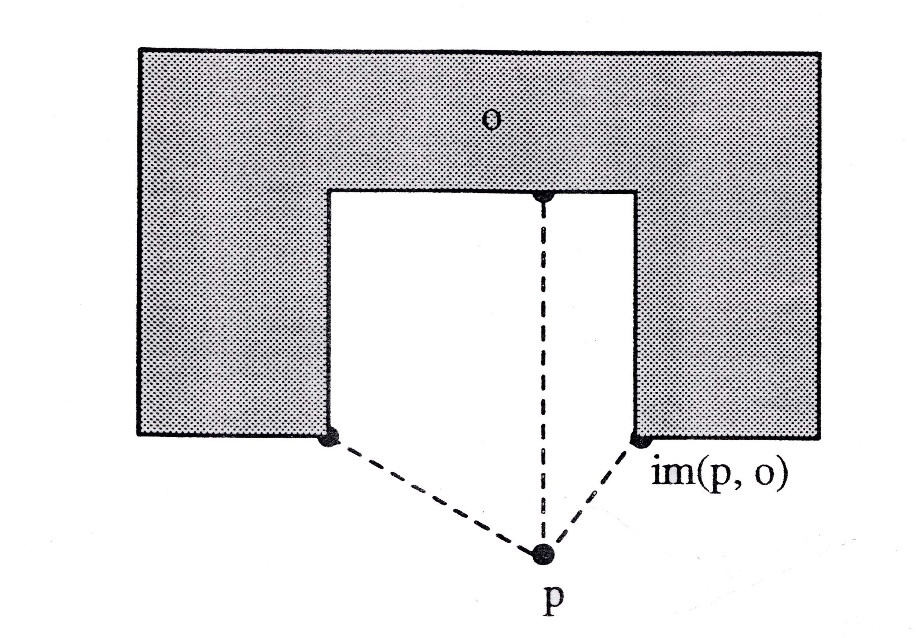
We assume a global two-dimensional Cartesian coordinate system in a plane *E2.* Given two distinct points *P1* = *(*x1 *,* y1 *)* and *P2* = (x2 *,* y2 )in *E2,* The Euclidean distance *d(p1,p2)* between them is defined as:

(2.21)

Assume that there is an object *O* in a plane. An object might be a point, a line, an open line segment, a polygon, or other set of points. The shortest distance *d(p, o)* between a point *p* and an object *O* is defined as follows:

(2.22)

Eq. 2.22 generalizes the function *d* defined by Eq. 2.21



*Figure* *3.23 Image on object*

***Definition****:* A point *p1 in* *O* which satisfies *d(p,p1)* = *d(p, o)* is said to be an *image* of *p* on *o* and is denoted by *im(p, o)* (Figure 3.23).

If a world W has more than one object, an image *im(p,* W) is defined as the image *im(p,* oi) such that *d(p,* oi) is the minimum over all objects in W (figure *3.24*).

**O2**

**O1**

im(p,o2)

**O3**

im(p,o1)

P

im(p,o3)

*Figure 3.25: Images on world*

### **3.5.1 Visibility from Point to Polygon**

Assume that we are given a convex polygon *B* = *(v1 , . . . . , vn)* and a point   
*p*  free(B). The significant notion for our purpose is the following classification of each vertex vi of *B* with respect to the segment*.* Each vertex of *B* is said to be *visible, invisible,  
 cw-tangential,* or *ccw-tangential*  (Figure 2.25).

V9

V8

*Cw-tangental*

**P**

V10

V7

**Visible**

**B**

V6

V1

V2

**invisible**

V3

V4

V5

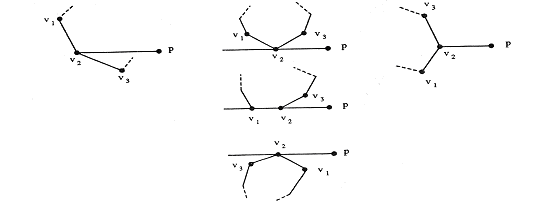
*Ccw-tangental*

*Figure 3.25 : Visibility from point p to convex polygon B.*

***Definition****:* Let *B* be a convex polygon, and let a point *p*  free(B).

* A vertex vi is *tangential* from point p if the two vertices adjacent to vi lie on the same side of the line containing*.*
* A vertex vi is *visible* if the segment does not intersect the interior of *B* and the two vertices adjacent to vi lie on opposite sides of the line containing 
* A vertex vi is invisible if the segment intersects the interior of B

Let *cw-tangential(p,vi,B)* denote that vertex *vi* of *B* is clockwise tangential with respect to the segment , *ccw-tangential(p,vi,B)* denote that vertex *vi* of *B* is counterclockwise tangential with respect to the segment *, visible(p,vi,B)* denote that vertex *vi* of *B* is visible with respect to the segment *,* and *invisible(p,vi,B)* denote that vertex *vi* of *B* is invisible with respect to the segment *.* It is now easy to establish the following lemma.



(b) cw and ccw tangential

(c) visible

(a) invisible

*Figure 3.26: Classifications of vertex vi of polygon B with respect to a segment *

**Lemma 5.2** *Given a convex polygon B and a point p*  *free(B), the vertex vi is one of the following four types:*

*cw –tangential (p,vi,B)* ≡ ~*ccw* (p,*vi*,Φˉ¹(*vi*)) *& ~ccw(p,vi,Φ(vi))* (5.3)

*ccw –tangential(p,vi,B)* ≡ ~*ccw* (p,*vi*,Φˉ¹(*vi*)) *& ~cw(p,vi,Φ(vi))* (5.4)

*visible(p,vi,B)* ≡ *ccw* (p,*vi*,Φˉ¹(*vi*)) *& cw(p,vi,Φ(vi))* (5.5)

*invisible(p,vi,B)* ≡ *cw* (p,*vi*,Φˉ¹(*vi*)) *& ccw(p,vi,Φ(vi))* (5.6)

*Proof .*

For the first part (Eq.5.3), *vi* is *cw-*tangential if the two vertices adjacent to *vi,* andlie on the same side of the line containing, and we have the following three cases.

* **case 1:** *cw(p,vi,Φˉ¹(vi)) & cw(p,vi,Φ(vi))*

If *pvi* and *vi Φˉ¹(vi)* make a right turn at *vi, PΦˉ¹(vi)* is clockwise from *pvi*, And *pvi* and *vi Φ(vi)* make a right turn at *vi, PΦ(vi)* is clockwise from *pvi then vi is cw-tangential.*

* **case 2:** *col(p,vi, Φˉ¹(vi)) & cw(p,vi,Φ(vi))*

If *p,vi* and *Φˉ¹*(*vi*) are collinear and *pvi*  and *viΦ(vi)* make a right turn at vi, *PΦ(vi)* is clockwise from *pvi*, then *vi* is *cw-tangential*.

* **case 3:** *(p,vi,Φˉ¹(vi)) & cw(p,vi,Φ(vi))*

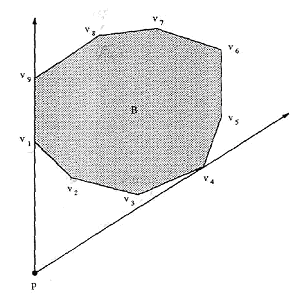
If *pvi* and *viΦˉ¹(vi)* make a right turn at *v i, PΦˉ¹(vi)* is clockwise from *pvi*, And *pvi* and *Φ(vi)*are collinear**,** then *vi* is *cw-tangential*.

This gives a proof of (Eq.4.3), in other word *vi*, is *cw-tangential* from *p* (figures 3.25, 3.26)

The second part (Eq.5.4) is proven similarly. (Eq.4.3 = Π = ss ξ1 . . . . ξk sg k ≥ 1)

For the third part (Eq.5.5), since the t+wo vertices adjacent to *vi* lie on the opposite side of the line containing *pvi* , and *pvi* doesn’t intersect the interior of *B*, therefore *pvi and viΦˉ¹(vi)* make a left turn at *vi, pΦˉ¹(vi)* is counter clockwise from *pvi,* and *pvi and viΦ(vi)* make a right turn at *vi, pΦ(vi)* is a clockwise from *pvi,* this gives a proof of equation (Eq.5.5)

For the last part (Eq.5.6), since *pvi* intersects the interior of *B,* therefore *pvi* and *pvi and viΦˉ¹(vi)* make a right turn at *vi, pΦˉ¹(vi)* is clockwise from *pvi* this gives a proof of (Eq.5.6)



*Figure 3.27 Visibility from point p to convex polygon B*

For example, in Figure *3.27*, vertex *v1* is *cw-tangential,* vertex  is *visible,* vertex  is *ccw-tangential* and vertex is *invisible*

### 3.5.2 Type of an Image from a Point to a Convex Poly­gon

Let *B* denote a convex polygon with n vertices. Let a point *p*  free (B). The image of *p* may be on an edge or a vertex of convex polygon. If an image of *p* is on an edge, the image moves when p moves slightly. However, if the image of *p* is on a vertex, it does not move when *p* moves slightly. The following theorem determines the image occurs either on an edge or on a vertex.

**Theorem 5.1:** let B = {v1,…,vn} be a convex polygon, and let p be a point in free(B) and define θ, θ1 and θ2 by:

*θ = ψ(vi, Φ(vi))+ П/2*

*θ1= ψ(p, vi)*

*θ2= ψ(vi, Φ(vi))*

Let vertex *vi* be *cw-tangential* from point p. there exists a vertex *vj* *( i = j or i ≠ j )* such that the image of p on B is of one of the following two types.

*(І) If* (*θ1> θ*) ^ (*θ2< θ*) (5.7)

*then the image lies on an edge viΦ(vi) of polygon B.*

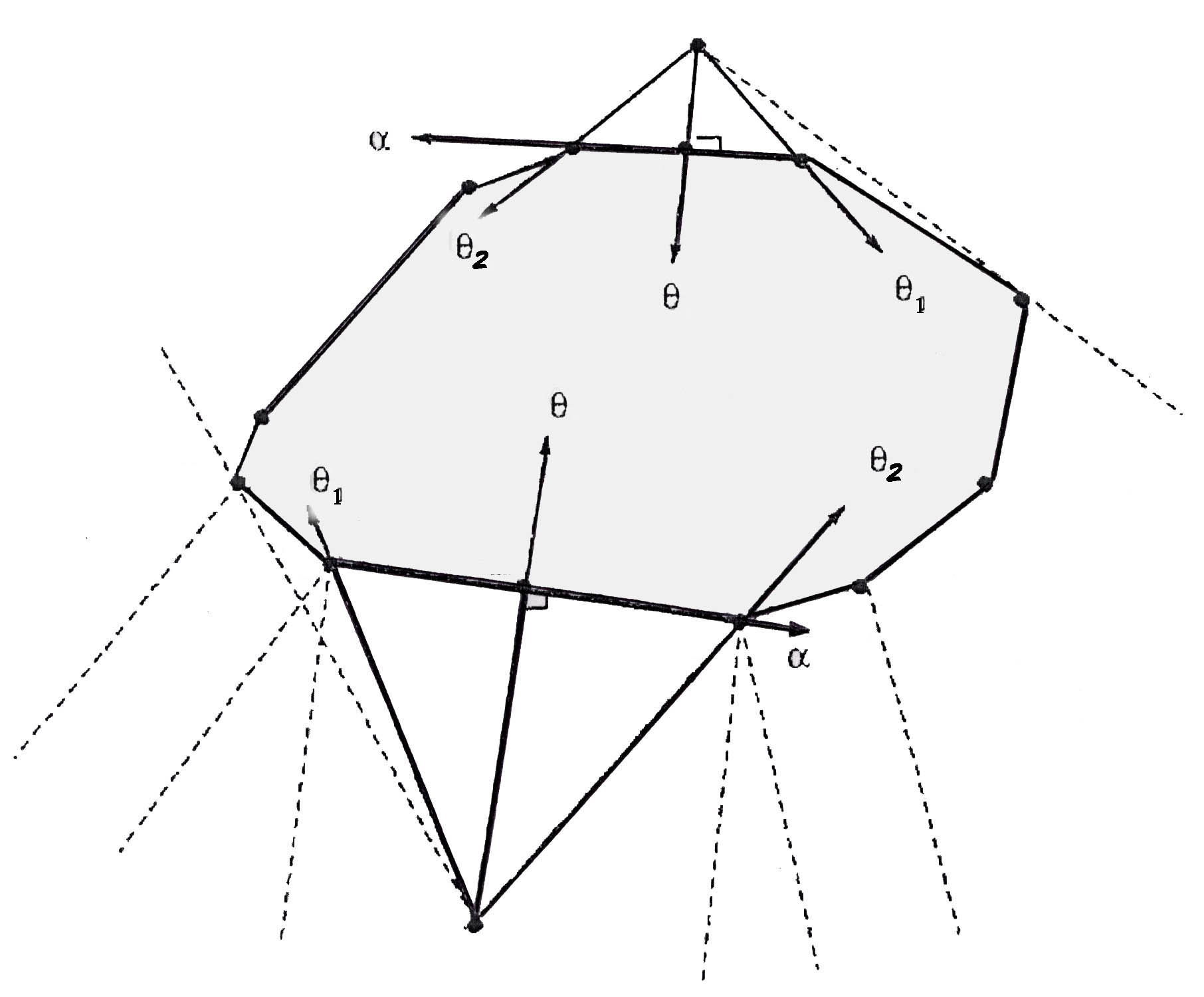
*(ІІ) If*  (*θ1 ≤ θ*) ^ (*θ2 ≤ θ*) (5.8)

*then the image of p is a vertex v of polygon B.*

*Proof.*

Consider two straight lines, one joining *p* with *vi* and the other joining *p* and . The orientations of these two lines are *θ*1 and *θ*2 respectively. Also, denote by *α is* the orientation of  from *vi* to and is the perpendicular from *p* to.

For the first part of the proof (Eq. 5.7), let *pim* be the intersection of two lines whose orientations are α and *θ* (Figure *2.28*). Assume that the hypothesis of   
Eq.5.7 is true. Since *θ1* > *θ,* then and make a *left turn* (see appendix A) at *pim .* Also, *θ2 < θ,* then and  make a right turn at *pim*. It follows that *pim* is visible from *p* by lemma 5.2. This means that *vi* and  are on opposite sides of *pim .* Therefore, *pim* lies on the boundary of *B.* In other words, *pim* lies on an edge of B.



Cw-tangential

Cw-tangential

**P**

**P**

**Pim**

**Pim**

**B**

**V1**

**V2**

**V3**

**V4**

*Figure 3.28: Images of point p lies on an edge of convex polygon B*

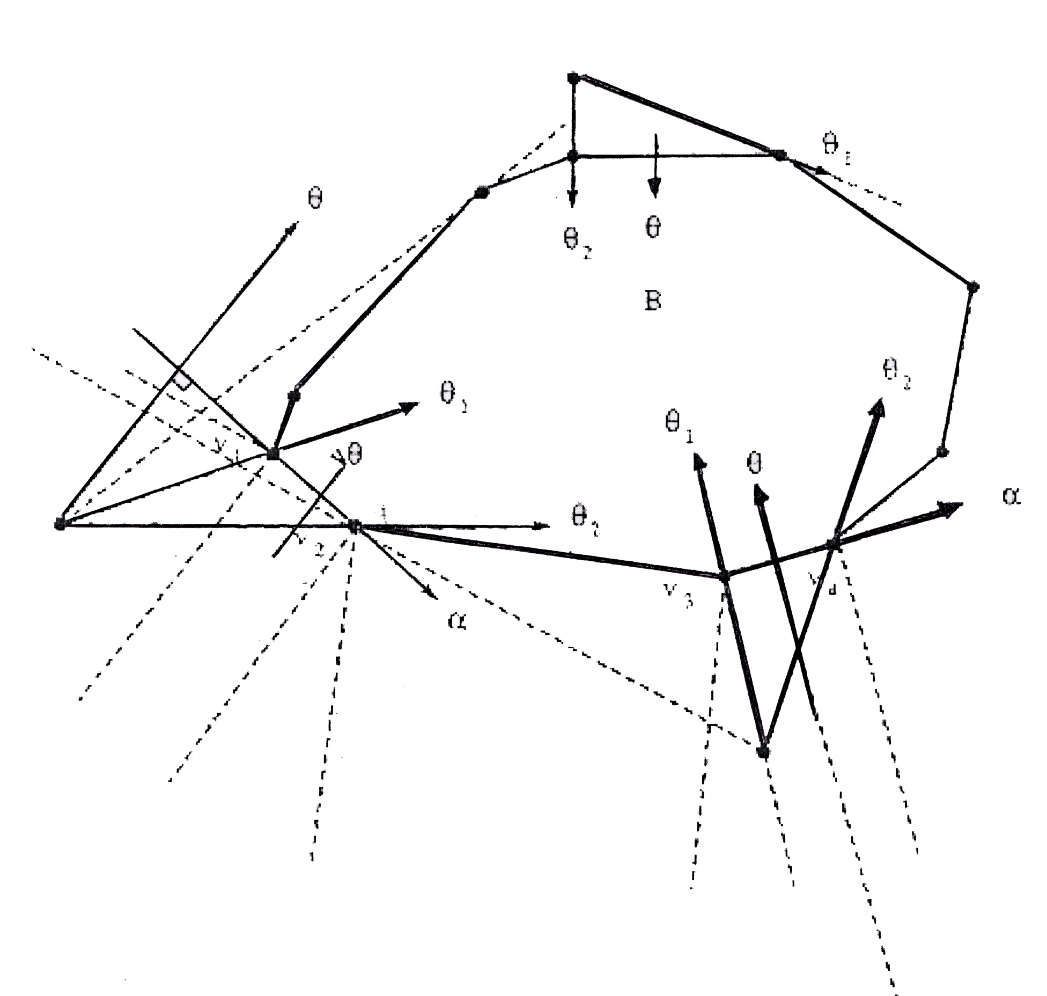
For the second part (Eq. 5.8), assume that the hypothesis of Eq. 5.8 is true; we have the following three cases (see Figure 5.12).

• Case *1: θ1<θ* AND *θ2<θ*

Since *θ >* *,* and *θ >*  *-* Therefore the image of *p* does not lie on the edge .But *<θ*, since is the next vertex to *vi.* Then *vi* is a closed point from *p.* Therefore, the image of *p* is a vertex vi.

• Case 2: *=θ* AND *<θ*

Since *=θ*and *<θ*, then the image of *p* does not lie on the edge .But *<*since is the next vertex to *vi.* Then vi, is a closed point from *p.* Therefore, the image of *p* is a vertex vi.



*Figure 3.29: Image of point p lies on vertex vi of convex polygon B*

**P**

**P**

cw-tangential

cw-tangential

cw-tangential

**B**

**P**

• Case 3: *<θ* AND *=θ*

Since *<θ*and *=θ,* then the image of *p* does not lie on the edge But *<θ2* , since is the next vertex to *v*i. Then  is a closed point from *p.* Therefore, the image of *p* is a vertex .

This gives a proof of Eq. 5.8. in other words, *pim* occurs on a vertex of *B.*

Since there are no vertices in the interior of a convex polygon B, then by Theorem 5.1 we obtain the following corollary.

**Corollary 5.1:** *For any point P  free(B) and a convex polygon B, there exists only one image from p to a convex polygon B.*

### **3.5.3 The Image Type Algorithm**

Now, we describe the construction of an algorithm for image type. The block diagram for finding image type is shown in Figure *2.29*. The inputs are a convex polygon *B* and a point *p * free(B). The outputs are an image type (vertex type or edge type), a vertex *v*i, the orientation from *p* to its image, and the closed distance from *p* to the image. For a vertex type image, vertex *vi* is the image of *p* on *B,* but for an edge type image, the image of *p* on *B* lies on an edge  , In pseudo-code the method is as follows:

Convex Polygon B

Point p

Closed Distance

Image type (edge or vertex)

Find Convex Image

Algorithm

Vertex v

Orientation from p to image

*Figure 3.29 : Image type block diagram*

**Convex Image( p, B )**

Input: *point* p (** free(B))

Convex polygon B = (v1,…,vn)

Output: *image* image type (vertex-type or edge-type)

vertex v

orient (the orientation from p to image)

closed (the distance from p to image)

**BEGIN**

v = first vertex(B)

\*\*\*Find clockwise tangential\*\*\*

While *(ccw(p,v,Φˉ¹(v)) or ccw(p,v,Φ(v))*

*v= Φ(v*)

Loop

\*\*\*Find image type\*\*\*

Do Until (*image reached*)

*θ = 00)+ П/2*

*θ1= ψ( p, v)*

*θ2= ψ( P, Φ(v))*

if ((*θ1≤ θ) ^ ( θ2≤ θ))* then

image.type = VERTEX

image.posi = v

image.orient = *θ1*

image.closed = Computer\_Education\_Distance(*p,v*)

else if ((*θ1> θ) ^ ( θ2< θ))* then

image.type = EDGE

image.posi = v

image.orient = *θ*

image.closed = Compute\_Dist(p,v)

else

*v= Φ(v)*

*end if*

Loop

**END**

The algorithm simply loops until the *image* is reached. First, the algorithm loops until a cw-tangential vertex is reached.

Through the second loop, we check the condition for vertex image type. If the condition is not satisfied, the condition for edge image type is checked. Also, if it is not satisfied, we take the next vertex in ccw direction. We continue in this process until one of the above two conditions satisfies.

The subroutine "Compute\_Euclidean\_Distance" computes the distance between two points (). The subroutine "Compute\_ Dist" computes the closest distance from point *p to* its image which lies on an edge. The subroutine for Compute\_Dist is as shown below:

**Compute\_Dist ( *p, v* )**

Input: point *p ( free(B)) V* first vertex of edge where the image on it

output: closed the closet distance from *p* to *image*

**begin**

*area =* Compute\_Area\_Triangle(*p,v,Φ(v)*)

*dist =* Compute\_Euclidean\_Distance(*v,Φ(v)*)

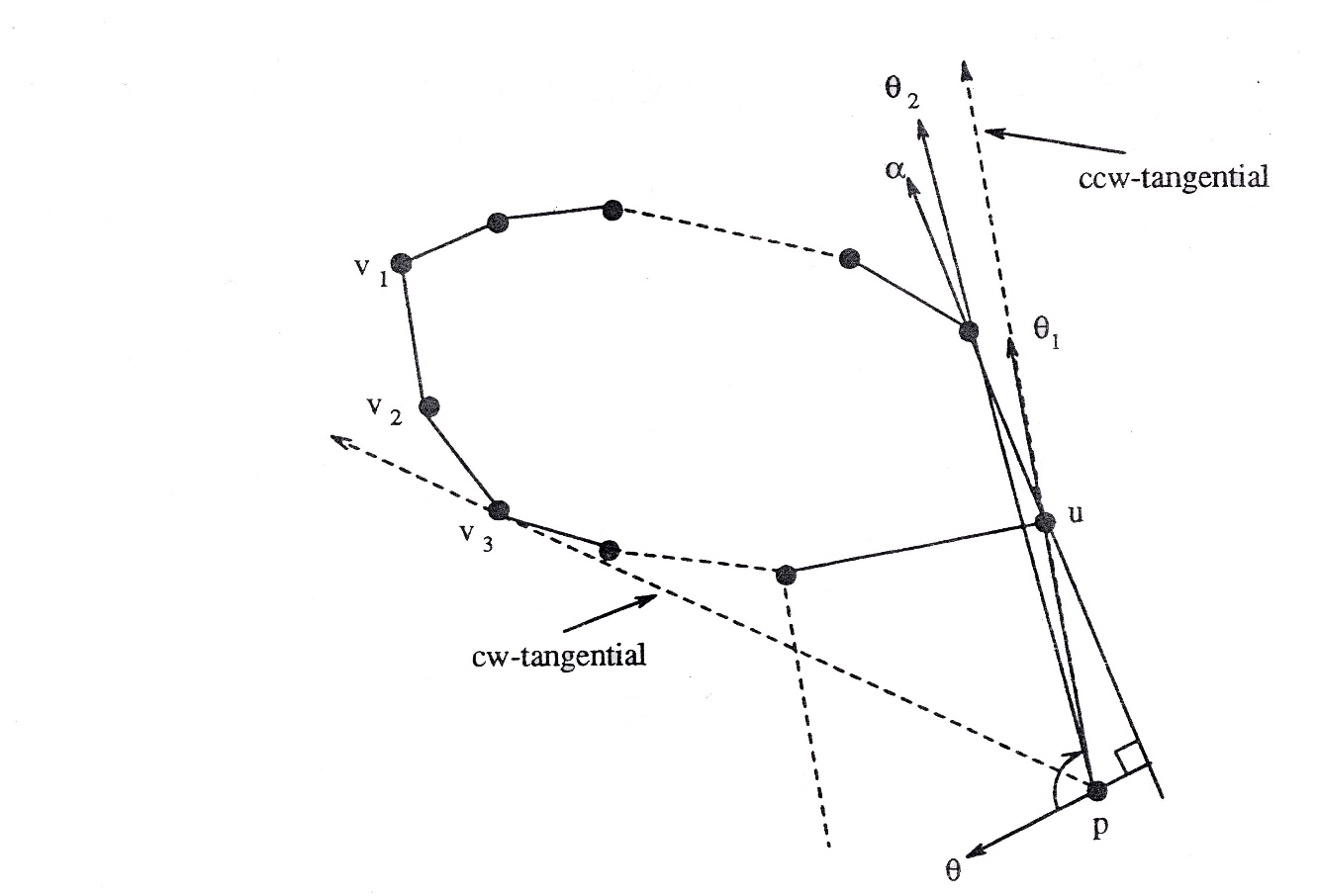
*closed=2×area/dist*

return *closed*

**end**

The subroutine Compute\_ Area\_ Triangle computes the area of triangle (see Appendix A).

#### 3.5.3.1 Correctness of the Algorithm



*Figure 3.30: Correctness of image type algorithm*

To prove the correctness of the above algorithm, we want to show that the algorithm always returns an *image* structure when the Do-loop is executed. In other words, the Do-loop is never executed forever.

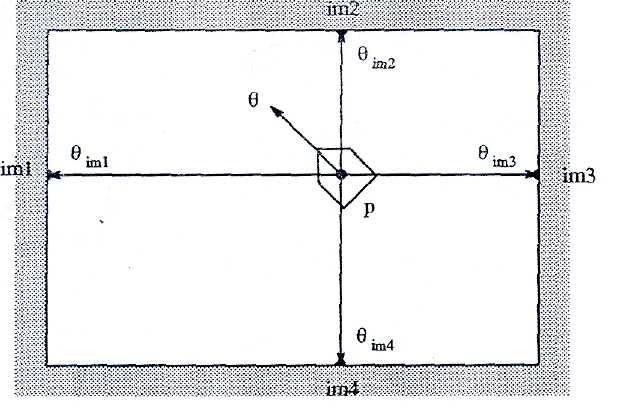
Assume that *v1* is the starting vertex of polygon *B* (Figure 2.30). Since v3 is  *cw-tangential,* the while-loop returns *v* = v3. It follows that, at the beginning of the Do-loop,   
*v* will be checked to determine the image type.

If the two if conditions are not satisfied, we take the next vertex, v = Φ(v). In the worst-case, we continue in this process until vertex v = vn . Vertex v is ccw-tangential, but the first if condition will be satisfied (θ1 < θ ^ θ2 < θ). It follows that the algorithm returns the image type of point p as vertex type and vertex v. This proves that the Do-loop is always terminated.

#### 3.5.3.2 Analysis of the Worst-Case Time

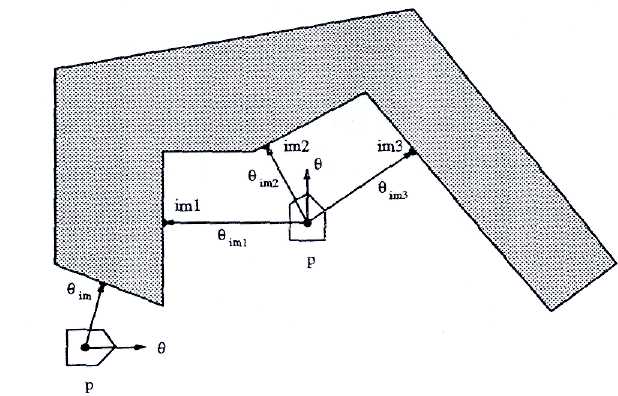
The operations each takes O(1) time, except. The while loop will be taken *O(n)* time in the worst-case. The Do-until-loop takes *0(n)* time in the worst-case. The overall running time of the algorithm is *0(n).*

### **3.5.4 Finding an Image on a Non-convex Polygon**



*Figure 3.31: Image of a point p on cw concave polygon B*

Suppose we have an outermost non-convex *cw* polygonal boundary (Figure 3.31) or non-convex *ccw-*polygon obstacle inside the boundary (Figure 3.32). Let a point *p *free(B). In the case of an outermost non-convex *cw*-polygon, there is more than one image. The image always lies on an edge of *B.* In the case of non-convex *ccw* polygon, there may be one or more images depending upon the position of the robot. The image may be one of the vertices of *B* or it may lie on an edge of *B.* We have the following observations. First, the image may be behind the vehicle. For instance, in Figure 3.31, *pim3* and *pim4* are behind the vehicle. In this case, this can not be an image.



*Figure 3.32: Image of a point p on ccw concave polygon B*

The following remark illustrates how we can know whether the image is behind a vehicle. Let *θ* denote a vehicle's heading (the direction from *p)* and let ψ*(p,pim)* denote the direction from *p* to *pim*

**Remark 5.1:** *given a non convex polygon B and a point* *p * *free (B)*

*(a) if*

| Ф (θ – ψ ( p, pim )) | ≤ П/2 (5.9)

*then the image of p is usable.*

*(b) if*

| Ф (θ – ψ ( p, pim )) | > П/2 (5.10)

*then the image of p is behind the vehicle.*

The second observation, for the usable image (Eq. 5.9) the following remark illustrates how we can know if the image is in the right, left or front of the vehicle.

**Remark 5.2:** *the real image is of one of the following three types.*

*(a) if*

Ф (θ – ψ (p, pim )) > 0 (5.11)

*Then the image of p is on the right of the vehicle.*

*(b) if*

Ф (θ – ψ (p, pim )) < 0 (5.12)

*Then the image of p is on the left of the vehicle.*

*(c) if*

Ф (θ – ψ (p, pim )) = 0 (5.13)

*Then the image of p is on the front of the vehicle.*

To summarize: in the case of a non-convex polygon, we conclude that

* We need another algorithm to find the image(s).
* We need another data structure for the image. In this case, we may have one or more images. Therefore, we need an array of image structures. The size of this array is the maximum numbers of images.
* If the initial orientation, *θ,* of the vehicle is in the opposite direction to the desired motion of the vehicle, then we cannot reject the, image which lies behind the robot.

According to above, the division of the boundary cw-polygon into set of sub polygons will let us use the same algorithm for convex polygons and the same data structure for the image.

## 3.6 Path Classes and Sub-Polygons

The objective of path classes using polygonal world is to provide useful in­formation for local motion planning. The directed v-edges sequences, of a world W which consists of a finite number of polygons n is independent of the position of the vehicle inside the free(W). For example, in Figure 3.17, suppose the path class π = [B1 / Bo] [B2 / B0] and the start configuration of the vehicle are given. Also, we know that any point within free(W) has its left and right images on the two boundary polygons. We proved previously that there is only one image of a point (that lies in free space) to a convex polygon and more than one image for a non convex polygon. When representing the path class in a polygonal world, we have the following disadvantages:

* In Figure 3.33, B1 and B2 are *ccw* convex polygons and a *B0* is *cw* non convex  
  polygon. When the vehicle starts to navigate through the given path π, left image is *im3* and its right images are *im1* and *im2.* Since the start orientation of the vehicle is  
  *θ*, the right images are *im1* and *im2 ,* but *im2* is behind the vehicle.
* We can not construct the connectivity graph if a world W consists of two poly­gons B0 and B1, where W is encircled by an outermost *cw* polygonal boundary B0, and has one *ccw* polygonal obstacle B1 inside the boundary (Figure 3.34), since every v-node of the connectivity graph is the common intersection of three or more Voronoi boundary segments.

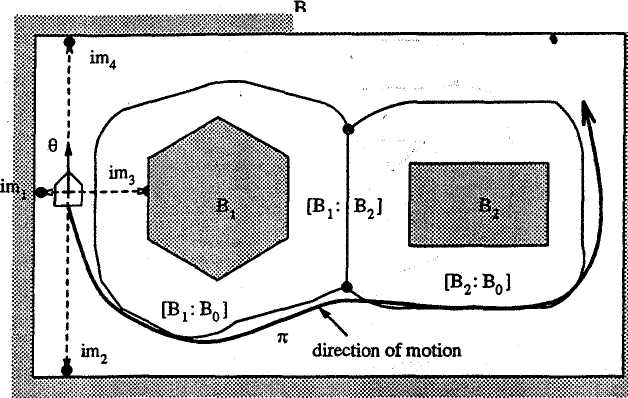


Figure 3.33: Problem1: initial orientation of a vehicle is different from the direction of a motion

Due to the above shortcomings, we need more information when we represent the path classes in order to simplify local motion planning. The use of the sub-polygons (refer to sec. 5.1) will solve the above problems and give more information for the local motion planning task.

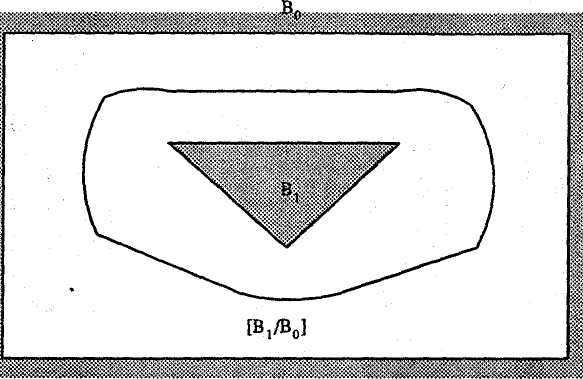


Figure 3.34 : Problem 2: Voronoi diagram of polygonal world consisting of two polygons (ccw *polygon inside cw polygon boundary)*

Consider the same world W in Figure 3.33 The non convex polygon *B*can be broken into four sub-polygons *B**, B*,B, and *B03* (Figure 3.35). The basic connectivity graph generated that world is shown in Figure 3.36. There are six   
v-nodes *(v*1*, ...*, *v**)* and seven undirected v-edges:

[B1 : B00 ], [B1 : B01 ], [B1 : B03 ], [B2 : B01 ], [B2 : B02 ], [B2 : B03 ], [B1 : B2 ].

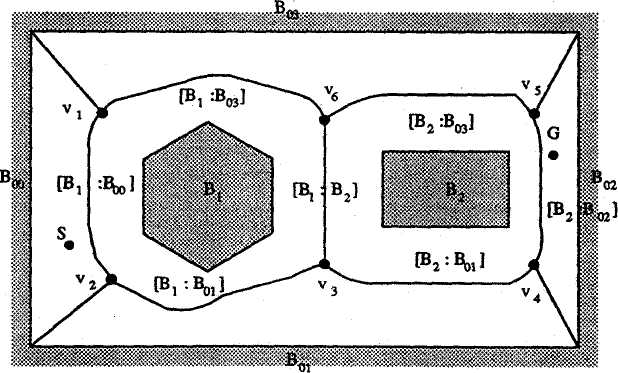


Figure 3.35: Solution of problem 1: Voronoi diagram of a sub polygonal world

Now, assume that a start configuration, *S,* and a goal configuration, *G,* are given in free(W) (Figure 3.35). The augmented connectivity graph generated by this world is shown in Figure 3.37. While there are four different path classes represented by a directed v-edges sequences as follows:

**[B1:B01]**

**V4**

**V3**

**V2**

**[B2:B01]**

**[B1:B00]**

**[B1:B2]**

**[B2:B02]**

**V5**

**V6**

**V1**

**[B2:B03]**

**[B1:B03]**

*Figure 3.36: Basic connectivity graph of a sub polygonal world*

π1 = *[B00/B1] [B03/B1] [B03 /B2] [B02/B2]*

π2 = *[B00 /B1] [B03 /B1] [B2 /B1] [B2 /B01] [B2 /B02]*

π3 = *[B1 /B00] [B1 /B01] [B1 /B2] [B03 /B2] [B02 /B2]*

π4 = *[B1 /B00] [B1 /B01] [B2 /B01] [B2 /B02]*

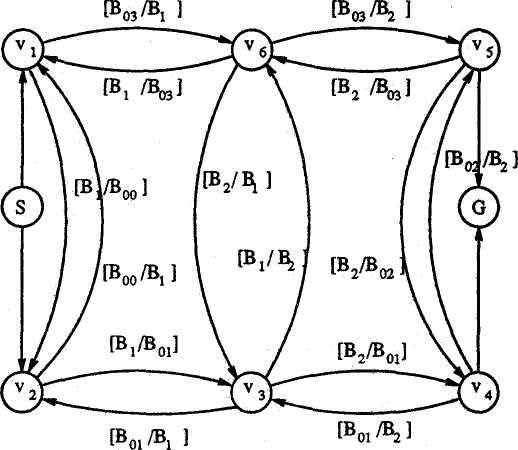
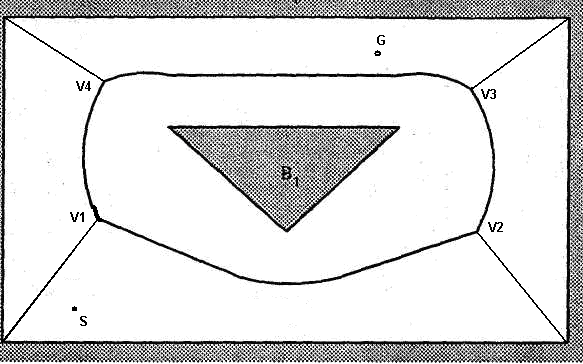


Figure 3.37: Augmented connectivity graph of a sub polygonal world

As a result, the use of sub-polygons solves the problem when the start orien­tation of the vehicle is different from the direction of the motion. In other words, path classes represented by sub-polygons possess more information for local motion planning than do those represented by polygons.

**B02**

****

**B03**

**B04**

**B01**

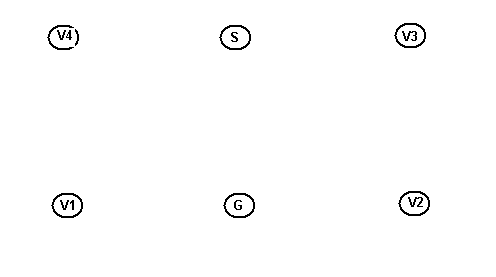
**[ B1 :B01 ]**

**[ B1 :B03 ]**

**[ B1 :B04 ]**

**[ B1 :B02 ]**

**[ B1 :B02 ]**



**[ B04 :B1 ]**

**[ B03 :B1 ]**

**[ B01 :B1 ]**

**[ B1 :B01 ]**

**[ B1 :B03 ]**

**[ B02 :B1 ]**

**[ B1 :B04 ]**

Figure 3.38: Solution of problem 2: world and augmented connectivity graph

## 3.7 Advantages of Path Class Representation Using Directed V-Edges Sequences

There are several advantages. They include:

1. A unique representation of a path class. In other words, this representation is unambiguous since a directed v-edge is defined by the "closest" two obstacle features (like vertices

sequence, obstacle shape and any other properties of the obstacle which will be used by the robot sensors through its navigation).

* It is exact free space decomposition, so that if a path exists, the local  
  motion planning should be able to find it.
* It simplifies the planning of collision-free paths for a robot among obstacles  
  once the directed v-edge sequence, in which the robot is located, is identified.
* The local motion planning problem is expected to be simpler if a path class representing by directed v-edge sequence is given.
* This PCMP algorithm of solving the motion planning problems is exactly the same as the human mind, solve any transition mission in any known environment.

CHAPTER.4

# SHORTEST PATH PLANNING ALGORITHM

## 4.1 CANONICAL PATHS AND TANGENT SEUENCES

This text introduces the concept of canonical paths and tangent sequences in the context of path planning within a world of convex polygons. Here's a breakdown:

**Canonical Paths:**

* Informally referred to as the **shortest paths** (locally) within a specific class of paths.
* The goal is to identify these "canonical paths" symbolically using a concept called a **tangent sequence**.

**World Definition (W):**

* Represented by the symbol W, it's a set of n convex polygons (n > 0).
  + Convex polygons have all interior angles less than 180 degrees and no inward corners.

**Oriented Polygons (Bi+, Bi-):**

* Each polygon Bi in the world W can be represented in two orientations:
  + Bi+ (counter-clockwise): This signifies following the polygon's boundary in a counter-clockwise direction.
  + Bi- (clockwise): This signifies following the polygon's boundary in a clockwise direction.
* This concept of orientation is crucial for path planning algorithms.

**Set of All Oriented Polygons (Z):**

* Represented by the symbol Z, it's the collection of all possible oriented polygons derived from the original set W.
* Formula: Z = {B1+, B1-, ..., Bn+, Bn-} (contains 2n elements, one for each polygon in both orientations).

**Tangent Sequence (σ):**

* Defined as a sequence of oriented polygons.
* A key restriction: The sequence cannot contain subsequences like B+B- or B-B+ where B belongs to the original set W (prevents going around a single polygon in both directions).
* Empty sequence: denoted by ε.

**Summary:**

This explanation sets the stage for understanding how canonical paths (shortest paths within a class) can be identified and represented using tangent sequences, which define valid sequences of oriented polygons for navigating within a world of convex obstacles. By analyzing tangent sequences, the algorithm can potentially determine the most efficient path for the robot to travel.

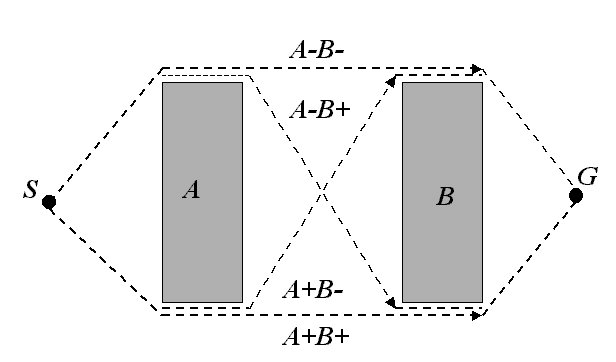
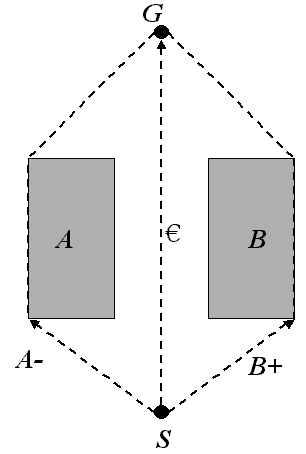


Figure 4.1. Examples of canonical paths

## Interpreting Tangent Sequences into Paths and Visibility

This text builds upon the previous explanation of canonical paths and tangent sequences, introducing how these sequences translate to actual paths and the concept of visibility in the context of path planning.

**Interpreting Tangent Sequences (I(σ)):**

* The text defines a function I(σ) that takes a tangent sequence (σ) and interprets it as a path (π).
* The specific details of this interpretation process are likely explained later, but it essentially translates the sequence of oriented polygons into a valid path for the robot to follow within the world.

**Canonical Paths (π):**

* A path (π) is considered **canonical** if there exists a corresponding tangent sequence (σ) such that π = I(σ).
* In simpler terms, a path is considered "shortest within its class" if it can be represented by a valid tangent sequence.

**Visibility in Free Space (free(W)):**

* The text introduces the concept of visibility between two points (p1 and p2) within the free space (free(W)).
* Two points are considered **visible** if there are no obstacles (polygons in W) that block a straight line segment connecting those points.
* This concept is crucial for path planning algorithms, as the robot needs to consider visibility between points when navigating the environment.

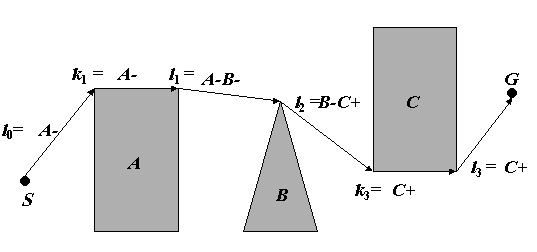


Figure 4.2. Tangent sequences A-B-C+ and Its canonical path

SG if visible (S, G)

(III.3)

Undefined SG if invisible (S, G)

**Where visible (S, G) means that S and G are visible in *W*:**

**Visibility for Start and Goal (visible (S, G)):**

* This notation signifies that the starting point (S) and the goal point (G) of the robot's path are visible within the world (W). In simpler terms, there are no obstacles blocking a straight line connecting S and G.

**Formal Interpretation of Tangent Sequence (I(σ)):**

* The formula I(σ) = lo k1l1…kqlq defines how a tangent sequence (σ) is interpreted as a path.
* Here's a breakdown of the formula:
  + σ = Bμ1i1…Bμqiq (represents the tangent sequence with oriented polygons and their signs)
  + q ≥ 1 (ensures there's at least one polygon in the sequence)
  + μ1, ..., μq ∈ {+1, -1} (signs indicating polygon orientations)
* The right side of the equation represents the concatenation (combination) of smaller sub-paths:
  + lo, l1, ..., lq (these represent sub-paths based on tangent lines)
  + k1, ..., kq (minimum portions of polygon boundaries)
* likely Figure 3.2

**Details of Sub-Paths and Visibility:**

* lo: This is a **μ1-tangent sequence** from the starting point (S) to the first polygon (Bi1) in the sequence. It considers the visibility between S and Bi1.
  + If this tangent is not visible (blocked by an obstacle), the entire path I(σ) is undefined.
* lj (for 1 ≤ j ≤ q-1): These represent **μj μj+1 common tangents** between consecutive polygons (Bij to Bij+1). These tangents connect points on the boundaries of adjacent polygons while considering visibility.
  + Similar to lo, if any of these common tangents is not visible, the path I(σ) is undefined.
* lq: This is a **μq-tangent sequence** from the last polygon (Biq) in the sequence to the goal point (G). It considers the visibility between Biq and G.
  + Again, if this final tangent is not visible, the path I(σ) is undefined.

**Minimum Portions of Boundaries (kj):**

* For each j (1 ≤ j ≤ q), kj represents the **minimum portion** (possibly empty) of the **μij-oriented boundary** of polygon Bij.
* This portion connects the point of intersection with the previous tangent (lj-1) and the point of intersection with the next tangent (lj+1) on the polygon's boundary.
* Essentially, kj defines the specific part of the polygon's boundary that the robot needs to traverse along the path.

**Proposition III.1** *For any given W, S and G, a canonical path is the shortest one among all paths in a homotopy class.*

*Proof*. Let σ be a tangent sequence such that π = I (σ). If σ = ε, the strait segment SG is the shortest. Let σ ≠ ε. Consider a normal at each osculating point (each endpoint) of every tangent,

li (0 ≤ i ≤ q), in π. The canonical path π and all paths which is paths which are homotopic to π intersects each of these normals exactly one time. If π intersects any of these normals other than the polygon boundary, each path segment becomes longer than lis and kis in the canonical path.

**Corollary III.1** *For given W, S and G, the shortest path is a canonical path.*

As the previous analysis shows, a locally shortest path (or canonical path) consists of a sequence of four kinds of path segments:

* A tangent from S to a polygon.
* Common tangents between two polygons.
* A tangent from a polygon to G.
* Portions of oriented polygon boundaries

Therefore, the shortest path finding problem is essentially a search problem in a world using tangents and oriented boundaries. Since there are only a finite number of tangents in the given world, this problem is converted into a graph search problem.

## 4.2 TANGENTS ON POLYGONS

This section dives into the details of finding tangents and common tangents between a ray (directed line) and a convex polygon.

**Tangents on a Convex Polygon (B):**

* The text defines a tangent on a convex polygon (B) using set theory notation:
  + L: The ray (directed line) in the free space surrounding the polygon (free(B)) or on the polygon's boundary (B).
  + [(L ∩ B) ≠ Ø]: Ensures the line intersects the polygon (B) at least once.
  + [(L ∩ int(B)) = Ø]: Ensures the line doesn't intersect the interior (int(B)) of the polygon.

**Osculating Point:**

* Based on the definition, a tangent line (L) must touch the polygon (B) at either:
  + A vertex (υ) of the polygon.
  + An edge (υφ(υ)) of the polygon (where φ(υ) refers to the next vertex in the counter-clockwise order).
* This point of contact between the tangent and the polygon is called the **osculating point**.

**Orientation of Tangents (Plus/Minus):**

* The text introduces the concept of tangent orientation based on the position of the polygon's interior:
  + **Plus Tangent (Counter-clockwise):** When the polygon's interior (int(B)) is on the left side of the tangent line (L).
  + **Minus Tangent (Clockwise):** When the polygon's interior (int(B)) is on the right side of the tangent line (L).

**Additional Notes:**

* This section focuses on convex polygons, where all interior angles are less than 180 degrees, and there are no inward corners.
* The concept of tangent orientation will likely be crucial for later discussions on finding common tangents between two polygons.

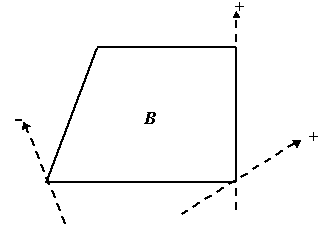


Figure 4.3. Tangents

### **4.2.1 Tangents from point to polygon**

Assume that we are given a polygon *B*, a point *p* Є free (*B*), and an orientation μ of tangents (μ = +1 or –1).

Let tangnet01 (*p*, *B*, μ, *v*) denotes a condition for the ray p*v* being a μ tangent on *B*.

Proposition III.2

(I) Tangnet01 (*p*, *B*, +1, *v*) =

[*ccw* (*p*, *v*, φ-1(*v*)) ^ *ccw* (*p*, *v*, φ(*v*))] v

[ *col* (*p*, *v*, φ(*v*)) ^ *ccw* (*p*, *v* ,φ-1(*v*)) ^ *ccw* (*p*, *v*, φ2(*v*))] (III.5)

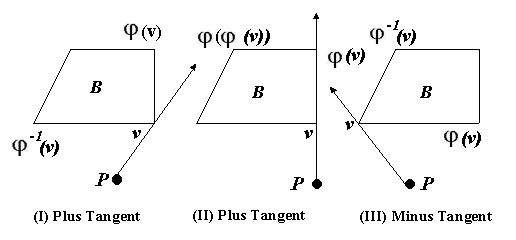


Figure 4.4 Tangents from point to polygon

**First Condition:** [ccw (p, v, φ-1(v)) ^ ccw (p, v, φ(v))]

This condition uses the concept of "ccw" (counter-clockwise). It might check if point p, vertex v, and its neighbors (φ-1(v) and φ(v)) form a counter-clockwise sequence.

If this condition is true, the tangent line likely touches vertex v and goes between φ-1(v) and φ(v).

**Second Condition:** [col (p, v, φ(v)) ^ ccw (p, v ,φ-1(v)) ^ ccw (p, v, φ2(v))]

This condition uses "col" (collinear) and "ccw". It might check if point p, vertex v, and its next vertex (φ(v)) are collinear (on the same line).

Additionally, it might check if point p, vertex φ-1(v), and the vertex two positions ahead (φ2(v)) form a counter-clockwise sequence.

If this condition is true, the tangent line likely coincides with the edge between v and φ(v) and goes between φ-1(v) and φ2(v).

**Understanding φ(v):**

* φ(v) refers to the next vertex of polygon B in the counter-clockwise direction from vertex v.

(II)

Tangnet01 (*p*, *B*, -1, *v*) =

[*cw* (*p*, *v*, φ-1(*v*)) ^ *cw* (*p*, *v*, φ(*v*))] v

[ *col* (*p*, *v*, φ-1(*v*)) ^ *cw* (*p*, *v* ,φ(*v*)) ^ *cw* (*p*, φ-1(*v*), φ-2(*v*))] (III.6)

Two conditions are presented, each likely defining the tangent line under different geometric circumstances:

**First Condition:** [cw (p, v, φ-1(v)) ^ cw (p, v, φ(v))]

This condition uses the concept of "cw" (clockwise). It might check if point p, vertex v, and its neighbors (φ-1(v) and φ(v)) form a clockwise sequence.

If this condition is true, the tangent line likely touches vertex v and goes between φ-1(v) and φ(v).

**Second Condition:** [col (p, v, φ-1(v)) ^ cw (p, v ,φ(v)) ^ cw (p, φ-1(v), φ-2(v))]

This condition uses "col" (collinear) and "cw". It might check if point p, vertex v, and its previous vertex (φ-1(v)) are collinear (on the same line).

Additionally, it might check if point p, vertex φ(v), and the vertex two positions before (φ-2(v)) form a clockwise sequence.

If this condition is true, the tangent line likely coincides with the edge between v and φ-1(v) and goes between φ-2(v) and φ(v).

**Understanding φ(v):**

* φ(v) refers to the next vertex of polygon B in the counter-clockwise direction from vertex v.

**Proposition III.3** we consider a problem of finding the osculating point (or the first osculating point if there are two) υ on B such that the ray pv is a μ tangent on B ("μ tangent " means plus tangent if μ = +1 and a minus tangent if μ = -1). This point (vertex) v is said to be a landing point in this problem (see Fig. 11). For any p, B, and μ, if p Є free (B), there exists one and only one μ tangent from p to B.

**Plus-tangent-from-point-to-polygon (*p*, *B*)**

begin

υ := first\_vertex (*B*)

**doforever**

**if** ccw (*p*, *v*, φ-1(*v*))

**then if** *ccw* (*p*, *v*, φ(*v*))

**then** return (*v*)

**else** *v* := φ(*v*)

**else** *v* := φ-1(*v*)

**end.**

In order to give a first idea, given a point p and a polygon *B*, an algorithm of finding the plus landing point (landing vertex) *v* is shown above. We assume the position of *p* is "general" so that only one vertex is on the resultant tangent on *B*. Here first\_vertex is defined as a function, which returns the first vertex of a polygon. We move *v* towards the right until it gets at the correct landing point through the next or previous function. Its computation time is O(|*B*|). We generalize this program so that it can handle the cases where μ = ±1 and the case where an edge (not only a vertex) is on the resultant tangent.

**Tangent-from-point-to-polygon (*p*, *B*, μ)**

begin

*v* := first\_vertex (*B*)

**doforever**

**if** *O*(*p*, *v*, φ-μ (*v*)) = μ

**then if** O(*p*, *v*, φμ (*v*)) ≠ -μ

**then** return (*v*)

**else** *v* := φμ (*v*)

**else** *v* := φ-μ (*v*)

**end.**

### **4.2.2 Tangents from Polygon to Point**

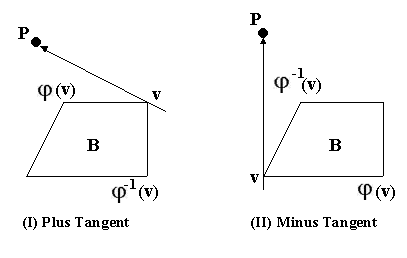
 Now, given a polygon *B*, a point *p* Є free (*B*), and an orientation μ, we consider a problem of finding the osculating point (or the second osculating point if there are two) *v* on *B* such that the ray *v*p is the μ tangent on *B*. This point (vertex) *v* is said to be a leaving point in this problem (see Figure 3.5).

Figure 4.5 Tangents from polygon to point

Let Tangnet10 (*p*, *B*, μ, *v*) denotes a condition for the ray *vp* being a μ tangent on *B*.

**Proposition III.4** *For any point p, convex polygon B, and μ,*

*if p Є free (B) and if v is a vertex on B,*

### 

*Tangnet*10 (*p*, *B*, μ, *v*) = *Tangnet*01 (*p*, *B*, -μ, *v*) (III.7)

Due to the last Proposition, we will use the point-to-polygon algorithm in the previous section to find polygon-to-point tangents instead of creating a new algorithm.

**Proposition III.5** *For any p, B, and μ, if p Є free (B), there exists one and only one μ tangent from p to B.*

### **4.2.3 Common Tangents Between Two Polygons**

Assume that we are given two convex polygons *B*1, *B*0. whose insides and boundaries are not intersecting. We are going to find rays which are tangents on both polygons. Because a ray can have an independent orientation on each polygon, there are four distinct orientations in all, *B*1+ *B*2+, *B*1+ *B*2-, *B*1- *B*2+, and *B*1- *B*2- (see Figure13). If the direction of a tangent is reversed, both orientations are also flipped. For instance, when *B*1+*B*2+ is considered a common tangent from *B*2 to *B*1 by changing its direction, that tangent becomes the *B*2-*B*1- tangent. We consider a problem of finding the osculating point (or the first osculating point if there are two) *v* on *B* such that the ray p*v* is a μ tangent on *B*.6

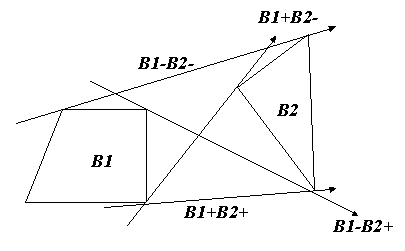


Figure 4.6 Tangents between two polygons

Let *v*1, *v*2 be vertices on polygons *B*1 and *B*2 respectively. If the ray *v*1V2 is a μ1 tangent on *B*1 and a μ2 tangent on *B*2, we write Tangent11 (*B*1, *v*1, μ1, *B*2, *v*2, μ2). We call the tangent a μ1 μ2 common tangent on *B*1 and *B*2.

**Proposition III.6** *For any polygons B1, B2, vertices v1, v2, and orientations μ1 and μ2, ifv1 is in B1 and υ2 is in B2,*

*Tangent*11 (*B*1, *v*1, μ1, *B*2, *v*2, μ2) =

*Tangent*01 (*v*1, *B*2, μ2, *v*2) ^ *Tangent*10 (*v*2, *B*1, μ1, *v*1). (III.8)

The proposition states that the common tangent between B1 and B2 (Tangent11) can be found by combining two simpler functions:

Tangent01(v1, B2, μ2, v2): This function, as discussed earlier, finds a tangent line from vertex v1 of B1 towards polygon B2 with orientation μ2, touching it at vertex v2.

Tangent10(v2, B1, μ1, v1): This function (likely defined elsewhere) might find a tangent line from vertex v2 of B2 towards polygon B1 with orientation μ1, touching it at vertex v1.

**Proposition III.7** For any polygons B1, B2, vertices V1, υ2, and orientations μ1 andμ2, ifv1 is in B1,v2 is in B2, and the insides and boundariesof these polygons are not intersecting, there exists one and only one μ1 μ2 tangent on these polygons*.*

The following algorithm finds a common tangent between two polygons.

This takes O (|*B*1| + |*B*2|), since it searches the solution on *B*1 and *B*2 alternatively.

**Tangent-between-two-polygons (*B*1, μ1, B2,μ2)**

**begin**

*v*1 := first\_vertex(*B*1)

*v*2 := first\_vertex(*B*2)

**do**

flag1 := flag2 := false

**if** *O* (*v*1,*v*2,φ-μ22 (*v*2)) = μ2

**then if** *O* (*v*1,*v*2,φμ22 (*v*2)) ≠ -μ2

**then** flag2 := true

**else** *v*2 :=φμ22 (*v*2)

**else** *v*2 :=φ-μ22 (*v*2)

**if** *O* (*v*2,*v*1,φ-μ11 (*v*1)) = -μ1

**then if** *O* (*v*2,*v*1,φμ11 (*υ*1)) ≠ -μ1

**then** flag1 := true

**else if** *υ*1 := φ-μ1 (*v*1)

**else** *v*1 := φμ1 (*v*1)

**until** flag1 ^ flag2

**return** (*v*1, *v*2)

**end.**

## 4.3 VISIBILITY

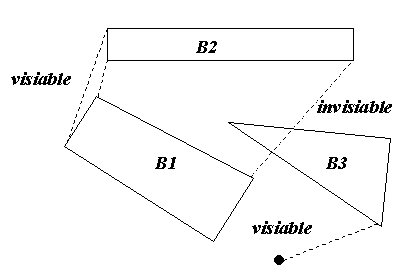
 In a world *W* with n polygons, there is a possibility that some tangents are “occluded” by one or more polygons. Let *p*1 and *p*2 be two points in the free space of *W*. The two points are said to be visible if there is no polygon *B* Є *W* such that the segment p1p2 has an intersection with B; otherwise they are said to be invisible (see Figure 14). Given a start S and goal G for a shortest path planning problem, we need to obtain three visible tangent sets: (i) The set of all visible tangents from S to polygons, (ii) the set of all visible from polygons to G, and (iii) the set of all visible common tangents among polygons. The visibility test is the most time-consuming subtask in the shortest path planning task.

Figure 4.7 Visibility

First consider the problem of finding all visible tangents from one point S to all oriented polygons in *W*. we compute all visible and invisible tangents *B*+ and *B*- for each polygon *B* from S without visibility tests. We obtain the landing vertices *v*+, *v*-, and their directions ψ+, and ψ- for each polygon *B*. This computation takes O (mn) time, where m is the maximum number of vertices in a polygon and n the number of polygons in *W* (see Figure 3.8)

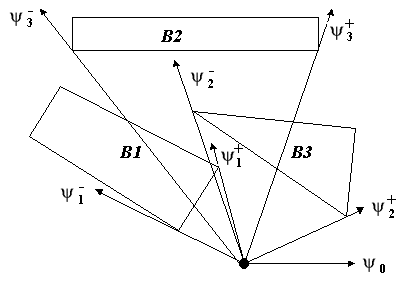


Figure 4.8 All tangents from S

After this preparatory step, we make an ordered list of tangents by its tangent directions ψμ. The visibility of a tangent from S depends on the “distance” between the polygon and S. we sweep a ray with a direction ψ from S keeping the information of the current list of polygons intersecting the ray as well as the order about closeness. As the ray sweeps, the list is updated at the direction of each tangents. Each updating operation is an insertion or a deletion of a polygon. A tangent is visible only when there is no polygon which is more closely located than the current one. This computation takes O(n log n) time. (This algorithm assumes that there is a direction ψ such that the ray

(S, ψo) does not intersect any polygons. If there is no such ψo, we need to slightly modify the algorithm.)

**Visible\_Tangents (*W*, S, G)**

**begin**

1. Make a list *L* of tangents *B*μ = (*vμ*,ψμ) from S

For each polygon *B* Є *W* and for and for μ = ±1 (visible or invisible).

1. Find ψo such that the ray (S, ψo) does not intersect any polygons.
2. Sort *L* by their directions ψμ based on ψo.
3. Initialize a priority queue P.
4. For each tangent *B*μ in *L* do
5. **If** μ = +1
6. **then** Insert *B* = (*v*+, ψ+, ν- , ψ-) in P.
7. **if** the *B* is placed in the “closest” position in P,
8. then Mark the *B*+ as visible.
9. **Else** Mark the *B*+ as invisible.
10. **Else** Find the corresponding *B* in P.
11. **if** the *B* is placed in the “closest” position in P.
12. **then** Mark the *B*- as visible.
13. **Else** Mark *B*- as invisible.
14. Delete the *B* from P.

**end.**

The “closeness” test actually should not be done by the simple length of each tangent, but the “orientation test” with the “diameter” *v*+*v*- of each polygon. The whole computation takes O (mn + n log n) = O (n (m + log n)).

If there is no ψo such that the ray (S, ψo) intersects no polygons, we rotate the ray one complete turn to initialize P correctly, and the main processing is done at second complete turn. This two complete sweep task does not increase the computational complexity.

The visibility testing for tangents from polygon to a point G is done in a similar manner. The visibility testing for tangents among polygons will be done in this way: (i) All tangents among polygons are obtained without visibility test. This computation takes O (mn2) time. (ii) Take one polygon B at one time. We will do a similar sweeping work to find visible common tangents. This task takes O (n2 log n) time. By applying all these algorithms to the example world, we obtain a result as shown in Figure 4.9

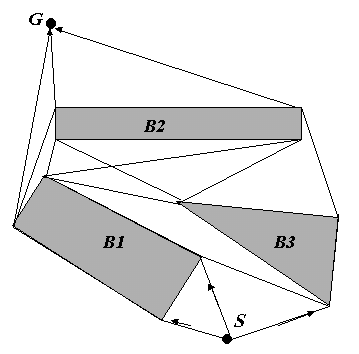


Figure 4.9 All visible tangents

## 4.4 SHORTEST PATH FINDING AGORITHM

We will describe the top-level algorithm in this section. The sub problems of finding tangents and common tangents are discussed in earlier sections. Since a canonical path is represented by a tangent sequence, we consider a “graph” consisting of the set of “nodes”,

{S, *B*1+, *B*1-, …, *B*n+, *B*n-, G}, (III.9)

and the set of “oriented edges” which is the set of tangents. This observation justifies the use a variation of the Dijkstra’s algorithm for graph search in this theory. However, this algorithm is not as simple as the basic Dijkstra’s algorithm, because a oriented polygon (as a “node”) is not a point, but a polygon. For instance, the fact that a landing vertex to and a leaving vertex of a oriented polygon may not be the same makes this planning task difficult. Before discussing the algorithm itself, we show a behavior of the algorithm for a world. The exact coordinates of two points and three convex polygons (See Figure 3.8).

S = (0, 0),

*B*1 = {ν11, ν12, ν13, ν14} = {(-15, 7), (-3, 1), (-1, 5), (-7,8)},

*B*2 = {ν21, ν22, ν23} = {(-3, 8), (6, 1), (9, 6)},

*B*3 = {ν31, ν32, ν33, ν34} = {(-10, 9), (8, 9),(8, 11), (-10,11)},

G = (-5, 16).

As the preprocessing, all visible tangents this world are first computed (see Figure 3.8). This problem is actually discussed in the previous section of this chapter.

For each oriented polygon z (S and G are also considered as special oriented polygons), we prepare the following data structure in order to run a generalized-Dijsktra’s algorithm.

(z.mark, z.cost, z.land, z.prev, z.leav, z.area) (III.10)

The meaning of each member is:

z.mark: =1 if z.cost is confirmed as the smallest one, =0 otherwise.

z.cost: the smallest cost to z known so far.

z.land: the landing osculating vertex on z.

z.prev: the previous oriented polygon or point.

z.leave: the leaving vertex of the previous oriented polygon.

z.area: the partial area of the current partial to z.

Let us give an example behavior of the algorithm. We start from S to enlarge the domain of “marked” oriented polygon until G itself is marked. A star in the “mark” slot means that this oriented polygon was newly marked. The z.area slot is not explicitly shown here.

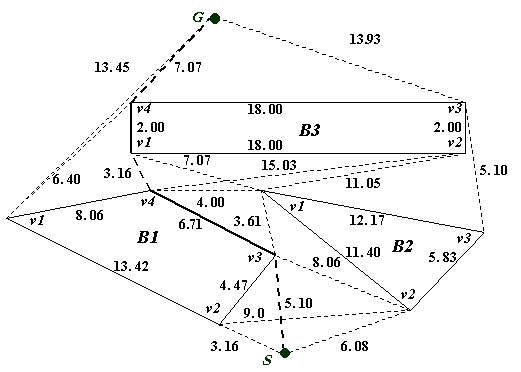


Figure 4.9 All visible tangents

**STEP 0**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| z | mark | cost | land | Prev | leave |
| S | **1\*** | **0.00** | **S** | **-** | **-** |
| B1+ | **0** | **∞** | **-** | **-** | **-** |
| B1- | **0** | **∞** | **-** | **-** | **-** |
| B2+ | **0** | **∞** | **-** | **-** | **-** |
| B2- | **0** | **∞** | **-** | **-** | **-** |
| B3+ | **0** | **∞** | **-** | **-** | **-** |
| B3- | **0** | **∞** | **-** | **-** | **-** |
| G | **0** | **∞** | **-** | **-** | **-** |

**STEP 1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| z | mark | cost | land | Prev | leave |
| S | **1** | **0.00** | **S** | **-** | **-** |
| *B*1+ | **0** | **5.10** | **V13** | **S** | **S** |
| *B*1- | **1\*** | **3.16** | **V12** | **S** | **S** |
| *B*2+ | **0** | **6.08** | **V22** | **S** | **S** |
| *B*2- | **0** | **∞** | **-** | **-** | **-** |
| *B*3+ | **0** | **∞** | **-** | **-** | **-** |
| *B*3- | **0** | **∞** | **-** | **-** | **-** |
| G | **0** | **∞** | **-** | **-** | **-** |

**STEP 2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| z | mark | cost | land | Prev | leave |
| S | **1** | **0.00** | **S** | **-** | **-** |
| *B*1+ | **1\*** | **5.10** | **V13** | **S** | **S** |
| *B*1- | **1** | **3.16** | **V12** | **S** | **S** |
| *B*2+ | **0** | **6.08** | **V22** | **S** | **S** |
| *B*2- | **0** | **28.64** | **V21** | **B1-** | **V14** |
| *B*3+ | **0** | **39.67** | **V32** | **B1-** | **V14** |
| *B*3- | **0** | **22.98** | **V34** | **B1-** | **V11** |
| G | **0** | **30.03** | **G** | **B1-** | **V11** |

**STEP 3**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| z | mark | cost | land | Prev | leave |
| S | **1** | **0.00** | **S** | **-** | **-** |
| *B*1+ | **1** | **5.10** | **V13** | **S** | **S** |
| *B*1- | **1** | **3.16** | **V12** | **S** | **S** |
| *B*2+ | **1\*** | **6.08** | **V22** | **S** | **S** |
| *B*2- | **0** | **8.71** | **V21** | **B1+** | **V13** |
| *B*3+ | **0** | **19.76** | **V32** | **B1-** | **V14** |
| *B*3- | **0** | **14.97** | **V31** | **B1+** | **V14** |
| G | **0** | **30.03** | **G** | **B1-** | **V11** |

**STEP 4**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| z | mark | cost | land | Prev | leave |
| S | **1** | **0.00** | **S** | **-** | **-** |
| *B*1+ | **1** | **5.10** | **V13** | **S** | **S** |
| *B*1- | **1** | **3.16** | **V12** | **S** | **S** |
| *B*2+ | **1** | **6.08** | **V22** | **S** | **S** |
| *B*2- | **1\*** | **8.71** | **V21** | **B1+** | **V13** |
| *B*3+ | **0** | **17.01** | **V33** | **B2+** | **V23** |
| *B*3- | **0** | **14.97** | **V31** | **B1+** | **V14** |
| *G* | **0** | **30.03** | **G** | **B1-** | **V11** |

**STEP 5**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| z | mark | cost | land | Prev | leave |
| S | **1** | **0.00** | **S** | **-** | **-** |
| *B*1+ | **1** | **5.10** | **V13** | **S** | **S** |
| *B*1- | **1** | **3.16** | **V12** | **S** | **S** |
| *B*2+ | **1** | **6.08** | **V22** | **S** | **S** |
| *B*2- | **1** | **8.71** | **V21** | **B1+** | **V13** |
| *B*3+ | **0** | **17.01** | **V33** | **B2+** | **V23** |
| *B*3- | **1\*** | **14.97** | **V31** | **B1+** | **V14** |
| G | **0** | **30.03** | **G** | **B1-** | **V11** |

**STEP 6**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| z | mark | cost | land | Prev | leave |
| S | **1** | **0.00** | **S** | **-** | **-** |
| *B*1+ | **1** | **5.10** | **V13** | **S** | **S** |
| *B*1- | **1** | **3.16** | **V12** | **S** | **S** |
| *B*2+ | **1** | **6.08** | **V22** | **S** | **S** |
| *B*2- | **1** | **8.71** | **V21** | **B1+** | **V13** |
| *B*3+ | **1\*** | **17.01** | **V33** | **B2+** | **V23** |
| *B*3- | **1** | **14.97** | **V31** | **B1+** | **V14** |
| G | **0** | **30.03** | **G** | **B1-** | **V11** |

**STEP 7**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| z | mark | cost | land | Prev | leave |
| S | **1** | **0.00** | **S** | **-** | **-** |
| *B*1+ | **1** | **5.10** | **V13** | **S** | **S** |
| *B*1- | **1** | **3.16** | **V12** | **S** | **S** |
| *B*2+ | **1** | **6.08** | **V22** | **S** | **S** |
| *B*2- | **1** | **8.71** | **V21** | **B1+** | **V13** |
| *B*3+ | **1** | **17.01** | **V33** | **B2+** | **V23** |
| *B*3- | **1** | **14.97** | **V31** | **B1+** | **V14** |
| G | **1\*** | **24.04** | **G** | **B3-** | **V34** |

By this result, the shortest path obtained is

G ← *B*3- (v34) ← *B*3- (v31) ←*B*1+ (v14) ←*B*1+ (v13) ← S (III.11)

And its tangent sequence is B1+ B3-. Now a formal algorithm is described here (the variable δ in line 14 will be described in the next sub section).

**Find\_Shortest\_Path (*W*, S, G)**

**begin**

1. Find all visible tangents from S
2. Find all visible tangents from each oriented polygon
3. **Forall** z Є Z U {S, G} **do**
4. Z.mark :=0;
5. z.cost :=0;
6. z :=S;
7. z..cost :=0;
8. z.land :=0;
9. z.area :=0;
10. **doforever**
11. z.mark :=1;
12. if z = G then return
13. forall z´ visible from z with leaving vertex ν0 on z and landing vertex ν1 on z´ do
14. w := z.cost + lg (z(z.land, ν0)) + d (ν0, ν1);
15. a0 := z.area + D (z.land, ν0) + D (ν0, ν1);
16. a1 := z´.area + D (z´.area , ν1) – a0;
17. **if** (z´.cost = ∞) V

(sign (*O*(z´)) = sign (a1) ^ w + lg (z´( ν1, z´.land)) < z´.cost) V (sign (*O* (z´)) ≠ sign (a1) ^ w – lg (z´ (z´.land, ν1)) < z´.cost)

**then**

1. z´.cost := w;
2. z´.land := ν1;
3. z´.prev := z;
4. z´.leave := ν0;
5. z´.area := a0;
6. z := the unmarked oriented polygon with the minimum z.cost;

**end**.

Here lg (z (u, u´)) the z-orientation boundary length of z polygon from u to u´ assuming u and u´ vertices of a z-polygon.

### 4.4.1 **Comparison between Two Paths**

The complex test in line 17 in the previous algorithm is related to the fact that there might be two paths reaching a same oriented polygon z' but at distinct landing vertices. Specifically, consider the situation depicted in Figure III.11. In this situation, the cost to z has been just established and there is a visible tangent ν0 ν1 from z to z', where a path to z' was already found with a landing vertex v'.land. Therefore, we must evaluate which one (the newly found one through z or the old one) is better. First task is to evaluate the new cost w to z' (at the new landing vertex v1) and the new partial area a0.

w = z.cost + lg (z (z.land, v0)) + d (v0, v1) (III.12)

a0 = z.area + D (z.land, v0) + D(v0, v1) (III.13)

In case (I) (Figure 3.10), the current cost z'.cost and w+lg (z' (v1, v'.land)) must be compared. However, in case (II), cost z'.cost and w-lg (z' (v'.land ,v1)) should be compared. If the former one is smaller in both caes, the new path is better, and we must replace the old one by the new one.

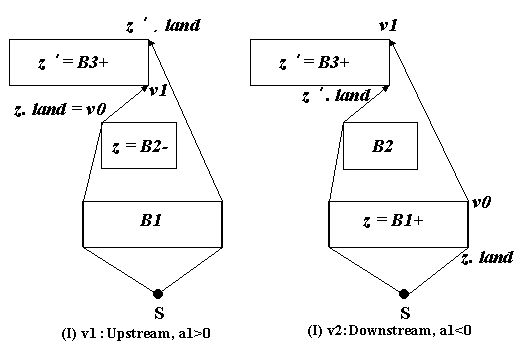


Figure 4.10. Comparison of paths

The next question is “which case the current situation falls into?” the best way is to compute the orientation of the polygon formed by two paths and the line segment v1, z'.land by computing its area. We compute the signed area

A1 = z'.area + D (z'.land, v1) – a0. (III.13)

Through testing of the sign of a1, we can correctly compare the old one and new path as shown in the previous program. We need a similar one like Figure III.11 with O (z') = -1, but it is omitted here.

CHAPTER.5

# CONCLUSION

This project addressed new algorithm in finding the shortest path for point robot in a world of convex polygons (which represent the projection of the obstacles in the 2-D). This shortest path algorithm was very successful on all of the problems attempted. This algorithm has been widely used because it does determine the existence of a free path and because it finds the shortest path if one is available.

Significant features of this method are:

1. A variation of the Dijkstra’s algorithm is used.
2. A directed polygon is the unit of the searching process instead of an edge. Since the number of directed polygons is 2*n*, this greatly reduces the computational complexity of this task as opposed to the known results.
3. As a part of item 2, an algorithm for comparing all paths reaching to the same directed polygon is established.

**Future work**

* + - Enhancing the safety framework with dynamic obstacle handling and real-time path adjustments.
    - Extending the algorithms to three-dimensional environments.
    - Integrating machine learning techniques to predict and adapt to changes in obstacle configurations.
    - Improving computational efficiency for larger and more complex scenarios.
    - Testing and validation in real-world applications, such as autonomous vehicles or robotic navigation in dynamic environments.

Chapter 6

# System Design

**1. Introduction**

This appendix provides a detailed design of the path planning application. The software is intended to compute the shortest path between a start and goal point while navigating around obstacles. The design ensures modularity, maintainability, and clear separation of concerns.

**2. System Architecture**

The system architecture consists of several interconnected components, each responsible for specific functionality within the application. The main components are:

1. **Graphical User Interface (GUI) Module**
2. **Path Planning Module**
3. **Polygon Identification Module**
4. **Utility Functions**

**3. Class and Module Descriptions**

**3.1 PathPlanner Class**

1. **Responsibilities**: Manages the core path planning logic, including setting start/goal points, running the pathfinding algorithm, identifying traversed polygons, and updating the graphical interface.
2. **Attributes**:
   1. start: Tuple representing the start point.
   2. goal: Tuple representing the goal point.
   3. polygons: Dictionary containing polygon data.
   4. polygon\_vertices: List of vertices defining the polygons.
   5. obstacle\_edges: List of edges defining obstacles.
   6. canvas: Canvas object for drawing.
   7. ax: Axes object for drawing.
   8. info\_label: Label object for displaying information.
3. **Methods**:
   1. set\_start\_goal(start, goal): Sets the start and goal points based on user input.
   2. run\_path\_planning(): Executes the path planning algorithm and updates the UI.
   3. identify\_traversed\_polygons(path): Determines which polygons are traversed by the path segments.
   4. get\_polygon\_to\_left(p1, p2): Identifies the polygon to the left of a path segment.
   5. get\_polygon\_to\_right(p1, p2): Identifies the polygon to the right of a path segment.
   6. draw\_path(path): Visualizes the computed path and labels the traversed polygons.

**3.2 Point Class**

1. **Responsibilities**: Represents a point in 2D space.
2. **Attributes**:
   1. x: X-coordinate of the point.
   2. y: Y-coordinate of the point.
3. **Methods**:
   1. \_\_init\_\_(x, y): Initializes the point with given coordinates.
   2. \_\_repr\_\_(): Provides a string representation of the point.

**3.3 Utility Functions**

1. **Responsibilities**: Provide helper functions for mathematical computations and data structures used in path planning.
2. **Functions**:
   1. generate\_visibility\_graph(vertices, edges): Generates the visibility graph from vertices and obstacle edges.
   2. create\_visibility\_graph(vertices, edges): Creates a graph representation for pathfinding.
   3. dijkstra(graph, start, goal): Finds the shortest path between two points using Dijkstra's algorithm.
   4. dijkstra\_with\_parents(graph, start, goal): Similar to dijkstra(), but also keeps track of parent nodes for path reconstruction.
   5. cross\_product(p1, p2, p3): Computes the cross product to determine the relative position of points.
   6. is\_point\_in\_polygon(point, polygon): Checks if a point is inside a given polygon.

**4. Detailed Design**

**4.1 Path Planning Workflow**

1. **Input Handling**: User inputs the start and goal points.
2. **Visibility Graph Generation**: The generate\_visibility\_graph function constructs a visibility graph from the polygon vertices and obstacle edges.
3. **Pathfinding**: The dijkstra\_with\_parents function computes the shortest path on the visibility graph.
4. **Polygon Identification**: The identify\_traversed\_polygons method determines which polygons are traversed by the computed path segments.
5. **Path Visualization**: The draw\_path method visualizes the path and labels the traversed polygons on the canvas.
6. **Output**: The result is displayed on the GUI, showing the path and the polygons traversed.

**4.2 GUI Design**

1. **Canvas**: Displays the map, obstacles, and path.
2. **Info Label**: Displays start/goal points, path distance, and traversed polygons.
3. **Buttons**: Allows users to set start/goal points and run the path planning algorithm.

**5. Algorithms**

**5.1 Dijkstra’s Algorithm**

* **Purpose**: To find the shortest path between two points in a weighted graph.
* **Steps**:
  + Initialize the distance to the start node as 0 and to all other nodes as infinity.
  + Use a priority queue to explore nodes with the smallest known distance.
  + Update the distance to neighboring nodes if a shorter path is found.
  + Continue until the goal node is reached or all reachable nodes are explored.

**5.2 Cross Product for Polygon Identification**

* **Purpose**: To determine the relative position of a point with respect to a polygon edge.
* **Steps**:
  + Calculate the cross product of vectors formed by three points.
  + Use the sign of the cross product to determine if the point lies to the left or right of the edge.

**6. Error Handling**

* Ensure start and goal points are valid and within the bounds of the map.
* Handle cases where no path can be found by displaying an appropriate message to the user.
* Validate user input to prevent invalid coordinates or malformed data.

**7. User Interface**

1. **Canvas Interaction**: Allows users to visually set the start and goal points and see the resulting path and obstacles.
2. **Information Display**: Provides real-time feedback on the current state of the pathfinding process and results.
3. **Control Elements**: Includes buttons and input fields for starting the pathfinding process and resetting the interface.

**8. Conclusion**

This software design provides a clear and structured approach to implementing the path planning application. By organizing the functionality into distinct modules and classes, the design ensures that the application is both modular and maintainable. The detailed descriptions and workflows outlined in this document serve as a comprehensive guide for development and future enhancements.

# APPENDIX A. SOURCE CODE

import tkinter as tk

from tkinter import ttk, messagebox

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.patches import Circle, Polygon

from collections import namedtuple

from queue import PriorityQueue

# Define a named tuple 'Point' to represent 2D points

#by defining x,y this rebresent  our environment in 2D

# the reason why we represent the envirenment in 2D is becuse the robot cannot work in 3D or 4D

#we rebresent it in 2D by input the coordenates of each polygon as x,y not by names

Point = namedtuple('Point', ['x', 'y'])

# This function checks if two line segments (p1, q1) and (p2, q2) intersect.

# Helper function to find the orientation of an ordered triplet (p, q, r).

def do\_edges\_intersect(p1, q1, p2, q2):

    def orientation(p, q, r):

        val = (q[1] - p[1]) \* (r[0] - q[0]) - (q[0] - p[0]) \* (r[1] - q[1])

        if val == 0:

            return 0  # collinear

        elif val > 0:

            return 1  # clockwise

        else:

            return 2  # counterclockwise

    # Helper function to check if point q lies on line segment pr

    def on\_segment(p, q, r):

        if q[0] <= max(p[0], r[0]) and q[0] >= min(p[0], r[0]) and \

           q[1] <= max(p[1], r[1]) and q[1] >= min(p[1], r[1]):

            return True

        return False

    # Find the four orientations needed for the general and special cases

    o1 = orientation(p1, q1, p2)

    o2 = orientation(p1, q1, q2)

    o3 = orientation(p2, q2, p1)

    o4 = orientation(p2, q2, q1)

    # General case

    if o1 != o2 and o3 != o4:

        return True

    # Special cases

    # p1, q1 and p2 are collinear and p2 lies on segment p1q1

    if o1 == 0 and on\_segment(p1, p2, q1):

        return True

    # p1, q1 and q2 are collinear and q2 lies on segment p1q1

    if o2 == 0 and on\_segment(p1, q2, q1):

        return True

    # p2, q2 and p1 are collinear and p1 lies on segment p2q2

    if o3 == 0 and on\_segment(p2, p1, q2):

        return True

    # p2, q2 and q1 are collinear and q1 lies on segment p2q2

    if o4 == 0 and on\_segment(p2, q1, q2):

        return True

    # If none of the cases apply, the segments do not intersect

    return False

# Check if edge is obstructed

def is\_edge\_obstructed(p1, p2, obstacles):

    for obstacle in obstacles:

        if do\_edges\_intersect(p1, p2, obstacle[0], obstacle[1]):

            return True

    return False

# Generate visibility graph with start and goal included

def generate\_visibility\_graph(vertices, obstacles, start=None, goal=None):

    vertices = list(vertices)  # Convert vertices to a list

    if start:  # Add start point to vertices if provided

        vertices.append(Point(\*start))

    if goal:  # Add goal point to vertices if provided

        vertices.append(Point(\*goal))

    edges = []  # List to store edges of the visibility graph

    # Iterate over all pairs of vertices

    for i in range(len(vertices)):

        for j in range(i + 1, len(vertices)):

            # Check if the edge between vertices[i] and vertices[j] is not obstructed

            if not is\_edge\_obstructed(vertices[i], vertices[j], obstacles):

                edges.append((vertices[i], vertices[j]))  # Add the edge to the list

    return edges  # Return the list of edges

# This function creates a visibility graph from a list of edges.

def create\_visibility\_graph(edges):

    visibility\_graph = {}  # Dictionary to store the visibility graph

    for edge in edges:

        if edge[0] not in visibility\_graph:

            visibility\_graph[edge[0]] = []  # Initialize list if vertex not in graph

        if edge[1] not in visibility\_graph:

            visibility\_graph[edge[1]] = []  # Initialize list if vertex not in graph

        distance = np.linalg.norm(np.array(edge[1]) - np.array(edge[0]))  # Calculate the distance between the two endpoints of the edge

        visibility\_graph[edge[0]].append((edge[1], distance))

        visibility\_graph[edge[1]].append((edge[0], distance))

    return visibility\_graph  # Return the constructed visibility graph

# This function implements Dijkstra's algorithm to find the shortest path in a graph.

def dijkstra(graph, start, goal):

    distances = {vertex: float('infinity') for vertex in graph}  # Initialize distances to infinity for all vertices

    distances[start] = 0  # Set the distance to the start vertex as 0

    priority\_queue = PriorityQueue()  # Create a priority queue

    priority\_queue.put((0, start))  # Add the start vertex with a distance of 0

    while not priority\_queue.empty():  # Loop until the priority queue is empty

        current\_distance, current\_vertex = priority\_queue.get()  # Get the vertex with the smallest distance

        for neighbor, weight in graph[current\_vertex]:  # Iterate over neighbors and weights of the current vertex

            distance = current\_distance + weight  # Calculate new distance to the neighbor

            if distance < distances[neighbor]:  # If new distance is smaller, update the distance

                distances[neighbor] = distance

                priority\_queue.put((distance, neighbor))  # Add the neighbor to the priority queue

    return distances[goal]  # Return the shortest distance to the goal vertex

# This function implements Dijkstra's algorithm and also tracks the parent vertices to reconstruct the path.

def dijkstra\_with\_parents(graph, start, goal):

    distances = {vertex: float('infinity') for vertex in graph}  # Initialize distances to infinity for all vertices

    distances[start] = 0  # Set the distance to the start vertex as 0

    priority\_queue = PriorityQueue()  # Create a priority queue

    priority\_queue.put((0, start))  # Add the start vertex with a distance of 0

    parents = {vertex: None for vertex in graph}  # Initialize parents dictionary to track the path

    while not priority\_queue.empty():  # Loop until the priority queue is empty

        current\_distance, current\_vertex = priority\_queue.get()  # Get the vertex with the smallest distance

        for neighbor, weight in graph[current\_vertex]:  # Iterate over neighbors and weights of the current vertex

            distance = current\_distance + weight  # Calculate new distance to the neighbor

            if distance < distances[neighbor]:  # If new distance is smaller, update the distance

                distances[neighbor] = distance

                priority\_queue.put((distance, neighbor))  # Add the neighbor to the priority queue

                parents[neighbor] = current\_vertex  # Update the parent of the neighbor

    return parents  # Return the parents dictionary

def cross\_product(p1, p2, p3):

    """Calculate the cross product of vectors p1p2 and p1p3."""

    return (p2.x - p1.x) \* (p3.y - p1.y) - (p2.y - p1.y) \* (p3.x - p1.x)

def is\_point\_in\_polygon(point, polygon):

    """Check if a point is inside a given polygon."""

    x, y = point.x, point.y

    n = len(polygon)

    inside = False

    px, py = polygon[0].x, polygon[0].y

    for i in range(n + 1):

        nx, ny = polygon[i % n].x, polygon[i % n].y

        if y > min(py, ny):

            if y <= max(py, ny):

                if x <= max(px, nx):

                    if py != ny:

                        xinters = (y - py) \* (nx - px) / (ny - py) + px

                    if px == nx or x <= xinters:

                        inside = not inside

        px, py = nx, ny

    return inside

def identify\_traversed\_polygons(self, path):

    traversed\_polygons = []

    for i in range(len(path) - 1):

        p1, p2 = path[i], path[i + 1]

        left\_polygon = self.get\_polygon\_to\_left(p1, p2)

        right\_polygon = self.get\_polygon\_to\_right(p1, p2)

        traversed\_polygons.append((left\_polygon, right\_polygon))

    return traversed\_polygons

def get\_polygon\_to\_left(self, p1, p2):

    for polygon\_name, polygon\_vertices in self.polygons.items():

        for vertex in polygon\_vertices:

            if cross\_product(p1, p2, vertex) > 0:

                if is\_point\_in\_polygon(vertex, polygon\_vertices):

                    return polygon\_name

    return None  # Return None if no polygon is found on the left

def get\_polygon\_to\_right(self, p1, p2):

    for polygon\_name, polygon\_vertices in self.polygons.items():

        for vertex in polygon\_vertices:

            if cross\_product(p1, p2, vertex) < 0:

                if is\_point\_in\_polygon(vertex, polygon\_vertices):

                    return polygon\_name

    return None  # Return None if no polygon is found on the right

# Class to represent the GUI

class PathPlanningGUI:

    def \_\_init\_\_(self, root):

        # Initialization

        self.root = root

        self.start = None

        self.goal = None

        self.setting\_start = False

        self.setting\_goal = False

        self.polygon\_vertices = [

            (1, 5), (1,4),(4,1),(1,8),(3,8),(7,1),(10,4),

            (5, 1), (5, 5), (5, 10),

             (10, 5)

        ]

        self.obstacle\_edges = [

            ((4, 4), (4, 1.5)), ((4, 4), (1.5, 4)), ((1.5, 4), (4, 1.5)),

            ((7, 2), (9.5, 4)), ((9.5, 4), (6, 4)), ((6, 4), (7, 2)),

            ((6, 6), (9, 6)), ((9, 6), (9, 9)), ((9, 9), (6, 9)),

            ((6, 9), (6, 6)), ((3, 6), (4, 7)), ((4, 7), (4, 8)),

            ((4, 8), (3, 9)), ((3, 9), (2, 8)), ((2, 8), (2, 7)),

            ((2, 7), (3, 6)),

            ((0, 0), (11, 0)),((0, 11), (0, 0)), ((11, 11), (11, 0)), ((11, 11), (0, 11))

        ]

        self.polygons = {

            "b01": [Point(0, 0), Point(0, 11),],  # Left edge of the environment

            "b02": [Point(0, 11), Point(11, 11)],  # Top edge of the environment

            "b03": [Point(11, 11), Point(11, 0)],  # Right edge of the environment

            "b04": [Point(11, 0), Point(0, 0)],  # Bottom edge of the environment

            "b1": [Point(4, 4), Point(4, 2), Point(2, 4)],

            "b2": [Point(7, 2), Point(9.5, 4), Point(6, 4)],

            "b3": [Point(6, 6), Point(9, 6), Point(9, 9), Point(6, 9)],

            "b4": [Point(3, 6), Point(4, 7), Point(4, 8), Point(3, 9), Point(2, 8), Point(2, 7)]

        }

        # Setup GUI

        self.frame = ttk.Frame(root)

        self.frame.pack(fill=tk.BOTH, expand=True)

        # Button Frame

        self.button\_frame = ttk.Frame(self.frame)

        self.button\_frame.pack(side=tk.TOP, fill=tk.X)

        # Button for running path planning

        self.run\_path\_button = ttk.Button(self.button\_frame, text="Run Path Planning", command=self.run\_path\_planning)

        self.run\_path\_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Button for resetting

        self.reset\_button = ttk.Button(self.button\_frame, text="Reset", command=self.reset\_gui)

        self.reset\_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Status Label

        self.status\_label = ttk.Label(self.button\_frame, text="Ready", foreground="green")

        self.status\_label.pack(side=tk.LEFT, padx=5, pady=5)

        # Canvas for Matplotlib plot

        self.fig, self.ax = plt.subplots()

        self.canvas = FigureCanvasTkAgg(self.fig, master=self.frame)

        self.canvas.draw()

        self.canvas.get\_tk\_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

        self.canvas.mpl\_connect("button\_press\_event", self.on\_plot\_click)

        # Sub-window to display start, goal, shortest path value, and path details

        self.info\_frame = ttk.Frame(self.frame)

        self.info\_frame.place(relx=1.0, rely=0.0, anchor='ne', x=10, y=10)

        self.info\_label = ttk.Label(self.info\_frame, text="Start: None\nGoal: None\nDistance: None\nPath: None")

        self.info\_label.pack(side=tk.TOP, padx=5, pady=5)

        self.plot\_obstacles()

    def reset\_gui(self):

        self.start = None

        self.goal = None

        self.ax.clear()

        self.plot\_obstacles()

        self.info\_label.configure(text="Start: None\nGoal: None\nDistance: None\nPath: None")

        self.canvas.draw()

    def plot\_obstacles(self):

        # Draw and label the polygons

        for name, vertices in self.polygons.items():

            polygon = Polygon(vertices, fill=None, edgecolor='k')

            self.ax.add\_patch(polygon)

            centroid = np.mean(vertices, axis=0)

            self.ax.text(centroid[0], centroid[1], name, fontsize=12, ha='center')

            self.ax.set\_xlim(-1, 12)

            self.ax.set\_ylim(-1, 12)

            self.ax.set\_xlabel("X")

            self.ax.set\_ylabel("Y")

            plt.axhline(y=10, xmin=0.31, xmax=0.847, color='black', ls='-')

            plt.axhline(y=5, xmin=0.155, xmax=0.847, color='black', ls='-')

            plt.axhline(y=1, xmin=0.39, xmax=0.61, color='black', ls='-')

            plt.axvline(x=10, ymin=0.39, ymax=0.85, color='black', ls='-')

            plt.axvline(x=5, ymin=0.157, ymax=0.849, color='black', ls='-')

            plt.axvline(x=1, ymin=0.39, ymax=0.69, color='black', ls='-')

            plt.plot(1, 5,'go', markersize=12.5, linestyle='dotted', alpha=0.4)

            plt.plot([1, 3], [8, 10],color='black',)  # Add line between (1,8) and (3,10)

            plt.plot([7, 10], [1, 4],color='black')  # Add line between (7,1) and (10,4)

            plt.plot(5, 1,'go', markersize=12.5, linestyle='dotted', alpha=0.4)

            plt.plot(5, 5,'go', markersize=12.5, linestyle='dotted', alpha=0.4)

            plt.plot(5, 10,'go', markersize=12.5, linestyle='dotted', alpha=0.4)

            plt.plot([7, 10], [1, 4],color='black',)  # Add line between (7,1) and (10,4)

            plt.plot([1, 4], [4, 1],color='black',)  # Add line between (1,4) and (4,1)

            plt.plot(10, 10,'go', markersize=12.5, linestyle='dotted', alpha=0.4)

            plt.plot(10, 5,'go', markersize=12.5, linestyle='dotted', alpha=0.4)

    def on\_plot\_click(self, event):

        if event.inaxes is None:

            return

        point = (round(event.xdata), round(event.ydata))

        if self.start is None:

            self.start = point

            self.ax.add\_patch(Circle(self.start, 0.3, color='g', label="Start"))

            self.canvas.draw()

        elif self.goal is None:

            self.goal = point

            self.ax.add\_patch(Circle(self.goal, 0.3, color='r', label="Goal"))

            self.canvas.draw()

            # Update info label with start and goal coordinates

            self.info\_label.configure(text=f"Start: {self.start}\nGoal: {self.goal}\nDistance: Calculating...\nPath: Calculating...")

    def run\_path\_planning(self):

        if self.start is None or self.goal is None:

            messagebox.showerror("Error", "Please set both start and goal points.")

            return

        vertices = [Point(\*v) for v in self.polygon\_vertices]

        obstacles = self.obstacle\_edges

        edges = generate\_visibility\_graph(vertices, obstacles, self.start, self.goal)

        graph = create\_visibility\_graph(edges)

        parents = dijkstra\_with\_parents(graph, Point(\*self.start), Point(\*self.goal))

        if Point(\*self.goal) not in parents or parents[Point(\*self.goal)] is None:

            messagebox.showerror("Error", "No path found.")

            return

        path = []

        current = Point(\*self.goal)

        while current:

            path.append(current)

            current = parents[current]

        path = path[::-1]

        distance = dijkstra(graph, Point(\*self.start), Point(\*self.goal))

        self.info\_label.configure(text=f"Start: {self.start}\nGoal: {self.goal}\nDistance: {distance:.2f}\nPath: {path}")

        self.ax.plot([p.x for p in path], [p.y for p in path], 'b-')

        self.ax.plot(self.start[0], self.start[1], 'go')

        self.ax.plot(self.goal[0], self.goal[1], 'ro')

        traversed\_polygons = self.identify\_traversed\_polygons(path)

        traversed\_polygons\_str = []

        default\_label\_counter = 1

        for i, (p1, p2) in enumerate(zip(path[:-1], path[1:])):

            left\_polygon, right\_polygon = traversed\_polygons[i]

            if left\_polygon is None:

                left\_polygon = f"b{str(default\_label\_counter).zfill(2)}"

                default\_label\_counter += 1

            if right\_polygon is None:

                right\_polygon = f"b{str(default\_label\_counter).zfill(2)}"

                default\_label\_counter += 1

            mid\_point = ((p1.x + p2.x) / 2, (p1.y + p2.y) / 2)

            label = f"({left\_polygon}, {right\_polygon})"

            self.ax.text(mid\_point[0], mid\_point[1], label, fontsize=12, color='blue')

            traversed\_polygons\_str.append(label)

            print(f"Segment {i}: {p1} to {p2}, Polygons: {label}")

        self.canvas.draw()

        traversed\_polygons\_str\_output = ', '.join(traversed\_polygons\_str)

        print(f"Traversed Polygons: {traversed\_polygons\_str\_output}")

        messagebox.showinfo("Path Info", f"Traversed Polygons: {traversed\_polygons\_str\_output}")

    def identify\_traversed\_polygons(self, path):

        traversed\_polygons = []

        for i in range(len(path) - 1):

            p1, p2 = path[i], path[i + 1]

            left\_polygon = self.get\_polygon\_to\_left(p1, p2)

            right\_polygon = self.get\_polygon\_to\_right(p1, p2)

            traversed\_polygons.append((left\_polygon, right\_polygon))

        return traversed\_polygons

    def get\_polygon\_to\_left(self, p1, p2):

        for polygon\_name, polygon\_vertices in self.polygons.items():

            for vertex in polygon\_vertices:

                if cross\_product(p1, p2, vertex) > 0:

                    if is\_point\_in\_polygon(vertex, polygon\_vertices):

                        return polygon\_name

        return None  # Return None if no polygon is found on the left

    def get\_polygon\_to\_right(self, p1, p2):

        for polygon\_name, polygon\_vertices in self.polygons.items():

            for vertex in polygon\_vertices:

                if cross\_product(p1, p2, vertex) < 0:

                    if is\_point\_in\_polygon(vertex, polygon\_vertices):

                        return polygon\_name

        return None  # Return None if no polygon is found on the right

    def plot\_visibility\_graph\_and\_path(self):

        # Generate the visibility graph

        visibility\_edges = generate\_visibility\_graph(

            [Point(\*vertex) for vertex in self.polygon\_vertices],

            [(Point(\*obstacle[0]), Point(\*obstacle[1])) for obstacle in self.obstacle\_edges],

            start=self.start,

            goal=self.goal

        )

        visibility\_graph = create\_visibility\_graph(visibility\_edges)

        parents = dijkstra\_with\_parents(visibility\_graph, self.start, self.goal)

        # Backtrack the path

        current = self.goal

        path = [current]

        while current in parents and parents[current] is not None:

            path.append(parents[current])

            current = parents[current]

        path.reverse()

        path\_coordinates = np.array(path)

        self.ax.plot(path\_coordinates[:, 0], path\_coordinates[:, 1], 'r-', label='Shortest Path')

        # Annotate shortest path

        shortest\_path\_value = dijkstra(visibility\_graph, self.start, self.goal)

        traversed\_polygons\_transitions = self.identify\_traversed\_polygons(path)

        traversed\_polygons\_text = ' -> '.join(traversed\_polygons\_transitions)

        # Update info label with calculated details

        self.info\_label.configure(text=f"Start: {self.start}\nGoal: {self.goal}\nDistance: {shortest\_path\_value:.3f}\nPath: {traversed\_polygons\_text}")

        # Update path details subwindow

        self.canvas.draw()

# Run the GUI

def run\_gui():

    root = tk.Tk()

    app = PathPlanningGUI(root)

    root.mainloop()

# Start the GUI

if \_\_name\_\_ == "\_\_main\_\_":

    run\_gui()

# APPENDIX B. USER GIUDE

**1. Introduction**

This user guide provides step-by-step instructions for using the path planning application. The application allows users to input a start and goal point on a 2D map with obstacles and computes the shortest path between them. The guide covers installation, usage, and troubleshooting.

**2. Installation**

**Prerequisites:**

* Python 3.x installed on your machine.
* Required Python packages: matplotlib, networkx, numpy, tkinter.

**3. Using the Application**

**3.1 Launching the Application:**

* After running path\_planner.py, a graphical user interface (GUI) will open, displaying the map with obstacles.

A screenshot of a computer

Description automatically generated

**3.2 Setting Start and Goal Points:**

* **Set Start Point:**
  1. Set Start point likely (1,1) .
  2. Click on the desired start location on the map. The start point will be marked on the map.

A screenshot of a computer

Description automatically generated

* **Set Goal Point:**
  1. Set goal point likely (1,1)
  2. Click on the desired goal location on the map. The goal point will be marked on the map.

A screenshot of a computer

Description automatically generated

**3.3 Running the Path Planning Algorithm:**

* After setting the start and goal points, click the "Run path planning" button.
* The application will compute the shortest path and display it on the map.
* The traversed path will be marked with line segments.
* The information about the traversed polygons will be displayed.

A screenshot of a computer

Description automatically generated

**Example 1:**

A screenshot of a computer

Description automatically generated

**Example 2 :A screenshot of a computer

Description automatically generated**

**Example 3:A screenshot of a computer

Description automatically generated**

**Example 4:A screenshot of a computer

Description automatically generated**

**3.4 Understanding the Output:**

* **Path Segments:**
  + The path is divided into segments, each represented by a line connecting two points.
  + Each segment is labeled with the polygons it traverses.
* **Information Display:**
  + The information label at the bottom of the GUI will show details such as the coordinates of the start and goal points, the length of the shortest path, and the names of the polygons traversed.

**3.5 Resetting the Interface:**

* To reset the map and input new start and goal points, click the "Reset" button.

A screenshot of a computer

Description automatically generated

**4. Troubleshooting**

**Issue: Application does not start.**

* Ensure that all required Python packages are installed.
* Check for any syntax errors or missing files in the project directory.

**Issue: Start or goal points are not being set.**

1. Make sure you click the "Set Start" or "Set Goal" button before clicking on the map.
2. Ensure you are clicking within the bounds of the map.

A screenshot of a computer

Description automatically generated

**Issue: Path is not found.**

* Verify that there is a clear path between the start and goal points without any obstacles blocking the way.
* If the path is still not found, check the terminal/command prompt for any error messages.

**Issue: Incorrect polygons are displayed.**

* Ensure that the polygons are correctly defined in the code.
* Verify that the polygon identification logic is correctly implemented.

A screenshot of a computer

Description automatically generated

**5. Conclusion**

This user guide provides all necessary instructions to install, use, and troubleshoot the path planning application. By following these steps, users can effectively find and visualize the shortest path between two points on a 2D map with obstacles.

# REFREN‎CES

[1] T. Lozano-Prez and M.A. Wasely. An Algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*,22(10):560-570, Oct. 2011.

[2] R.A.Brooks. Solving the find-path problem by good representation of free space. *IEEE transactions on systems, Man, and Cybernetics*, SMC-13(2):190-197, Mar./Apr. 2015.

[3] B.R.Donald. *Motion Planning With Six Degrees of Freedom*. Technical Report AIM-791, MIT Artificial Intelligence Lavoratory, 2009.

[4] R.A.Brooks and T.Lozano-Perez. A subdivision algorithm in configuration space for find path with rotation. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(2):225-233, Mar./Apr. 2016.

[5] P.E.Hart, N.J.Nilsson, and B.Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transaction of Systems Science and Cybernetics*, SSC 4(2): 100-107, July 2012.

[6] O.Khatib. Real time obstcale avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90-98, Spring 2005.

[7] C.W.Warren, J.C.Danos, and B.W.Mooring. An approach to manipulator path planning. *The International Journal of Robotics Research*, 8(5):87-95, October 2014.

[8] C.W.Warren. A vector based approach to path planning. *In Proceedings of the 2012 IEEE International Conference on Robotics and Automation*, pages 1021-1026, Sacramento, CA, 2013.

[9] M.Erdmann and T.Lozano-Perez. On multiple moving obstacles. In proceedings of the *The 2004 IEEE International Conference on Robotics and Automation*, pages 1419-1424, San Francisco, CA 2007.

[10] V.J.Lumelsky. Continuous motion planning in unknown environment for a 3D Cartesian robot arm. In *Proceedings of the The 2009 IEEE International Conference on Robotics and Automation*, pages 1050-1055, San Francisco, CA, 2009.

[11] D. Zwilling, *Statndard Mathematical Tables and Formulae*, CRC Pre