Design A Graphical Application For Robot Motion Planning

Jun Guan

Bachelor of Science in Computer Science with Honours
The University of Bath
04 2015

Design A Graphical Application For Robot Motion Planning

Submitted by: Jun Guan

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

In the modern day, the robot motion planning system has been widely used in many areas, such as, games, car navigation system and GPS.In this project, we designs a graphic application which could efficiently navigate a robot avoiding obstacles from its start position to its goal position through a shortest path. The project is split into two different situations which regarding to the shape of robot - point robot and rectangle robot.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank the project supervisor, Prof. Nicolai Vorobjov, for assistance and advice throughout the project. I would also like to thank Huiyu Qiu for her encouragement.

Chapter 1

Introduction

1.1 Description of the Problem

In this project, we will design a 2D graphical application that could navigate a robot avoiding obstacles from its start position to its destination by the shortest path. The user will be able to draw the environment by themselves for the robot. The application could cope with any environment and could efficiently find the shortest path. First of all, we will treat the robot as a point in the plane. In this basic situation, we need to solve several problems sequentially in order to find the optimal path. The first problem is how to draw the obstacles in the plane. Due to the fact that the convex polygons could be in any shape with any size, we plan to allow the user to place the polygon vertices on the plane. After that, we could use these vertices to draw the polygon by using corresponding algorithms, such as the Quick Hull algorithm and Graham scan algorithm. The second problem is how to find all the possible paths for the robot and also identify which one is the shortest path. For the possible paths, we could use the visibility graph, which consists of all the possible paths. Afterwards, we could calculate the distance of these paths by using the visibility graph to achieve the shortest path. While we want to calculate the optimal path, there are two different algorithms we can choose: Dijkstras algorithm and A* algorithm. Furthermore, after finishing the basic situation, we will investigate a complex situations where the robot is a rectangular without rotation. For this situation, one more steps will be added. We will need to grow the obstacles according to the robot's shape before calculating the visibility graph.

1.2 Aim and Objectives

The main aim of this project is to develop a graphical application that could navigate the robot from its start position to its end position through the shortest path among combined convex polygons. Besides this main aim, the project also has the following secondary objectives:

1. Build the environment for the robot:
   - research how to draw a convex polygon by using several points on a plane, mainly focussing on using the Quickhull algorithm
   - research how to relate the convex polygon position to the robot position
2. Find all the possible paths and identify the shortest path, then use it to navigate the robot from its start position to its destination.
   - research the different algorithms to grow obstacles according to the robot's shape(point robot and rectangular robot).
   - research how to detect a collision between robot and convex polygon
   - research how to obtain the visibility graph of the environment
   - research how to find the shortest path by using the visibility graph
3. Display the robot's environment and the optimal path on the user interface of application

   All of these aims and objectives are the tasks for the final year project. However, I am very interesting in robot motion planning. Therefore, I will try to adapt and consider one more complex situation in this project: a circular robot with rotation.

Chapter 2

Literature Review

2.1 Robot Motion Planning Problem

2.1.1 Robot Motion Planning Classification

The motion-planning problem is also known as the "piano moves problem". There are a number of ways to characterise the motion planning problem. The most vital characteristic of a motion planner is according to the problem it solves. Nowadays, we usually classify motion planning into two types according to its different functions: one is a static problem, in which the system knows all the information about the robot's environment before it starts to navigate the robot. The other one is a dynamic problem, in which the system only has partial information about the robots environment before it starts to navigate the robot.

For the dynamic problem, the robot may need a sensor to detect the information about the obstacle's size, shape and position when it tries to find its optimal path. For the static situation, the system will find the optimal path for the robot before it starts to move. Therefore, the optimal path will not be changed until its environment has changed. However, for the dynamic situation, due to the system only knowing partial information about its environment at the beginning, the optimal path may continue to change as more information is obtained. Different ways of motion planning will be used in different areas, both of which are important for the development of robot motion.

2.1.2 Robot Motion Planning Overview

The solution of a motion planning problem categorized can be analysed as four tasks: navigation, coverage, localisation and mapping. In this system, we will mainly focus on the navigation task. Navigation is the problem of finding a collision-free path to navigate the robot from its start position to its goal position. However, in our project, we will not only find the entire possible path for the robot but also identify which is the shortest safe path

for the robot. In this task, we will discuss the growing obstacles algorithm, configuration space, visibility graph and path search algorithm. Apart from the navigation, we also will discuss the mapping task in this system. Mapping is the problem of obtaining the information about the environment to construct a representation that is useful for us to navigate the robot(Howie Choset and Thrun, n.d.). In this project, all the information about the environment will be constructed for the robot before the system starts to navigate the robot. In the mapping task, we will discuss how to achieve the useful information of the environment from the application user interface and how to use these information to navigate the robot.

For this project, there are two important features for the robot path: the optimal path, which is calculated by the system and which should be "safe" and the "shortest" path. "Safe" means that the system should guarantee that the robot will not collide with any obstacles if it walks on the optimal path. In order to find a safe path for the robot, we need to calculate the configuration space of the environment (the configuration space is the space that the robot can achieve without any collision). For the same environment, the configuration space will be different for the different robot's shape. After finding the configuration space, we could draw the visibility graph in the environment. The visibility graph should consist of all safety paths for the robot. After then, we could focus on another important feature, which is to make sure the path identified for the robot is the shortest path. Therefore, we need to firstly consider the visibility graph for the environment space and apply some effective algorithms afterwards to find the shortest path.

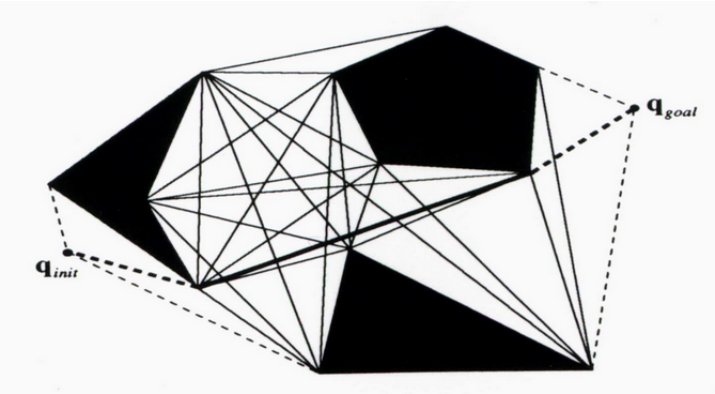


Figure 2.1: Overview of the robot motion-planning problem. This is the basic situation of this project that the robot is just a point. The black polygons are obstacles, the black lines are the visibility graph of this environment and the red line is the shortest safe path for the robot from its start point to its end point.(Joshua Fried and Pa, n.d.)

Generally speaking, in this chapter, we will discuss the methods of drawing convex hull, how to calculate the correct configuration space, how to draw the visibility graph and how to calculate the shortest path by using a visibility graph.

2.2 Convex Hull

2.2.1 What is Convex Hull?

First of all, we define S as the set of points in the plane, p and q are the points from set S. S is defined as convex if it contains a line segment that connects each pair of points. The convex hull that is created by points in S is the smallest convex polygon,such as the figure 2.2.

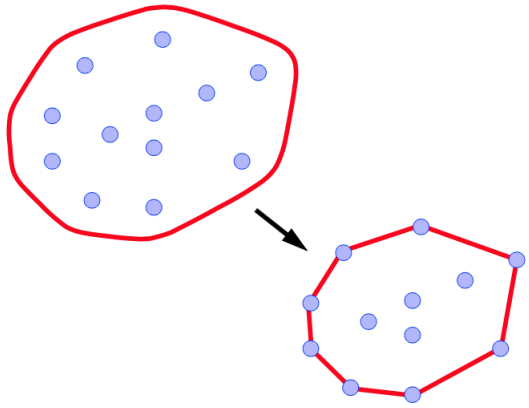


Figure 2.2: The right polygon is the convex hull that created by the points in the plane. (of Glasgow, n.d.)

So, we could now determine that if a polygon is convex by checking whether for any segment which by points p and q is non-intersecting. As the figure 2.3 shown:

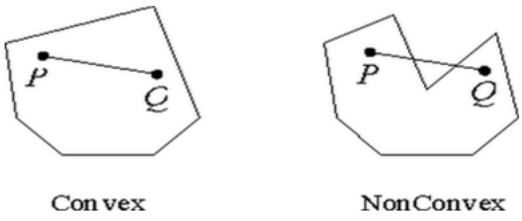


Figure 2.3: Convex Hull(Sunday, n.d.)

2.2.2 How to Draw a Convex Hull?

For this project, we will use a finite set of points to construct a convex hull in a 2-Dimensional space. There are many different algorithms which could be applied to draw a convex hull with different time complexity. For different algorithms, we will choose the optimal algorithm for our project based on the following factors: regarding to increase the

efficiency, the algorithm with less time complexity will have higher priority. However, if there exist two algorithms with same time complexity, we will use the algorithm that is relatively easy for implementation. Here is a list of potential algorithm descriptions.

- **QuickHull Algorithm**
  Time Complexity : $O(nlog(n))$, whereas in the worst case it takes $O(n^2)$.

  QuickHull algorithm shares some similarities with Quicksort, the core idea is an approach of divide and conquer. Firstly, QuickHull algorithm will find two points with which are the maximum and minimum value of x respectively. Secondly, we connect two points to form a line, and use this line to find two farthest points on its left and right side separately. Then the line with one of the farthest points could form a triangle and all points lying inside the triangle could be ignored. Then, the second step is repeated until all points are included in the convex hull.

  The QuickHull algorithm usually becomes slow when the points are spread in symmetry, however it is useful for solving the problems in which all points are spread in average. While the user using our application, they will know they are going to use these points to draw the obstacles so the points set by user should be always spread in average. Therefore, the QuickHull is suitable for our application.

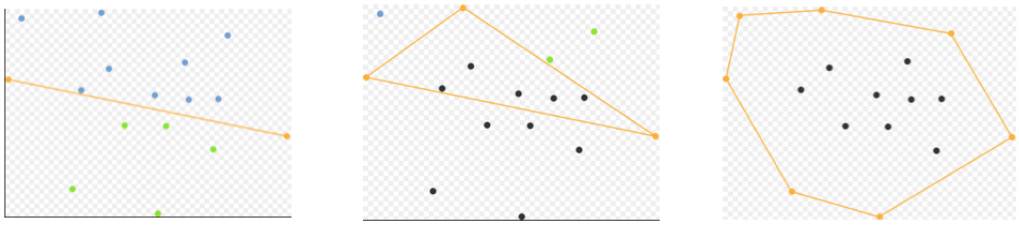


Figure 2.4: QuickHull Algorithm(Wikipedia, 2015)

- **Gift Wrapping Algorithm**
  Time Complexity : $O(nh)$
  n is the number of points and h is the number of the convex hull's node.

  Gift wrapping algorithm can be applied in a 2 dimensional space which were discovered by Jarvis. As we can see from the time complexity, the running time of Gift Wrapping Algorithm depends on both size of input and the size of output. Therefore, we regard this algorithm as an output-sensitive algorithm. Firstly, the algorithm will start to search the left-most point $P_i(i{=}0)$. Then, use this point to find another point P_{i+1} that to make all the points stay on the right side. The previous step will be continued until P_{i+1} is connected to P_0. The process is shown in Figure 2.5.

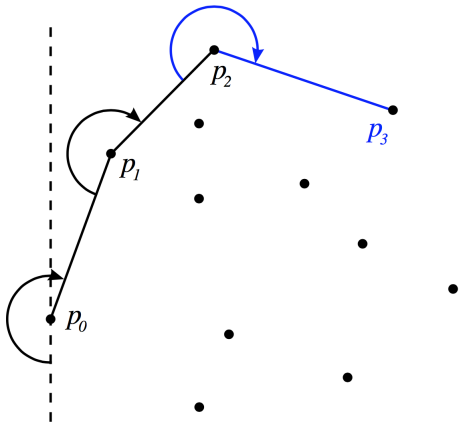


Figure 2.5: Gift Wrapping Algorithm (Wikipedia, 2015)

- **Graham Scan Algorithm**
  Time Complexity : O(n log n)

  The Graham Scan was discovered by Ronald Graham in 1972. This algorithm will start by the lowest point P_0 from the set of points. Next, it will use the start point P_0 to find point P_i and point P_{i+1}, after then the angle of the line is increased from line $\overline{P_0P_i}$ to $\overline{P_iP_{i+1}}$. This step will be continued until P_{i+1} could connect to the point P_0.The core idea of this algorithm is to decide whether the new line $\overline{P_iP_{i+1}}$ is "left turn" or "right turn". If the line formed by P_{i+1} and P_i is "left turn", then the point P_{i+1} is added to be one of the vertex of the convex hull, otherwise, we will drop the point and continue to test the next point.

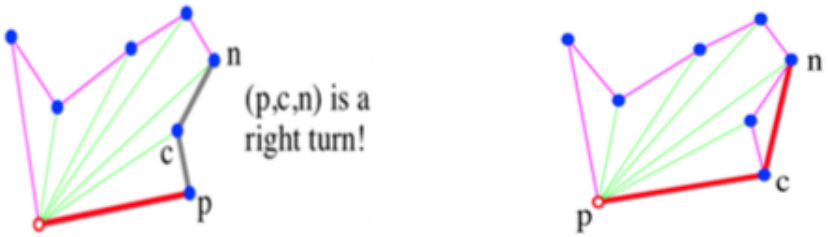


Figure 2.6: Right Turn and Left Turn. (of Glasgow, n.d.)

2.2.3 Algorithm Analysis

Comparing the 3 different Algorithms, we can notice that Graham Scan and QuickHull Algorithms have the same time complexity O(n log(n)), and the Gift Wrapping Algorithm

has the time complexity O(nh). Therefore, the Gift Wrapping Algorithm will be the most efficient algorithm only when h (the number of convex hulls node) is greater than log(n). However, this should be not happen in our project. In normal situations, the user will not use a huge number of points to draw one convex polygon, therefore h usually will cannot be greater than log(n). Hence, the Gift Wrapping Algorithm is not the best algorithm for our project. Due to Graham Scan and QuickHull Algorithm having the same time complexity, we will select the algorithm which is easier for the system to implement. As the QuickHull Algorithm is easy to understand and most people are familiar with this algorithm, we decided to use the QuickHull Algorithm to draw the obstacles in our project.

2.3 Robot's Configuration and Configuration Space

The configuration space of a robot is the space that the robot can achieve without collision. In order to ensure the robot will not collide with any obstacles, when the system starts to create motion plans for robot, it should know the specification of the every point's location. This specification is the configuration of the robot, and the configuration space is the space of all possible configurations of the system. Configuration space is generally non-Euclidean, which means it is not an n-dimensional Euclidean space R^n. The configuration's dimension is equal to the number of independent variables in the representation of the configuration, also known as degree of freedom (DOF). For example, the dimension of configuration space of a piano is 6: three for the position (x-y-z) and three for the orientation (roll-pitch-yaw).

In order to better illustrate the configuration space and the robot's configuration, we demonstrate an example here:
we have a circular robot with radius r in the plan and it will move without rotation. The robot's configuration is to specify the location of its center (x, y), relative to some fixed coordinate frame. Therefore, we could define the robot's configuration as q = (x, y), and we have

$$R(x, y) = \{(x^{'}, y^{'})|(x - x^{'})^2 \leq r^2\} \tag{2.1}$$

In this example, the parameters x and y are adequate to represent the configuration of the circular robot. Therefore, we can use the R^2 to represent the configuration space for the circle robot. However, the robot will be a point or a rectangle in our application, therefore, the configuration equation will be different.

2.4 Growing the Obstacles

At the basic case, the robot is a point; the obstacles represent the forbidden regions for the position of that point. However, in the real world, the robot cannot be a point, we need to grow the obstacles according to the robot's shape in order to acquire the correct configuration space. If the moving object is not a point, a new set of obstacles must be computed which are the forbidden regions. These new obstacles must describe the locus

of positions of this reference point which would cause a collision with any of the original obstacles(Lozano-Perez and Center, n.d.).

Now we firstly consider the robot as a circle with radius r. In the preview section 2.3, we have discussed the circular robot without rotation.However, there is a difference in our project, as the circular robot is allowed to rotate while moving. Therefore, the configuration space will be different as before for the vertex of polygon, as the figure 2.7 shows:

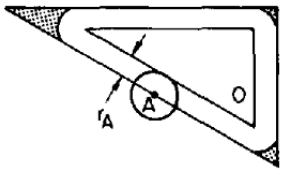


Figure 2.7: The shadow part is the modified part of the configuration if the robot is allowed to rotate. (Lozano-Perez and Center, n.d.)

While we want to grow the vertex of the polygon, we cannot just increase or decrease r to the x or y coordinate of the vertex to achieve the forbidden space. If we do this, as the figure 2.7 shows, it will waste some space near pointed corners. In order to obtain greater efficiency, we could slide and rotate the robot along the edge of the obstacles to produce accurate forbidden regions. By doing this, the robot will rotate around the vertex of the polygon while it need to walk around the polygon.

When robot is a rectangular robot, the first thing we need to do is to select a reference point for the robot to grow each vertex of the polygons. After that, we could construct the new growing polygons by growing vertex of polygon, as shown in Figure 2.8

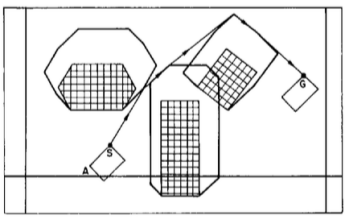


Figure 2.8: Growing obstacles for a rectangular robot (Lozano-Perez and Center, n.d.)

The different reference point we selected will have the different growing obstacles. In this application, we decide to use the central point of the rectangular robot as the reference point.

2.5 Visibility Graph

The defining characteristics of the visibility graph are that its nodes share an edge if they are within line of sight of each other, and that all points in the robot's free space are within line of sight of at least one node of the visibility map. (Howie Choset and Thrun, n.d.) Therefore, we could use the visibility graph to calculate all the possible paths for the robot from its start position to its destination. Here is the example of the visibility graph:

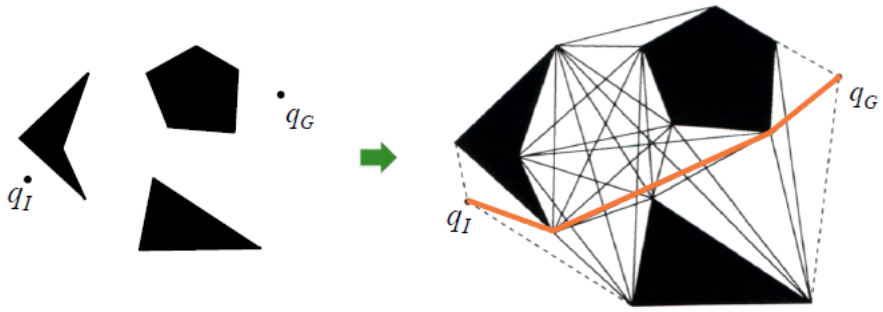


Figure 2.9: Visibility Graph (Joshua Fried and Pa, n.d.)

There are many methods which could be used to construct the visibility graph, and we will choose the most efficient one. Here is method that we will use to build the visibility graph: we use V to represent the set of all vertices of polygons in the configuration space as well as the start position and the end position. After then, we connect all the segments $\overline{v_i v}$ that do not intersect with any polygons. For each segment, $v, v_i \in V$, $v \neq v_i$. The set of these segments $\overline{v_i v}$ are the visibility graph.

2.6 Finding the shortest path

We have discussed the visibility graph in the previous section, now we could use the visibility graph to find the shortest path for the robot. There are many different Algorithms which could be used to find the shortest path. For this project, we mainly discuss the A* Algorithm and Dijkstra's Algorithm. In our project, the canvas of our application is a collection of nodes and edges. Hence, we could use the G=(V,E) to represent the robot's environment: G is the graph, V is the vertex of the polygon and E is the edge of polygon. Even though the A* Algorithm and Dijkstra's Algorithm both could both be used in our project, each of them still has distinct features. We now compare the two algorithms and discuss the advantage and disadvantage of each algorithm.

2.6.1 A* Algorithm

The worst case space complexity is:

$$O(|V|) = O(b^d) \tag{2.2}$$

b is the branching factor and d is the distance of the shortest path

Description

The A* Algorithm is one of the well-known informed search algorithms with many application in robotics. The A* Algorithm combined the D* Algorithm and best-first algorithm, and uses a best-first search and find a least-cost path for the robot from its start position to its destination in the node-and-edge graphs.

Firstly, for the A* Algorithm, we define the cost function as:

$$f(n) = g(n) + h(n)$$

- g(n) is the real cost for the robot to move from its start position to the current node n.
- h(n) is estimated cost for the robot to move from its current node n to its destination.

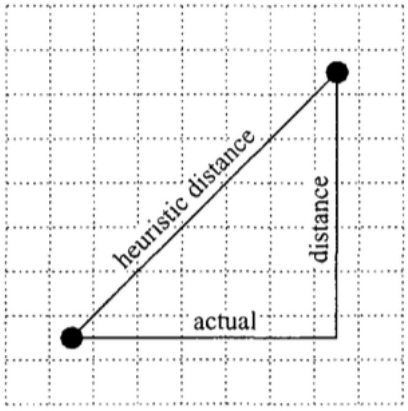


Figure 2.10: Heuristic cost (Howie Choset and Thrun, n.d.)

For the heuristic path-cost function h(n) in the equation, it must be an admissible heuristic which means the heuristic cost never overestimate the true cost. Therefore, in order to make sure the h(n) is the an admissible heuristic, the h(n) is usually a straight line from its current position to the destination, such like the figure 2.10 shows. Furthermore, if we want to use the A* Algorithm in our project, we need to assumes that the heuristic function

h(n) is monotonic. "Monotonic" means the heuristic function h(x) satisfies the condition for every edge of the graph(d(x,y) is the length of the edge):

$$h(x) \leq d(x, y) + h(y) \tag{2.3}$$

In such a case, when the heuristic function is monotonic, A* Algorithm will be more efficient and also, it will be equivalent to the D* Algorithm with the reduced cost

$$d'(x, y) := d(x, y) + h(y) - h(x)$$

(Wikipedia, 2015)

How it works?

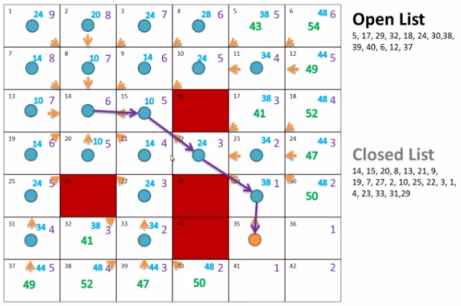


Figure 2.11: A* Algorithm example. (Panagiotis, n.d.)

Now, we will discuss in detail that how the A* Algorithm work in the system. First of all, we will have two sets in the A* algorithm, one is the open set and another one is the closed set. The open set is used to store the nodes that have been evaluated and the closed set is used to store the nodes that have not evaluated yet.

At the begin of algorithm, the closed set will be a empty set and the open set just contain the start node. After then, we will find all the nodes which are adjacent to the start node and add them into the open set. After these new nodes added into the open set, we set the start node as the parent node and remove the start node from the open set to the closed set. After then, we will choose the node in the open set which have the lowest f(n) to test if it is the destination. If it is the destination then we finished research, otherwise, we will keep check whether it has already in the closed set. if the test node is in the closed set, we will do nothing and continue to test next node in the open set. If it is in the open set, we will need to test if current node is the better node by compare the cost of g(n). If the real cost of the path is lower, we will set the current node as the parent node and continue to test its adjacent node. We will continue test all the possible nodes until robot arrives the destination or the open set is empty.If the system stopped because of the empty open set, it means the robot cannot move from its start position to its destination

2.6.2 Dijkstra's Algorithm

The worst case space complexity is:

$$O(ElogV) \tag{2.4}$$

V is the number of vertices of the polygons, E is the number of edges of the visibility graph.

Description

So far we have discussed the A* Algorithm, which is an informed search algorithm. This means that the system must know all the information about its environment and the environment is considered as static. However, in our project, the application should allow the user to modify the environment (adding obstacles after the optimal path has been calculated) and also change the size and shape of the robot. The growing obstacles will be different, if the size and shape of the robot is changed. Therefore, the robot's environment will need to be updated. We term such environments dynamic. The Dijkstra's algorithm is always used in unknown, partially known or dynamic environments that the planned path may need to be recalculated. The Dijkstra's algorithm was devised to "locally repair" the graph allowing for an efficient updated searching in dynamic environments(Howie Choset and Thrun, n.d.). Therefore, the Dijkstra's algorithm is more efficient than the A* algorithm in our project.

As we use the Dijkstra's algorithm to find a shortest path from a given source to another given source in a graph,hence, the motion problem in our project is also called the single-source paths problem.

Improve the Efficiency

If we decided to use the Dijkstras Algorithm to find the shortest path in our project, there will exist two ways that can be used to improve the Dijkstra's algorithm's efficiency. According to the Dijkstras Algorithm's time complexity, the first method is to decrease the number of edges of the visibility graph. The second method is to decrease the number of vertices of the polygons. For example: in our application, the system should allow the user to draw the obstacles anywhere by themselves. So it is possible that the user draw two intersected obstacles in the environment. In such situation, we could combine the two obstacles to create a new convex hull. If we do like this, it will decrease not only the number of vertex in the environment but also the number of edges of the visibility graph. Finally the efficiency of the Dijkstras Algorithm will be increased.

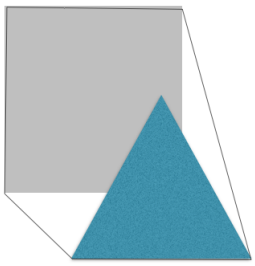


Figure 2.12: Combine obstacles to increase the efficiency

Pseudocode and example

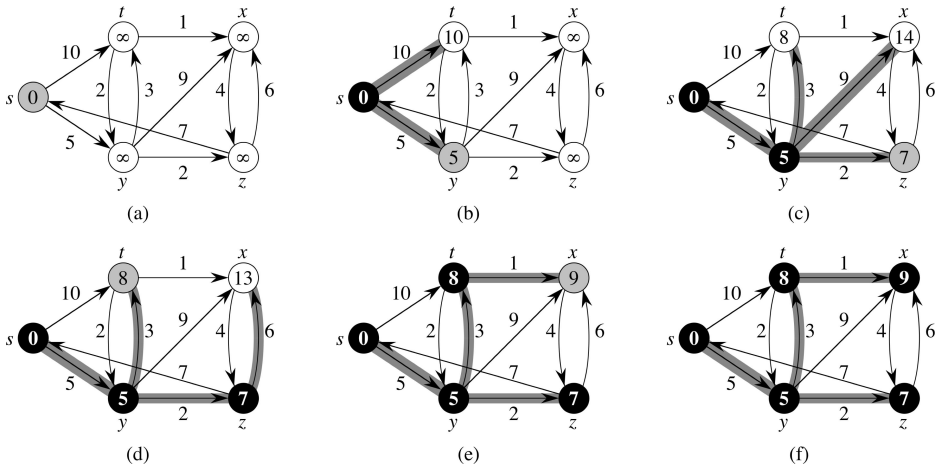


Figure 2.13: Dijkstra's Algorithm example.(USTC, n.d.)

As the Figure 2.2 shown, the Dijkstra's algorithm is an iterate algorithm. It starts from a given node and continuously calculates,compares and updates the distance from the start node to the longer nodes until it finds the shortest path from the start node to the end node. To be more specific, the Dijkstras algorithm can be consisted of three stages:

1. First of all, initial the distance of start point as 0 and set the previous point of start point as undefined.

2. After that we define all the nodes in the environment with the distance infinite(the max integer number).

3. Finally, we start the iterative loop until it find the optimal path for the robot. This stage is the central part of Dijkstra Algorithm's .

Table 2.1: Dijkstras algorithm pseudocode

Dijkstra's algorithm pseudocode
Input: the set of nodes in the environment and the start points
Output: shortest path for robot between start and goal nodes
Function DijkstraAlgorithm(setOfNodes, start): distance[start]:=0 previ[start]:=undefined **for every** nodes n of environment: previ[n] := undefined distance[n]:= Integer.Max() **While setOfNodes is not** empty: u:=node in setOfNodes has the distance[ ] remove u from setOfNodes **for every** neighbor n of u: altP := distance[u]+distaceBetween(u,v) if altP < distance[n] then distance[n] := altP previ[n]:=u **return** previ[ ]

Chapter 3

Requirements

3.1 Requirements Overview

The aim of this section is to describe the functional and non-functional requirements. In order to build an efficient and user-friendly application, we will try to fulfill all the requirements that we have specified.

The aim of this project is to produce an application that could navigate the robot avoiding obstacles from its start position to its destination through the shortest path in a 2-dimensional environment. We will mainly focus on how to find the correct and safe shortest path efficiently for the robot by choosing an appropriate algorithm in the program. The application should be easy to use for the user and it should run without any error when the user uses the application.

In the following sections, we will use the use case to describe the relationship between our application and the user. After that, we will apply the use case to specify the functional requirement and the non-functional requirement. We produce the use case firstly due to the face that it reflects the highest-level goals that users want to achieve and these goals could be considered as requirements for the application.

3.2 Use Cases

Use Cases are used to describe how the user interact with the application to achieve their respective goals. It shows the relationship between the user and application directly.

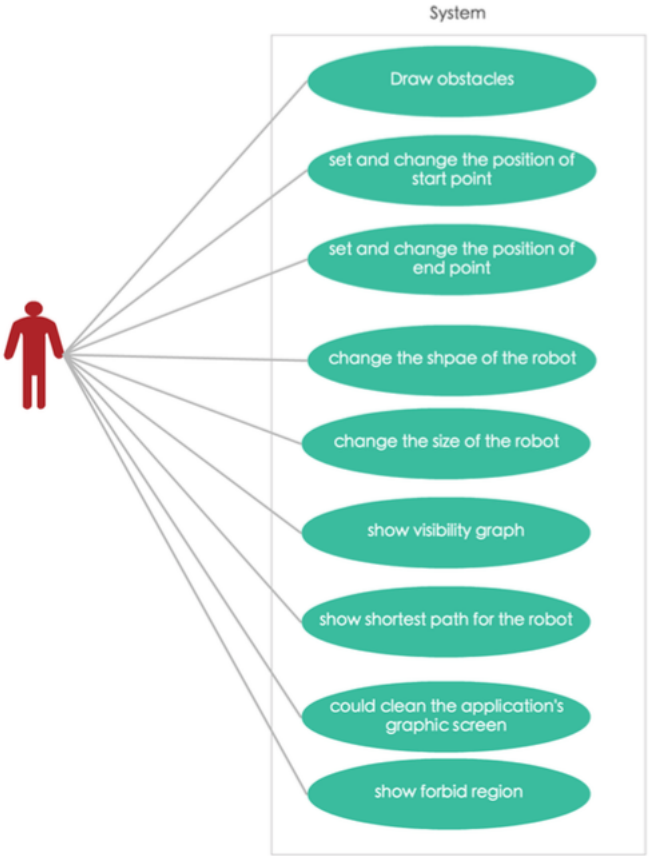


Figure 3.1: User Case Diagram

3.3 Functional Requirements

Functional requirements are used to specify some features or functions that the system is supposed to accomplish. According to the previous use case diagram, we can produce the requirements as follow:

ID: 01
Requirement: The user should be able to set the start and end positions of the robot
Priority: High
Rationale: The user could set the robot's start and end positions anywhere they want. The start position and the end position should be easily distinguished by applying different colours for those two different positions.

ID: 01.1
Requirement: The user should be able to change the start position and end position of the robot even though the positions have already been set previously.
Priority: High

Rationale: The use may want to change the start position or end position of the robot for various reasons. It is an important feature for the user to test this application because different positions will lead to different shortest path.

ID: 02
Requirement: The user should be able to draw the obstacles at any position on the canvas.
Priority: High
Rationale: It is important for the user to be allowed to build the environment for the robot themselves. There should not be any constraints for users to build various environments. To do so, the user could test the system's correctness under various environments.

ID: 02.1
Requirement: The application should allow the user to draw obstacles of any shape. .
Priority: High
Rationale: The robot motion planning system will be applied in a real world environment. So it would be unlikely that all obstacles would have the same shape. Therefore we ought to allow users to build a complex environment for the robot in the application.

ID: 02.2
Requirement: The application should allow the user to have an unobstructed view of all the obstacles in the plane
Priority: Medium
Rationale: The user should always know the state of the environment that they have already built. It would be helpful for the user to build a complex environment if the application could display all the obstacles in one plane.

ID: 03
Requirement: The application should display the visibility graph for the robot on the plane.
Priority: High
Rationale: The visibility graph will help the user to understand the robot's motion problem and at the same time, the user could use the visibility graph to check if the robot is going along the correct shortest path.

ID: 03.1
Requirement: The user should be able to set the visibility graph to invisible if they want.
Priority: Low
Rationale: In some complex environments, the plane will have many visibility lines for the robot. It may not be convenient for the user to check the shortest path if all visibility lines are shown on the screen.

ID: 04
Requirement: The user should be allowed to clean the canvas of the application.
Priority: Medium
Rationale: The user may want to retest our system in a new environment. In this situation, it will be convenient for the user to clean the current plane instead of reopening our application.

ID: 05
Requirement: The system should be able to find the correct shortest path and show it on the screen.
Priority: High
Rationale: Finding the correct shortest path is the main objective in this project. By showing it on the screen, users could test whether the answer is correct.

ID: 06
Requirement: The system should allow the user to change the robot's shape to a circle.
Priority: High
Rationale: A circular robot is representative in the real world. Therefore, it would be significant for our system has function that to set the robot's shape as a circle.

ID: 06.1
Requirement: The user should be able to change the radius of the circle robot.
Priority: Medium
Rationale: The size of the robot are not the same in the real world. Hence, it is essential to allow users to change the radius of the robot. And also, it will help the user to test our application usability by changing this variable.

ID: 07
Requirement: The system should allow the user to change the robot's shape to a rectangle.
Priority: High
Rationale: A rectangular robot is representative in the real world. Therefore, it is necessary for our system to be able to set the robot's shape as a rectangle.

ID:07.1
Requirement: The system should allow the user to change the length and width of the rectangle robot.
Priority: Medium
Rationale: The size of the robot is not fixed in the real world. Hence, it is essential to allow the user to change the radius of the robot. And also, it will help users to test our application's correctness by changing this variable.

ID:08
Requirement: The system should display the forbidden space on the screen.
Priority: High
Rationale: As we have discussed in the literature review, if the robot is not a point, there will exist a forbidden space for the robot in the plane. Hence, it is essential to illustrate the forbidden region on the screen in order to make the robot's motion problem easier to understand and also help the user to evaluate the application's correctness.

3.4 Non-Functional Requirements

A non-functional requirement is used to specify criteria that can be used to judge the operation of our system. For this project, it should achieve following non-functional requirements fully as much as possible:

ID: 09
Requirement: The application's user interface should be user-friendly.
Priority: High
Rationale: Many non-professional users may use our application to learn about robot motion problem. It is important for this type of users to easily understand how to operate this application.

ID 10
Requirement: The application should be able to respond to user's instruction quickly.
Priority: Medium
Rationale: It will improve the system's efficiency and save the user's time.

ID: 11
Requirement: The application should have high testability.
Priority: Medium
Rationale: It is essential for user to check this system's correctness. The the user should easily observe and understand the system's outcomes.

Chapter 4

Design

4.1 Instruction

We have completed the literature review and set the requirements for this project. Now, we could decide and construct the structure of our program according to the requirements. In order to design an efficient system structure, we will mainly focus on the following aspects: System Architecture, Class Diagram and Sequence Diagram. Constructing these structure diagrams will help us to constrain our application and to fulfill the requirements effectively.

4.2 System Architecture

A system architecture or systems architecture is the conceptual model that defines the structure, behaviour, and more views of a system.(Wikipedia, 2015) There exist several different types of system architectures, such as hardware architecture, software architecture and enterprise architecture. Here we will only discuss the software architecture. And also there are many different architectural patterns and styles that apply to the software architecture. For example: Backup, Client-server, Data-centric and Layered. We will select the most appropriate structure for our system.

As our application is just stand-alone and we do not have any client and server, the choice of architectural patterns is limited for us. After carefully researching these different software architectures, we decided to use the Model-View-Controller paradigm (MVC) model in our application. The MVC model consists of three roles: model, view and controller. These objects are separated by abstract boundaries which makes MVC more of a paradigm rather than an actual pattern since the communication with each other across those boundaries is not further specified. (Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, 2010) In the MVC model, the model object is used to hold data and define the logic will be applied when we implement that data. For instance, in our project, the model object will be the main part for storing all the information about the environ-

ment. Additionally, this object includes all the algorithms which are used to calculate the shortest path for the robot. The view object is something visible in the user interface, for example, in our project, the canvas, button panel and description panel are all view objects. The view object is used to display information in order to interact with users. The Controller object in the MVC model is used to connect the Model and View objects by transferring data between the two objects.

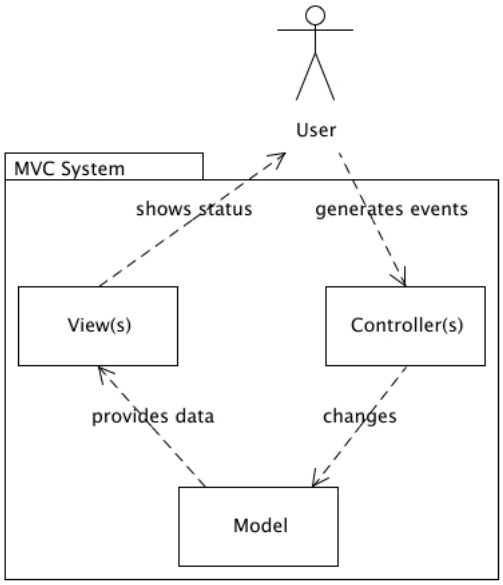


Figure 4.1: Model-View-Controller Model. (Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, 2010)

4.3 UML model

The Unified Modelling language (UML) is the blueprint for a system. It is used to visualise a system's architecture in a diagram. We have discussed the System architecture in the previous section, hence we could now design the UML model for the system according to its system architecture. The UML model is divided into two categories: structural information and behaviour information. We will mainly use the class diagram to discuss the structural information and apply sequence diagram to discuss the behavioural information of our system. By using the UML models we could ascertain how the Model, View and Control objects interact with each other in detail, and also what is the behaviour of each object.

4.3.1 Class Diagram

The class diagram is used to show the static structure of the system. It shows all the classes, attributes and functions in the system, and also the relationships among those classes. Each block of the diagram will represent a class in the system. Such as the figure 4.2:

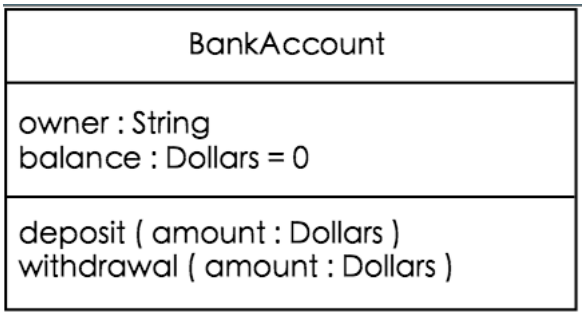
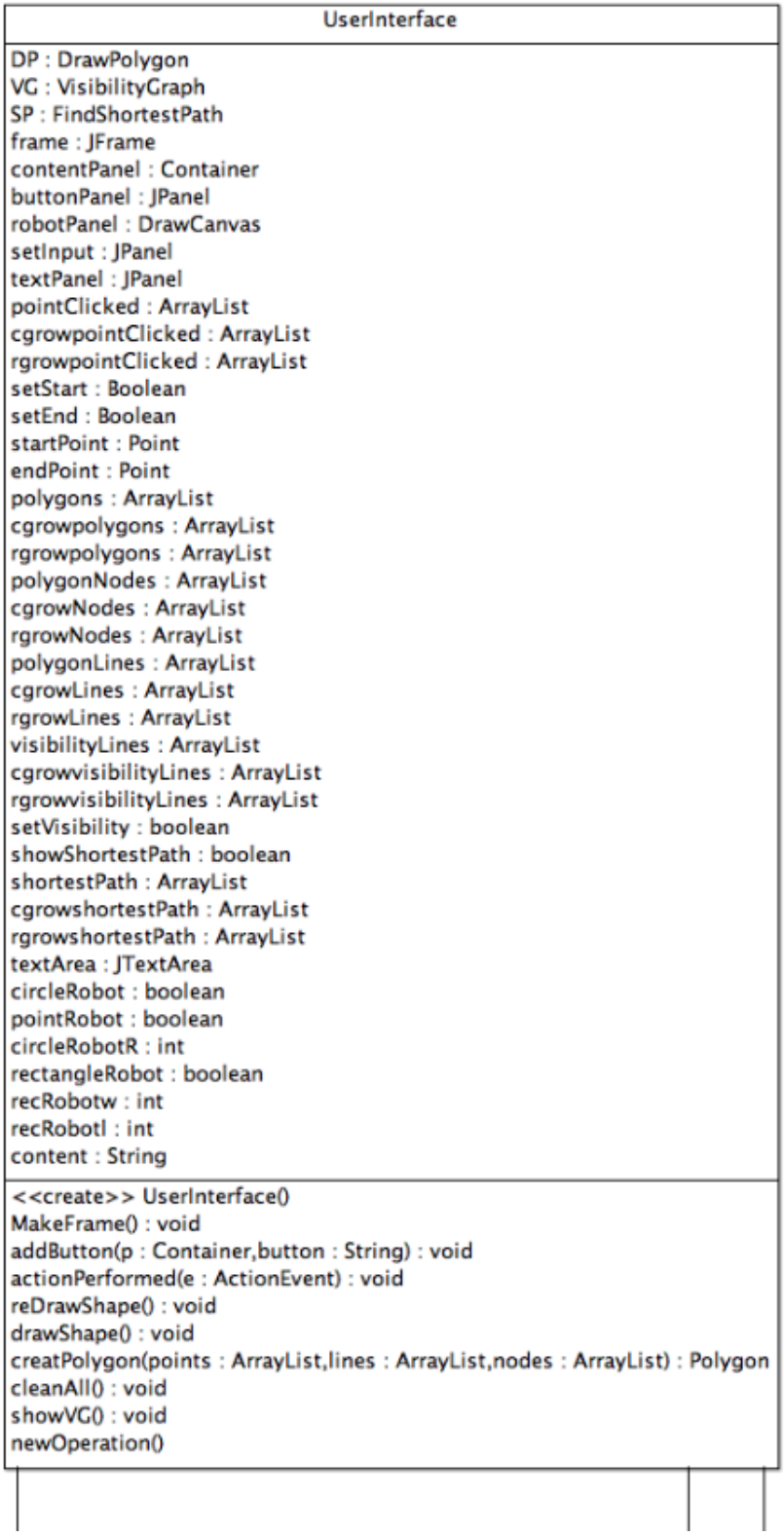


Figure 4.2: Class diagram example

In the Figure4.2, the first line in the block is the name of class, the second line contains the attributes of the class and the bottom line are the functions with its own parameters in the class. The class diagram is clear to show all the classes' information and their relationships, therefore, we could easily transfer the Class diagram to our program code.

Figure 4.3 is the Class Diagram that designed for our project:


UserInterface
DP : DrawPolygon
VG : VisibilityGraph
SP : FindShortestPath
frame : JFrame
contentPanel : Container
buttonPanel : JPanel
robotPanel : DrawCanvas
setInput : JPanel
textPanel : JPanel
pointClicked : ArrayList
cgrowpointClicked : ArrayList
rgrowpointClicked : ArrayList
setStart : Boolean
setEnd : Boolean
startPoint : Point
endPoint : Point
polygons : ArrayList
cgrowpolygons : ArrayList
rgrowpolygons : ArrayList
polygonNodes : ArrayList
cgrowNodes : ArrayList
rgrowNodes : ArrayList
polygonLines : ArrayList
cgrowLines : ArrayList
rgrowLines : ArrayList
visibilityLines : ArrayList
cgrowvisibilityLines : ArrayList
rgrowvisibilityLines : ArrayList
setVisibility : boolean
showShortestPath : boolean
shortestPath : ArrayList
cgrowshortestPath : ArrayList
rgrowshortestPath : ArrayList
textArea : JTextArea
circleRobot : boolean
pointRobot : boolean
circleRobotR : int
rectangleRobot : boolean
recRobotw : int
recRobotl : int
content : String
<<create>> UserInterface()
MakeFrame() : void
addButton(p : Container,button : String) : void
actionPerformed(e : ActionEvent) : void
reDrawShape() : void
drawShape() : void
creatPolygon(points : ArrayList,lines : ArrayList,nodes : ArrayList) : Polygon
cleanAll() : void
showVG() : void
newOperation()

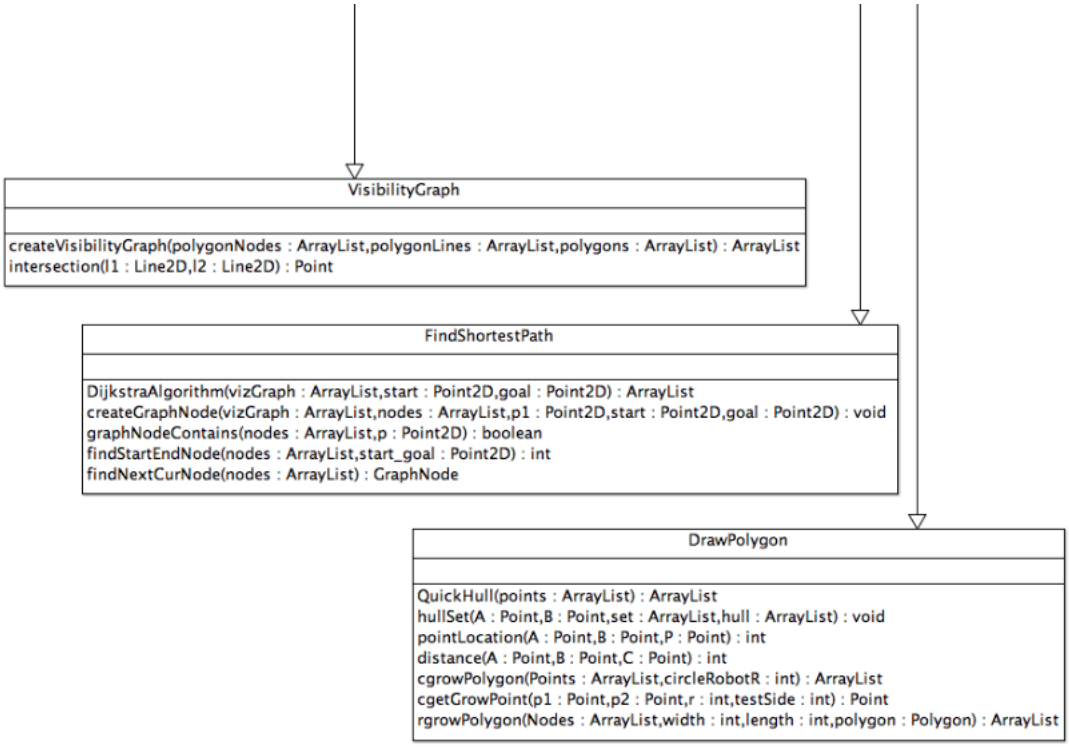


Figure 4.3: Class diagram

4.3.2 Sequence Diagram

A sequence Diagram(as shown in Figure 4.4) is also known as an event diagram. We can obtain a large amount of information from the sequence diagram, such as: the interaction between classes, the time sequence of each operation and the logic of the system. By producing this kind of sequence diagram we could see the interactive behaviour that exists in our system.

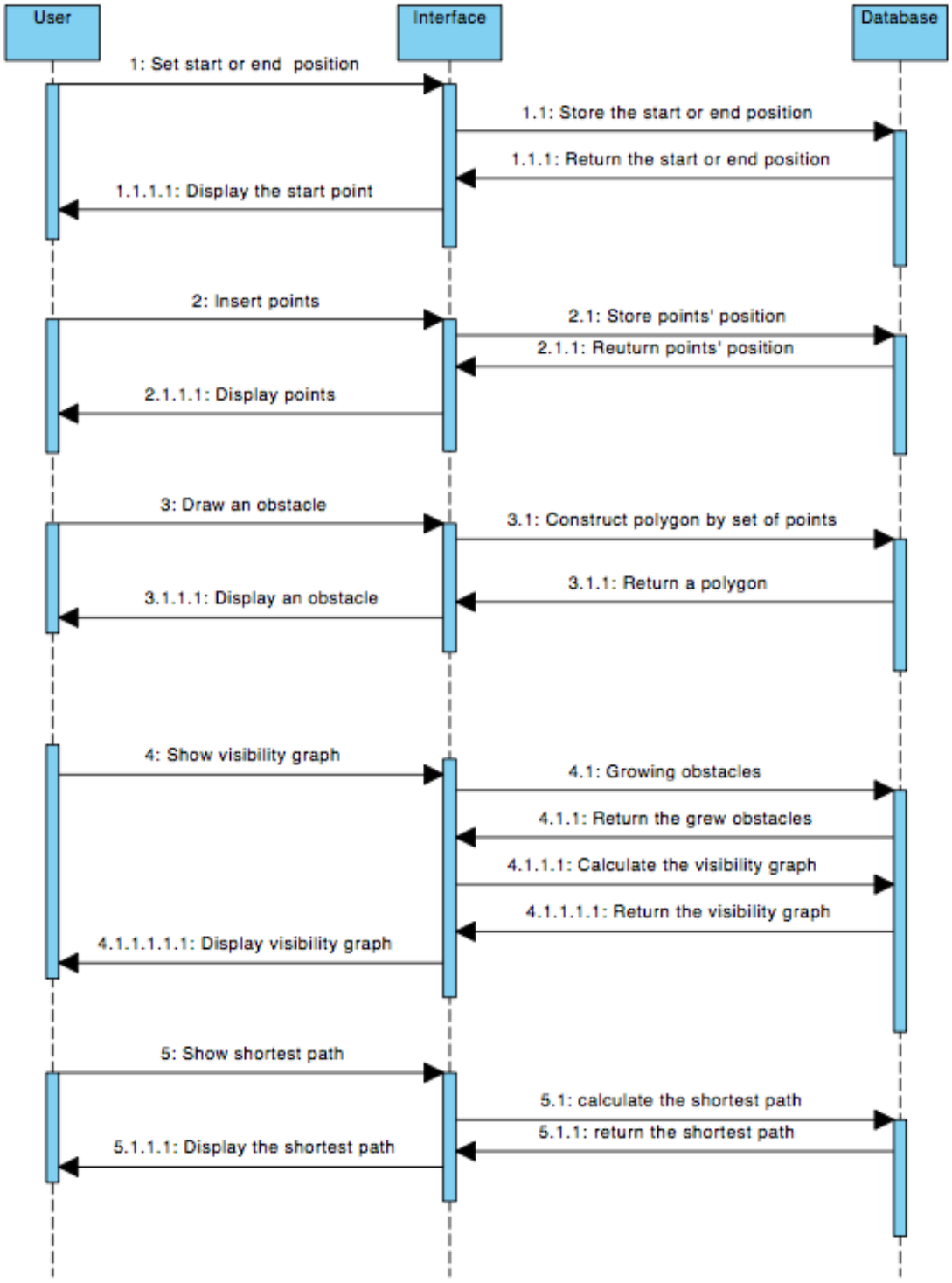


Figure 4.4: Sequence Diagram

The figure 4.5 is an example that shows the operation underwent if user want to change the size or shape of robot.

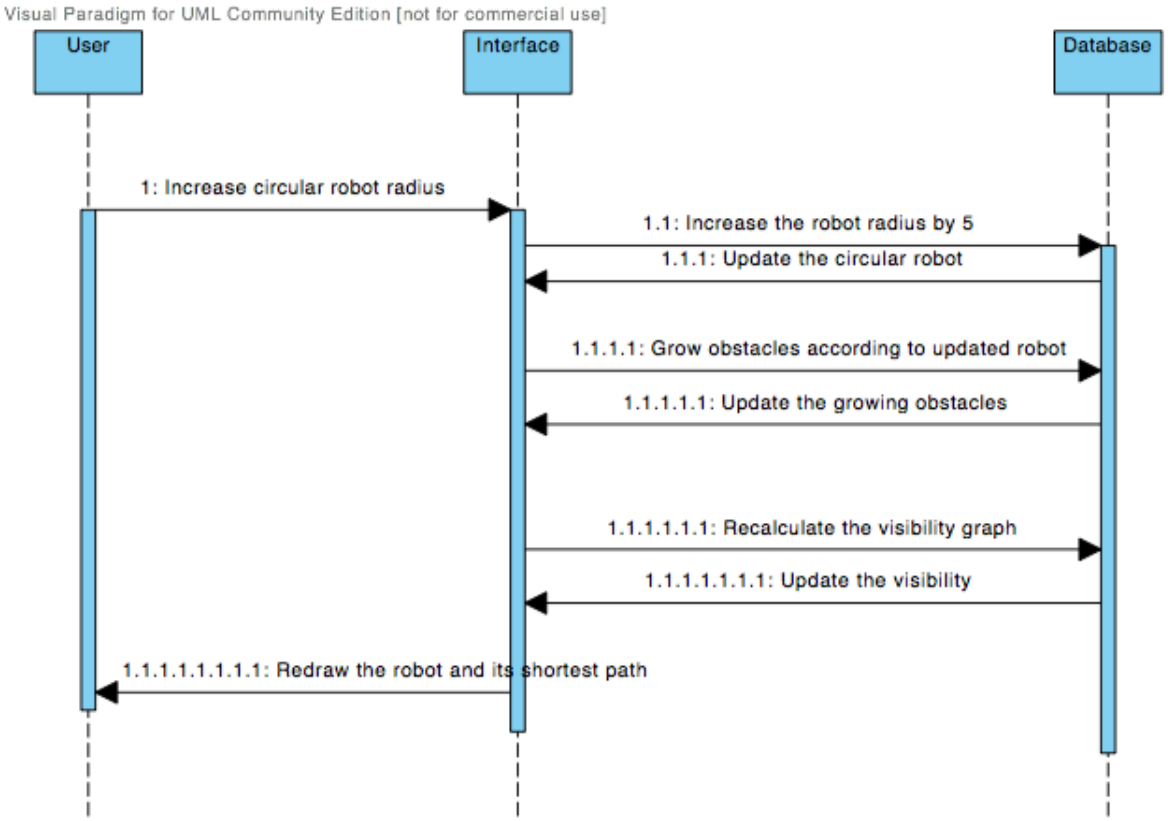


Figure 4.5: Sequence Diagram

4.3.3 Algorithm Design

In the literature review, we have discussed all the algorithms that we will adapt in the project. Now, we are going to discuss the algorithm structure in the system. As we have decided to employ the MVC model to be the system's structure, all the algorithms will be contained in the model object of our project. These algorithms should be used step by step in our application. The sequence of algorithms could be sorted into an order. In this project, the first task is to create the obstacles to build the environments for the robot. Therefore, the Convex Hull algorithm will be applied first. After the environment has been built, we need to grow the obstacles according to the shape of the robot. For different shapes of the robot, we need to apply different algorithms. After the obstacles have grown, we will use the corresponding algorithm to draw the visibility graph for the environment.

Finally, according to the visibility graph, we will use the Dijkstra's Algorithm to find the shortest path to navigate the robot. Generally speaking, we will design the algorithm structure as follow:

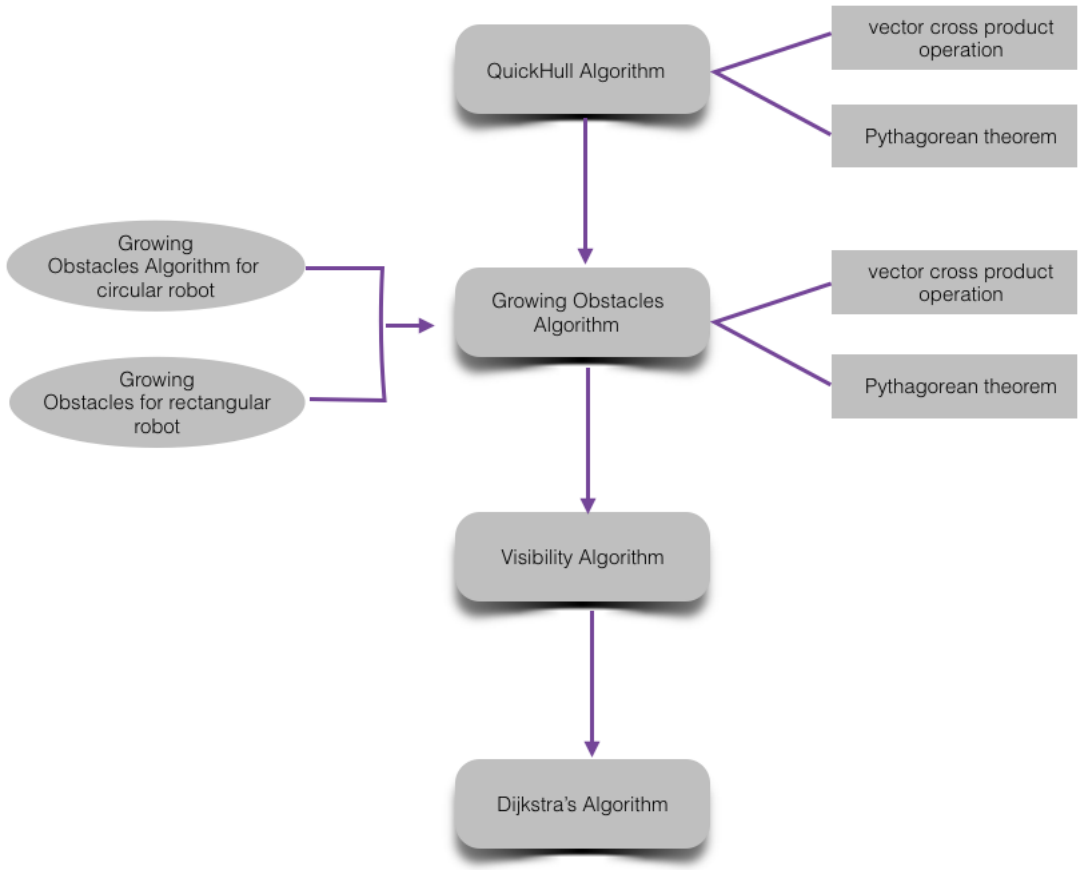


Figure 4.6: Algorithm Design

4.4 GUI Design

4.4.1 User Interface Analysis

The user interface is the only way that the system can directly interact with users. Generally, the goal is to produce a user interface which makes it easy (self explanatory), efficient, and enjoyable (user friendly) to operate a machine in the way which produces the desired result(Wikipedia, 2015).In the Chapter 3, we have gathered all the requirements that the user would like to achieve. Hence, here we are going to design a user interface which could be regarded as a medium to illustrate any specified requirements. The point that we need

to be concerned about is that the user may not have any technical background knowledge about the robot motion planning problem, so we should avoid any complex components in the user interface.

A perfect user interface should be user-friendly, efficiently and consistently, therefore, the GUI(Graphical User Interface) should satisfy the following criteria :

- All action buttons and all panels should be rendered proportionately in order to help users to distinguish them easily. For example, the user should easily be able to identify which area is used for the canvas. Otherwise, it may confuse the user when they are trying to create the environment for the robot.
- We should name the button correctly. We need to carefully name each button, because each button's name is used to directly tell the user which operation they performed. If the button's name is ambiguous, it may mislead the user and cause them to perform a wrong operation. For example: if we name a button as "start" , it could have two different meanings. On the one hand, the user may think the button is used to start the program. On the other hand, the user may believe the button is used to set the start point for the robot. Hence, if we want to set the button as start program button, we could name it as "Run" . On the other hand, if we want to use this button to set the start point for the robot, we could name this button as "set start position" .
- As the user may not have professional knowledge in the field of robot motion , we need to guide them to use this program. Therefore, we should add a description area which to suggest to the user what they could do as the next step. At the same time, when the user performed some wrong actions, we also could alert them by displaying an error message in that area.

4.4.2 User Interface Design

According to the requirements and previous user interface analysis, we will split the user interface into four different Panels:

1. **Operation panel:**
   The Operation panel will consist of 9 buttons which are
   - "Point Robot", the user would be able to use this button to set the robot as a point.
   - "Circle Robot",the user would be able to use this button to set the robot as a circle.
   - "Rectangle Robot", the user would be able to use the button to set the robot as a rectangle.
   - "Start Point", this allows the user to set the robot's start position at anywhere on the canvas.

- "End Point", this allows the user to set the robot's destination at anywhere on the canvas.
- "Draw Obstacles", this allows the user to draw the obstacles by using points.
- "Visibility Graph", the user could abstain a visibility graph of the environment by clicking this button.
- "Shortest Path", the user could identify the shortest path of the robot by clicking this button.
- "Clear ALL", the canvas will be cleaned up if the user clicks this button.

These buttons are all connected to the basic operations that the user may want to operate. Users could build the environments for the robot by using these buttons. Additionally, the user could acquire the shortest path of the robot in that environment by clicking these buttons one by one. Each function relating to the button is one of the steps for the system to calculate the optimal path for the robot. Hence, dividing the task into small steps is useful for the user to understand the process undergone in the robot motion problem and check the system correctness.

2. **Set robot property panel :**
This panel will contain 6 buttons which are "Radius+", "Radius-", "Length+","Length-","Width+" and "Width-". These button will decide the property of the robot. "Radius+" and "Radius-" could change the radius of the circular robot and "Length+/-" and "Width+/-" could be used to modify the length and width of the rectangular robot. These buttons are used for the user to test the system's correctness. For example: in some situations, the circular robot with radius 10cm may pass across successfully. However, if we change the robot's radius to 30cm, the robot may need to walk another way instead of crossing the previous position. In this case, after the robot's radius has been changed, the optimal path for the robot will need to be recalculated. It is an important feature that the system could be able to update the optimal path as the environment has changed.

3. **Canvas:**
There are two main responsibilities for the Canvas: one is to allow the user to build the environment for the robot. Another one is allow the application to display the result after the user clicked some buttons.

4. **Description panel:**
In case of the user not having have any professional background knowledge about robot motion, we decided to add a description panel. This panel has three main roles: we could show some text on the description panel to guide the user as to what they could do as the next step. At the same time, the panel will explain a particular operation if the user has clicked on any buttons. Finally, it will explain all the information of the environment that is displayed in the canvas.

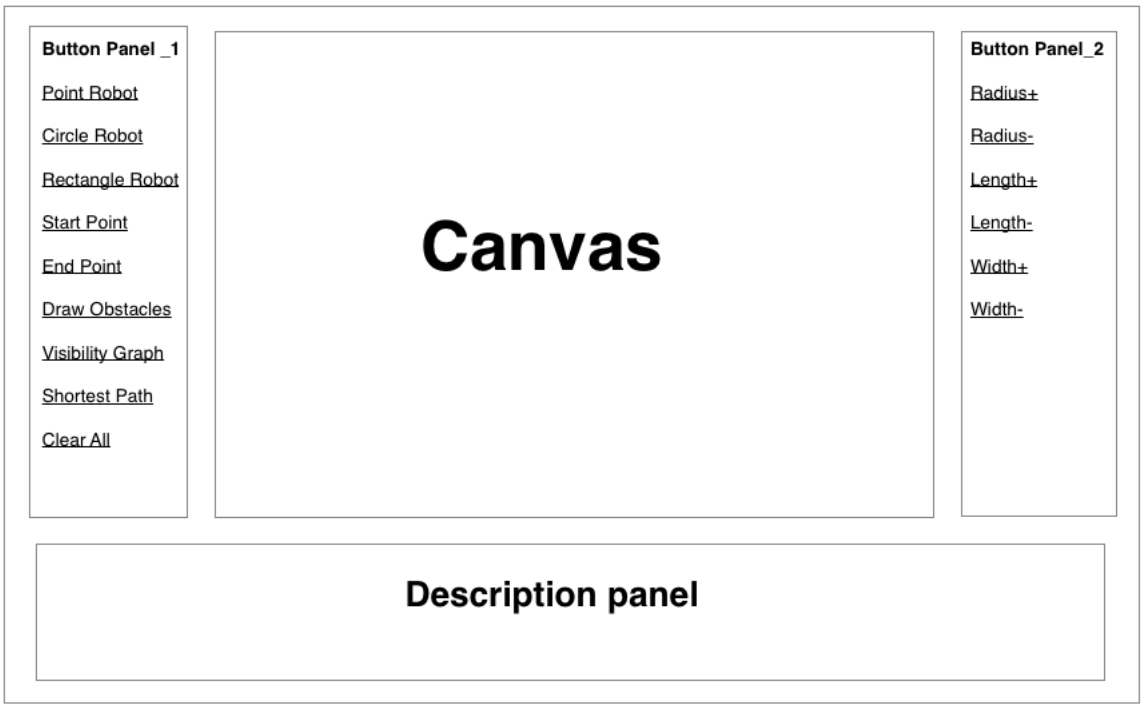


Figure 4.7: User Interface Design Prototyping

Chapter 5

Implementation

5.1 Instruction

In previous chapters, we have discussed the requirements, system and algorithm structure design and the user interface design. In this implementation chapter our focus will be on how to create our application. The first stage of the implementation is to build the user interface. According to the requirements chapter and user interface design section, we could design the user interface step by step. When we were building the user interface, it should satisfy all the requirements which are related to the user interface, such as requirements ID:0.2, ID:0.2.1 and ID:0.3. After the user interface has been built, we will implement the next stage: we create functions which are related to the user interface to allow the user to build the robot environment. In this stage, we will use the Convex Hull algorithm as the main algorithm to draw the obstacles in the environment panel. The final stage of the implementation is finding an optimal path for the robot. This last stage is the main stage in our application, and will include three sub-steps: growing obstacles, showing visibility graph and calculating the shortest path. Here, we will explain how to implement each stage in detail.

5.2 Select Programming Language

There are many different program languages that can be used to build this application. Here we decided to choose Java as our programming language. Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented. (Wikipedia, 2015) The java application could be compiled for once and run on all platforms. This feature of Java will give the user many benefits of using our application. We choose Java not only because of this benefit but also because Java language has an abundant built-in application programming interface (API) which could help us to build a beautiful and functional user interface.

5.3 User Interface Implementation

There are two different ways for us to build the user interface by using java language. One is using well-known graphical software "NetBeans" to build the interface. The NetBeans software is helpful for a programmer to build a user interface quickly. This is because it could automatically generate codes after the programmer uses it to build the user interface. However, the code which is generated by NetBeans is hard to understand and the Code Structure may be in a mess. Therefore, we decided to design the user interface by using Eclipse, which is an integrated development environment (IDE) for Java. By using Eclipse, we could control the code structure to ensure the applications code is effective, clear and readable.

In the 4.4 GUI design section, we have explained that the user interface will include four different panels: Operation panel, Set robot property panel, Canvas Panel and Description panel. Now, we are going describe how to implement them in detail.

5.3.1 Operation Panel and Set robot property Panel

The Operation panel and Set robot property panel are both used to contain buttons, hence, they have a similar operation. In this application, clicking a button triggers most operations. Hence, we need to detect the state of these buttons by adding action listener to these buttons. After the action listener is added, the button click event will be immediately detected by the "actionPerformed" function while user click the button to do an operation. The "actionPerformed" function (Listing 5.2) is used to detect the state of the button. It will call correspond function once a button is clicked by user. As every panel contains many buttons, we decide to create a function (Listing 5.1) to add a button to the panel, in order to improve the code's readability and avoid duplicate code.

Listing 5.1: Add button to panel

```
private void addButton(Container panel, String button){
        JButton btn = new JButton(button);
        btn.setPreferredSize(new Dimension(100, 30));
        btn.addActionListener(this);
        btn.setFocusable(false);
        panel.add(btn);
}
```

Listing 5.2: Detect Button Event

```
public void actionPerformed(ActionEvent e) {
        String choice = e.getActionCommand();
        if(choice == "Start Point"){
                setStartPoint();
        }else if(choice == "End Point"){
                setEndPoint();
        }
```

```
        ... ...
}
```

5.3.2 Canvas Panel

The Canvas Panel is used to build the environment for the robot. At the beginning, the Canvas Panel is an empty white panel without anything. Afterwards, the user could build a new environment by using the mouse to click on the Canvas. In our application, there are three stages we need to implement in the Canvas Panel:

- **Build environment:**
  As we discussed before, the user should be allowed to build the environment by clicking the mouse on the Canvas. In order to achieve this requirement, we have to add a listener to the mouse. By doing this, the system will be able to detect the mouse click action all the time. There are two different tasks for the mouse click action: set the start and end positions for the robot and set the vertices of obstacles. In order to manipulate these different actions, we need combine the button's function (Listing 5.2:Detect Button Event) and the mouse click action function(Listing 5.3). For example, if we want to set the start point, the first step is pressing the "Start Point", after then, once you click on the canvas, the point on canvas will be the start point. However, if you do not press any button and directly click on the canvas, the point will default set as obstacles' vertex.

  Listing 5.3: Build environment

```
public DrawCanvas(){
        this.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
        if(setStart == true){
                startPoint.x = e.getPoint().x;
                startPoint.y = e.getPoint().y;
                setStart = false;
        }else if(setEnd == true){
                endPoint.x = e.getPoint().x;
                endPoint.y = e.getPoint().y;
                setEnd = false;
        }else {
                pointClicked.add(e.getPoint());
        }
                repaint();
        }
        });
}
```

- **Store environment's information:**
  After we built the environment on the canvas, we will store all of this information in

our database. This is because the system will use these data to find the optimal path for the robot. For this environment, we need to store two different kinds of information about the environment: one is the information about the robot includeing the start and end position for the robot and the robot's shape; the other one is the information about obstacles including the shape of the obstacles and the position of the obstacles. We have basically discussed how to store the information about robot's start and end position in the previous build environment section (Listing 5.3 Build Environment). As the Listing 5.3 shows, we use the startPoint.x = e.getPoint().x to store the x value of start position and startPoint.y = e.getPoint().y to store the y value of end position.

The method that used to store the obstacles' information is different from storing the robot's information. We cannot directly acquire the obstacle's information from the canvas. At the beginning, we only identify the information of set of points which could be used to draw the obstacles - pointClicked.add(e.getPoint()) (Listing 5.3). After this, we need to use the Quick Hull algorithm to create the convex hull according to these points. Next, after the obstacles are created, all the detailed information of the obstacles could be abtained, such as: vertex position, edge's position and length.

Listing 5.4: Store obstacle's information

```
private void drawShape(){
        polygonVertex = QuickHull(pointClicked);
        polygons.add(creatPolygon(polygonVertex,polygonLines,polygonNodes));
        ... ...
}
```

- **Show result on the canvas:**
  The last stage is to show the result on the canvas. While the user gives an instruction to the system, the system should immediately execute the instruction and give a result to the user, This result should be shown on the canvas. In this last stage, we use the paintComponent() function (Listing 5.5) to draw the result on the canvas panel. The operation that the user did will decide what result will be shown on the canvas.

Listing 5.5: Show Result

```
public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.fillRect();
        ... ...
}
```

5.3.3 Description Panel

The description panel is used to guide the user to use our application. Due to large amount of information we would like to show in the description panel, we decide to use the JScrollPanel as the description panel. The Description panel consists of two parts. The

first part is overview contexts that describe each button's function. These contexts will be shown automatically in the Description Panel while the application open. The second part's context is decided by the action the user took. These contexts are used to guide the user as to what they could do as the next step. It will update once the user has performed a new operation.

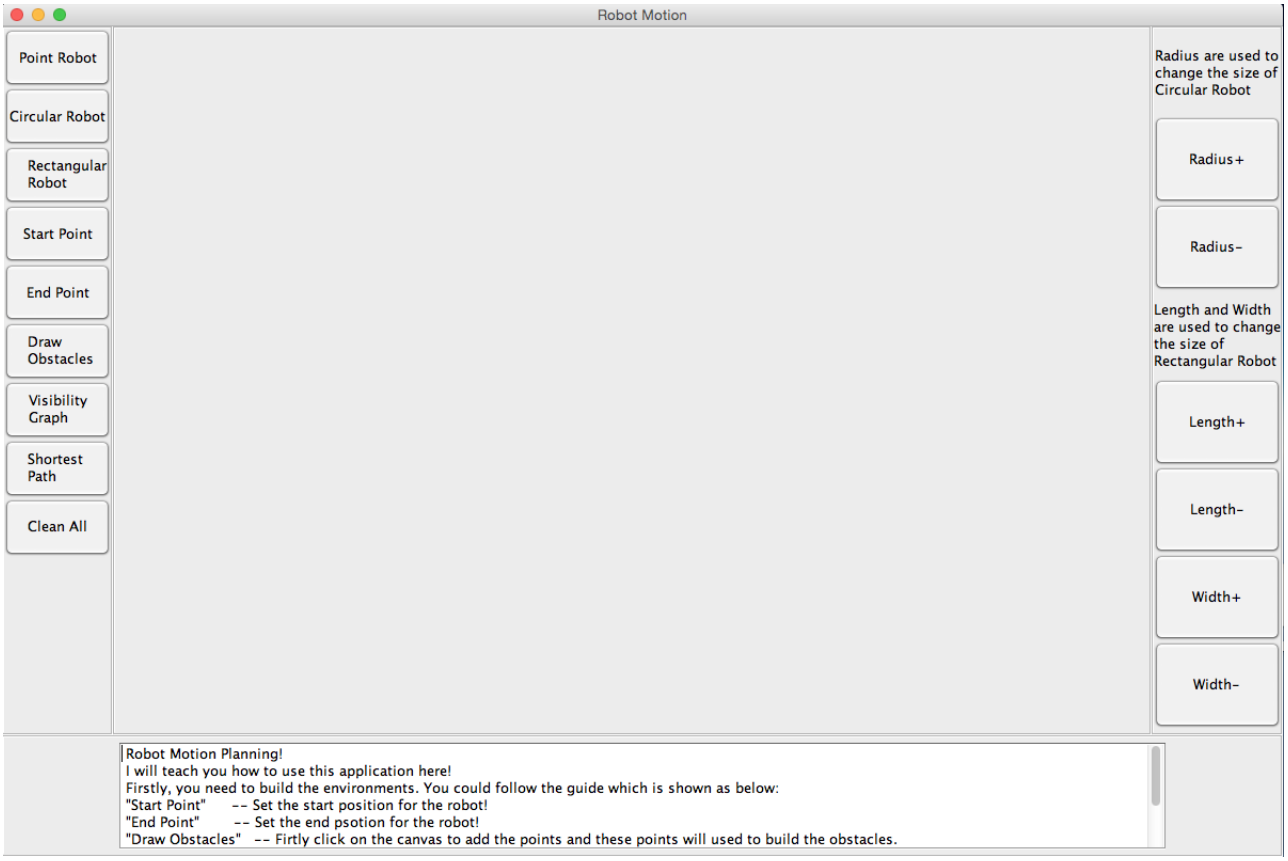


Figure 5.1: User Interface

5.4 Algorithm Implementation

5.4.1 Convex Hull Algorithm

As we have discussed before, we selected the Quick Hull algorithm as the Convex Hull Algorithm to create the obstacles in the environments. In the 5.3.2 Canvas Panel, we have discussed how to store the information of the set of points from the canvas, such as the Listing 5.3 and Listing 5.4. Here we are going to discuss in detail how to build the obstacles by using QuickHull algorithm. In other word, we will now talk about the

QuickHull() function which is used in polygonVertex = QuickHull(pointClicked).

QuickHull Algorithm
Input: set of points
Output: coordinate of vertices of the polygon

In order to have a better implementation of Quickhull algorithm, we divided the Quickhull algorithm into two steps:

- **Step 1:**
  First of all, we need to find the left-most point P_l and the right-most point P_r from the set of points. After then, we connect these two point to form a line $\overline{P_lP_r}$ and then use it to divide the set of points into two different sets. The points placed on the left side of the line $\overline{P_lP_r}$ will be allocated to the left set and the points on the right side of the line $\overline{P_lP_r}$ will be allocated to the right set. We will use the vector cross product method to decide the position of the point related to the line $\overline{P_lP_r}$. The vector cross product is defined by the formula:

  $$a \times b = \|a\|\|b\|sin\Theta n$$

  The $a \times b$ represent a vector that is perpendicular to both vector a and vector b. The $\|a\|\|b\|$ are the magnitudes of vectors a and b. The Θ is the angle between the vector a and b. The n is a unit vector which perpendicular to the plane, in this application, n is always equal to 1. After we found the P_l and P_r, we could use them to test the rest of points which named as P_t. Next, we could use P_t, P_l and P_r to create two vectors: vector a = P_rP_t and vector b = P_rP_l. Now, the vector a and b can be taken into the formula to calculate sinΘ. If the sinΘ is positive, this means that the point is on the left side of the line $\overline{P_lP_r}$. If the sinΘ is Negative, this means that the point is on the right side of the line $\overline{P_lP_r}$. However,as $\|a\|\|b\|$ is always positive, the sign of $a \times b$ will be stay the same as the sign of $sin\Theta$. Therefore, we could simply our code by just test the sign of $a \times b$. The code is shown as follow(return 1 means left set, return 0 means right set):

  Listing 5.6: Cross product method

```
private int pointLocation(Point A, Point B, Point P) {
        int pointPosition = (B.x–A.x)*(P.y–A.y) –
            (B.y–A.y)*(P.x–A.x);
        if(pointPosition > 0){
                return 1;
        }else{
                return –1;
        }
}
```

- **Step 2:**
  After we categorised the left set and the right set, we need to use the Listing 5.7 to find two farthest points which relate to the line $\overline{P_l P_r}$ in both right and left set. Two triangles can be formed by connecting the two farthest points with P_r and P_l . All the points in the triangles could be ignored now. After then,we need to use the Listing 5.7. to iteratively find the farthest point which related to the edges of the triangle until all the points are included. The points we found include P_r and P_l are the vertices of the polygon.

Listing 5.7: Find the farthest points

```
private void findFarthestP(Point A, Point B, ArrayList<Point>
    setPoints, ArrayList<Point> Points) {
            ArrayList<Point> leftSet = new ArrayList<Point>();
            ArrayList<Point> rightSet = new ArrayList<Point>();
     ... ...
}
```

5.4.2 Growing Obstacles

Before drawing the visibility graph, we need to grow the obstacles according to the shape of the robot. In our project, apart from the point robot, we have another two different situations of the robot: rectangular robot without rotation and the circular robot with rotation. Before we start to grow the obstacles, we firstly need to decide which point of the robot is the reference point. In this system, we decided to use the central point of the robot as a reference point. After then, we slide and rotate the robot along the edge of the obstacles to produce the new obstacles. To be more specifically, the growing obstacle also means to grow the vertex of each polygon and after then use the growing vertex to draw the new obstacles. Therefore, The growing method for circular robot is different from the rectangular robot. This is because the circular robot have an offsetting of the polygon but the rectangular robot does not have. Such as the Figure 2.7 shows in Chapter 2.

5.4.3 Visibility Graph

Visibility Graph is consisted of all the possible safe path for the robot from its start to its destination position. The method used to draw the visibility graph is not complicated. Firstly, we collected all the nodes of the polygon into a set includes the start and end point. After then, we connected each pair of points to form a line and then to test whether they intersected with any polygons(the line formed inside of polygon will be counted as intersection as well). If an intersection has been found, the system will disregard that line. Otherwise, the lines will be stored in visibility graph line set. Finally, the visibility graph will be formed based on the set. In the process of producing visibility graph, we should pay more attention in the following points.

1. **Filter the line lies inside of polygon:**
The testing method we have used is to check whether the vertices of a line lie in one polygon. If they are, then we would assume this line is formed inside of polygon and will be disregarded afterwards. At the beginning, we planned to apply a built-in function polygon.contain(point) to test if a line lies in the same polygon or not. However, this function is not appropriate to use in our project. The reason is that some of the vertices will be accounted as contained in the polygon while some are not. Test built-in function is shown in listing 5.8 shows:

Listing 5.8: Find the farthest points

```
private boolean testContain(Point A, Point B, ArrayList<Point>
    setPoints, ArrayList<Point> Points) {
        Polygon p = new Polygon(new int[]{300, 300, 500, 500}, new
            int[]{200, 400, 400, 200}, 4);
        System.out.println(p.contains(new Point(300,400)));
        System.out.println(p.contains(new Point(300,200)));
}
```

The first output is false and the the second output is true. This is because the definition for the "contain" in the java is different from the real case.

> Definition of insideness (java documentation, 2015):
> A point is considered to lie inside a Shape if and only if:
> it lies completely inside theShape boundary or
> it lies exactly on the Shape boundary and the space immediately adjacent to the point in the increasing X direction is entirely inside the boundary or
> it lies exactly on a horizontal boundary segment and the space immediately adjacent to the point in the increasing Y direction is inside the boundary.

Therefore we changed to use another method to test if any points are contained in the polygon. We firstly put all the vertices of the polygon into a set and after then we test if there exist a point in the set have the same coordinate with the test point. If such point in the set exists, we will account the test point as the point contained in the polygon.

2. **Combine Obstacles:**
As we have discussed in the section 2.6.2, in order to increase the efficiency of the Dijkstras Algorithm, we could decrease the number of edge of the visibility graph. Therefore, before we calculating the visibility graph, we could test if the environment exists two or more polygons were intersected with each other, if such polygons exist, we could collect all the nodes of these polygons and use the quickhull() Algorithm to create a new bigger convex hull. The new convex hull will not be shown on the canvas , it will only used to calculate the visibility graph.

5.4.4 Dijkstra's Algorithm

For the Dijkstra's Algorithm, We have discussed it comprehensively in the literature review chapter. We now could program the Dijkstra's Algorithm by apply the pseudo codes which were written in the section2.6.2. First of all, we need to construct all the properties of the node. After the properties have been defineded, we set the initial distance from the start point to each nodes as infinite which shown in listing 5.9. After then,we test and set all the node's properties. The properties include which point is reachable to the current point, what is the distance between them and so on. Finally, after all of the nodes' properties have been set, the Dijkstra's search loop can be applied to find the shortest path for the robot.

Listing 5.9: Initiallize the distance for each nodes

```
        double[][] graph = new double[nodes.size()][nodes.size()];
    for(int x=0; x<nodes.size(); x++){
    for(int y=0; y<nodes.size(); y++){
        graph[x][y] = Double.MAX_VALUE;
    }
}
```

Chapter 6

Testing and Results

After we finished the application, we need to test our application's correctness and functionality. We needed to test whether the application could achieve our expectations and whether it could satisfy all the requirements which we discussed in Chapter 3. We will test the application in 2 different stages. Firstly, we will use the requirements test table to help us to test the application. The test table in section 6.1 could describe the test and record our test result. It consists of five different parts: ID, Goal, Steps, Requirements and Pass. "ID" is the ID for our test, for example, T02 means the second test in our testing. "Goal" is the goal that we want to achieve. The "Test Steps" column is used to describe the steps which we used to achieve our goal. The "Requirement" column corresponds to the functional requirements described in chapter 3. The "Pass" column is used to show the result of our testing: "P" means that the test achieved the goal. "N" means that the test failed to achieve the goal and we need to modify the system until we achieve the goal. After we have done the requirements test, in section 6.2, we will test a special case that is not included in the requirements.

6.1 Requirements Test

ID	Goal	Test Steps	Requirement	Pass
T01	set the start position	press "Start Point" button to set the start position→ click on the canvas	ID:01	P
T02	set the end position	press "End Point" button to set the end position→ click on the canvas	ID:01	P

T03	reset the start or end position	After the position have been set, press the "End Point" or "Start Point"buttons → click on the canvas to reset the position	ID:01.1	P
T03	draw obstacles	use mouse click on the canvas to set the obstacles' vertices → click the "Draw Obstacles" button to draw obstacles	ID:02, ID:02.1	P
T04	show visibility	build environment → press "Visibility Graph" button	ID:0.3	P
T05	make the visibility graph invisible	if the visibility graph have been shown on the canvas → press"Visibility Graph" again to make it invisible	ID: 03.1	P
T06	clean the canvas	press "Clean All" button to clean canvas	ID:04	P
T07	show shortest path	after the environment have been created → press "Shortest Path" to show the optimal path	ID:05	P
T08	set the robot as a circle	press "Circular Robot" to set the robot as circle	ID:06	P
T09	change the circular robot's size	press "Circular Robot" to set the robot as circlr → press "Radius+" and "Radius-" buttons to modify the radius of circular robot	ID:06.1	P
T10	set the robot as a rectangle	press "Rectangular Robot" to set the robot as rectangle	ID:07	P
T11	change the size of rectangular robot	press "Rectangular Robot" to set the robot as rectangle → press the "Length+", "Length-","Width+" and "Width-" to change the size of rectangular robot	ID:07.1	P
T12	show the forbidden space	set environment → press "Visibility Graph"→the forbidden space will be shown as grey space on the canvas.	ID:08	P

6.2 Additional Test

In our application, there are a lot of special cases which are not included in the requirements test. We will test all the special case in detail in order to ensure our system could be used in any environments. In the following testing, we will firstly describe what the special case is, and after that, we will use some screenshots to show the result of testing.If the application work failed in a special environment, we modified the application until it could meet our expectations.

6.2.1 Additional Test 01

Goal:
The visibility graph and the shortest path will automatically be updated when the robot's shape is changed.
Test Steps:
set the environment → click the "Visibility Graph" and "Shortest Path" buttons → click the "Circular Robot" button → click "Rectangle Robot"button → click "Point Robot" button (the application set the robot default as a point)
Result:
It successfully achieved the goal. We test 2 different situations for this goal which are in different environments. The Figure 6.1 shows the result of the basic environment, even the shape of the robot has changed, and they still have a similar optimal path. In the Figure 6.2, it should shows the different optimal path. The optimal path for the robot is substantially different if the robot's shaped is changed. It will cannot pass though the two obstacles, it have to move around the obstacle.

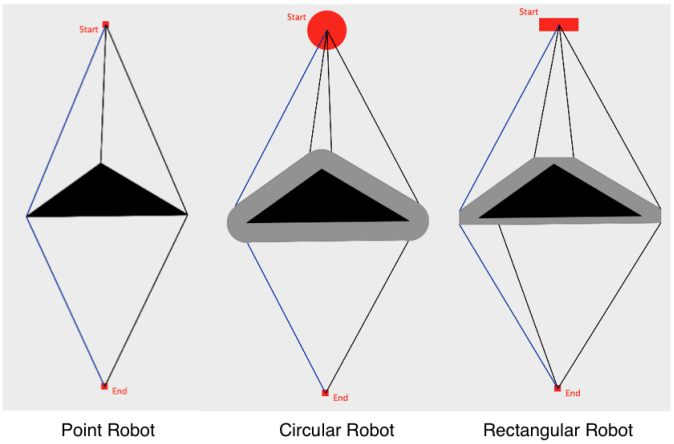


Figure 6.1: Additional Test 01

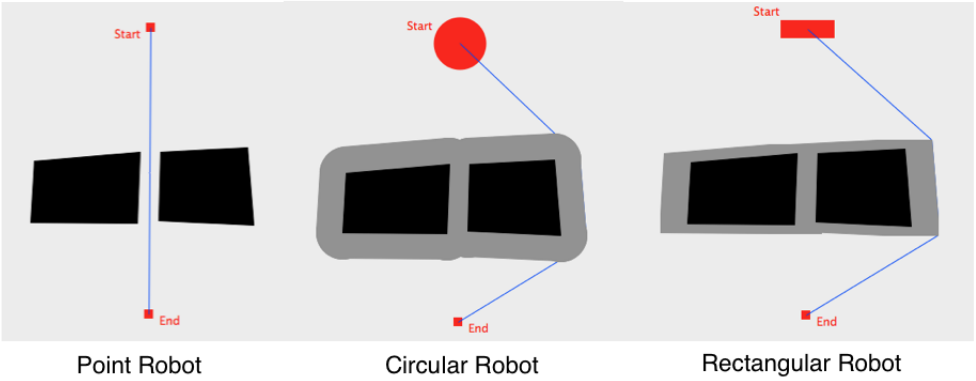


Figure 6.2: Additional Test 01

6.2.2 Additional Test 02

Goal:
The visibility graph and the shortest path will be automatically updated when the size of robot is changed.
Test Steps:
set the environment → click the "Visibility Graph" button and "Shortest Path" buttons → click the "Circular Robot" button → click the "Radius+" and "Radius-" buttons to change the size of robot → click the "Rectangular Robot" button → click the "Length+","Length-"," Width+" and "Width-" buttons to change the size of rectangular robot
Result:
It successfully achieved the goal. As shown in Figure 6.3, in the same environment, the shortest path for the robot is updated when the size of robot is changed.

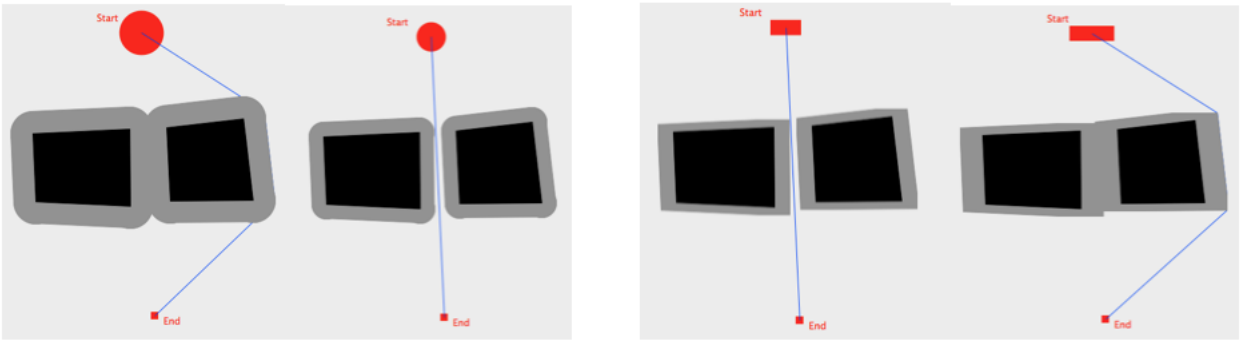


Figure 6.3: Additional Test 02 - In order to make the shortest path clearly shown on the screen,we set the visibility graph invisible here

6.2.3 Additional Test 03

Goal:
The visibility graph and the shortest path will be automatically updated when the environment is changed.
Test Steps:
set the environment → click the "Visibility Graph" button and "Shortest Path" buttons to show the visibility graph and optimal path →click the "Visibility Graph" and "Shortest Path" buttons to make the visibility graph and optimal path invisible → add obstacles in the environment(update the environment) → click the "Visibility Graph" and "Shortest Path" buttons to show the updated visibility graph and the optimal path.
Result:
It successfully achieved the goal. As we can see from the Figure 6.4, the optimal path is changed while we add a one more obstacle in the environment.

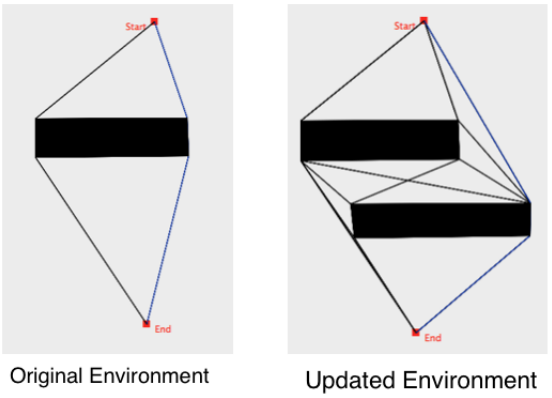


Figure 6.4: Additional Test 03

6.2.4 Additional Test 04

Goal:
The system still could find the shortest path while the user create two obstacles that are intersect with each other.
Test Steps:
create an obstacles in the environment → create an another obstacles in the environment which intersect with previous obstacle →click the "Visibility Graph" and "Shortest Path" buttons to show the visibility graph and optimal path
Result:
It successfully achieved the goal. The optimal path is correct even there exist two polygons intersected with other.

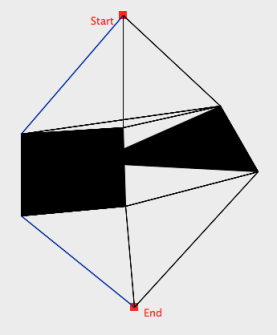


Figure 6.5: Additional Test 04

6.2.5 Additional Test 05

Goal: The system will alert user if the robot cannot move from its start position to its destination(no optimal path).
Test Steps:
create an closed environment for the start point →set the end position at outside of the closed environment → click the "Visibility Graph" and "Shortest Path" buttons to show the visibility graph and optimal path
Result:
It successfully achieved the goal. If the robot cannot move from its start point to its goal, the system will tell the user using the description panel .

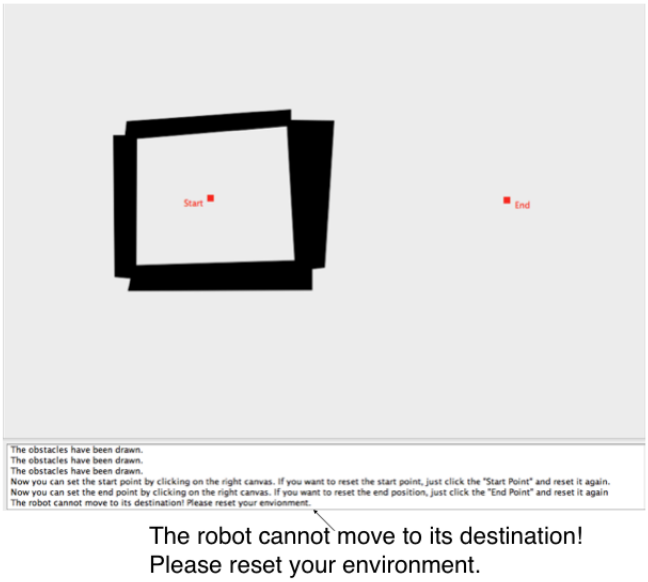


Figure 6.6: Additional Test 05

Chapter 7

Conclusions

7.1 Project Evaluation

In this project, we developed an application which can be used to navigate a robot from its start position to its destination among obstacles without any collision. Currently the application was developed successfully, and it was able to effectively find the optimal path for 3 different shapes of robots in any environment.

In the literature review chapter, we provided an overview of the research which has been done. The literature is helpful during the development process, as it provides all of the algorithms needed in our application. The literature review helped us to understand the project in detail and also gave us an idea of the structure of the application. .

In terms of the requirements of the application, the requirements chapter clearly defined the goals that we want to achieve in the final work. During the development stage, we mainly focused on the functional requirements, and all of these were achieved finally. However, our application did not achieve all the non-functional requirements. The application was not satisfactory in terms of its user-friendliness, because there are a lot of buttons placed on the interface, and it is not simple enough to make the user understand how to use the application. Another reason is that the user-interface is not attractive enough, and the structure of our user interface is not sufficiently clear. We should put all the buttons on one area or combine some buttons to make it clear. For instance, we could combine the "Radius+" button and "Radius-" button to one scroll bar. By doing this, the interface may look clearer and easier to understand.

In the design chapter, we discussed the structure of the program and the user interface. By creating a blueprint of a user interface, less time was spent on this when we moved to the real design. However, there are still some parts that could be improved. As we have discussed before, it would be better to reduce the number of buttons in the user interface. Apart from this point, the design made on overall system structure saved us a lot of time. The application followed the contents shown in the sequence diagrams(Figure

4.4 and Figure 5.5) step by step. The design phase provided a general guide for us and provided a better understanding what should we do at the next stage.

During the program implementation, there were some unexpected difficulties, for example, the problem about the polygon.contain() function which we have discussed in the implementation chapter. And also while we were drawing the visibility graph in the environment, we noticed that the algorithm we used is not efficient enough. We simply connected all the nodes and tested the lines if they intersected with polygons (line inside the polygon is also accounted as intersect with polygon). However, I believe there should exist a higher efficiency algorithm that we could use.

In conclusion, the application still have some areas could be improved. But it has already successfully met most of the requirements and our objectives, therefore, generally speaking, the system has been developed successfully.

7.2 Further Work

After I finished the project, I still believe that there are several areas that merit further improvement in the future. Firstly, regarding with the user interface of this application, we cannot drag the obstacles in the canvas. It may not convenient for user to build their environment. Secondly, in this project, we only dealt with three different kinds of robot: the point robot, rectangular robot and circular robot with rotation. However, there exist more complex situation in terms of the robot size, such as a rectangular robot with rotation. Thirdly, in our application, we always assume the obstacles are a convex hull. However, in the real world, there are lots of concave obstacles. Therefore, we could improve our application by making it usable in a more greater variety of situations.

7.3 Personal Evaluation

I am very glad that I selected this as my final year project. The first time I became aware of robot motion planning was when I saw an article about the Google driverless car. That article stimulated my interest in learning more about the topic of robot motion and its development. That is why I challenged myself in this project to solve complex problems: a circular robot with rotation. After I have done the project, I have more comprehensive understanding about robot motion planning, and I also gained a lot of experience of how to research a project by myself, such as time management, how to find the resources and how to ask help from other people. I believe the accumulated experience will help me continue to learn about the subject in the future.

Bibliography

Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, M. u. W. (2010), *Best Practice Software Engineering*, Spektrum Akademischer Verlag (Springer).

Eranki, R. (2002), 'A* search algorithm'. http://web.mit.edu/eranki/www/tutorials/search/.

GITTA (2015), 'Dijkstras search algorithm'. http://www.gitta.info/website/en/html/unit_about.html.

Howie Choset, Kevin M.Lynch, S. H. G. K. W. B. L. E. and Thrun, S. (n.d.), *Principles of Robot Motion*, The MIT Press.

java documentation (2015), 'polygon.contain()'.

Joshua Fried, E. D. and Pa, W. (n.d.), 'Motion planning in robotics'. http://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/robotics/basicmotion.html.

Lozano-Perez, T. and Center, M. A. W. I. T. J. W. R. (n.d.), An algorithm for planning collision-free paths among polyhedral obstacles, Technical report.

of Glasgow, U. (n.d.), 'Convexhull'. http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf.

Panagiotis, S. (n.d.), 'A* algorithm'. https://www.youtube.com/watch?v=KNXfSOx4eEE.

Patel, A. (2015), 'A* search algorithm'. http://www.redblobgames.com/pathfinding/a-star/introduction.html.

Sunday, D. (n.d.), 'Cthe convex hull of a planar point set'. http://geomalgorithms.com/a10-_hull-1.html.

USTC (n.d.), 'D* algorithm'. http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap25.htm.

Wikipedia (2015). http://en.wikipedia.org/wiki/Wiki.

Appendix A

Code

The following pages are the main codes for this application.

A.1 File: main.java

```
public class main {
  public static void main(String[] args){
    UserInterface I = new UserInterface();
  }
}
```

A.2 File: UserInterface.java

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.geom.Line2D;
import java.util.ArrayList;
import javax.swing.*;
import javax.swing.border.EtchedBorder;

public class UserInterface implements ActionListener{

  private DrawPolygon DP;
  private VisibilityGraph VG;
  private FindShortestPath SP;

  private JFrame frame;
  private Container contentPanel;
  private JPanel buttonPanel;
  private DrawCanvas robotPanel;
  private JPanel setInput;
  private JPanel textPanel;

  private ArrayList<Point> pointClicked;
  private ArrayList<Point> cgrowpointClicked;
  private ArrayList<Point> rgrowpointClicked;

  private Boolean setStart;
  private Boolean setEnd;
  private Point startPoint;
  private Point endPoint;
```

```
private ArrayList<Polygon> polygons;
private ArrayList<Polygon> cgrowpolygons;
private ArrayList<Polygon> rgrowpolygons;

private ArrayList<Point> polygonNodes;
private ArrayList<Point> cgrowNodes;
private ArrayList<Point> rgrowNodes;

private ArrayList<Line2D> polygonLines;
private ArrayList<Line2D> cgrowLines;
private ArrayList<Line2D> rgrowLines;

private ArrayList<Line2D> visibilityLines;
private ArrayList<Line2D> cgrowvisibilityLines;
private ArrayList<Line2D> rgrowvisibilityLines;

private boolean setVisibility;
private boolean showShortestPath;
private ArrayList<Line2D> shortestPath;
private ArrayList<Line2D> cgrowshortestPath;
private ArrayList<Line2D> rgrowshortestPath;
JTextArea textArea;

private boolean circleRobot = false;
private boolean pointRobot = true;
//the radius of circle robot
private int circleRobotR = 30;

private boolean rectangleRobot = false;
private int recRobotw = 20;
private int recRobotl = 60;

private String content;
private boolean haveSolution = true;

public UserInterface(){
  frame = new JFrame("Robot Motion");
  contentPanel = frame.getContentPane();
  buttonPanel = new JPanel();
  setInput = new JPanel();
  robotPanel = new DrawCanvas();
  textPanel = new JPanel();

  pointClicked = new ArrayList<Point>();
```

```
    cgrowpointClicked = new ArrayList<Point>();
    rgrowpointClicked = new ArrayList<Point>();

    setStart = false;
    setEnd = false;
    startPoint = new Point(-1,-1);
    endPoint = new Point(-1,-1);
    setVisibility = false;
    showShortestPath = false;

    DP = new DrawPolygon();
    VG = new VisibilityGraph();
    SP = new FindShortestPath();

    polygons = new ArrayList<Polygon>();
    cgrowpolygons = new ArrayList<Polygon>();
    rgrowpolygons = new ArrayList<Polygon>();

    polygonNodes = new ArrayList<Point>();
    cgrowNodes = new ArrayList<Point>();
    rgrowNodes = new ArrayList<Point>();

    polygonLines = new ArrayList<Line2D>();
    cgrowLines = new ArrayList<Line2D>();
    rgrowLines = new ArrayList<Line2D>();

    visibilityLines = new ArrayList<Line2D>();
    cgrowvisibilityLines = new  ArrayList<Line2D>();
    rgrowvisibilityLines = new  ArrayList<Line2D>();

    shortestPath = new ArrayList<Line2D>();
    cgrowshortestPath = new ArrayList<Line2D>();
    rgrowshortestPath = new ArrayList<Line2D>();

    MakeFrame();
}

private void MakeFrame(){
    // the main panel
    contentPanel.setLayout(new BorderLayout());

    //build buttonPanel
    buttonPanel.setLayout(new GridLayout(12,1));
    addButton(buttonPanel,"Point␣Robot");
    addButton(buttonPanel,"<html>Rectangular<br␣/>Robot</html>");
```

```
addButton(buttonPanel,"Circular Robot");
addButton(buttonPanel,"Start Point");
addButton(buttonPanel,"End Point");
addButton(buttonPanel,"<html>Draw<br/>Obstacles</html>");
addButton(buttonPanel,"<html>Visibility<br/>Graph</html>");
addButton(buttonPanel,"<html>Shortest Path</html>");
addButton(buttonPanel,"Clean All");

buttonPanel.setBorder(new EtchedBorder());
contentPanel.add(buttonPanel, BorderLayout.WEST);

robotPanel.setBorder(new EtchedBorder());
contentPanel.add(robotPanel,BorderLayout.CENTER);

setInput.setLayout(new GridLayout(8,1));
JLabel readMe1 = new JLabel("<html>Radius are used to<br>change the size of<br>Circular Robot</html>",
    SwingConstants.CENTER);
setInput.add(readMe1);
addButton(setInput,"Radius+");
addButton(setInput,"Radius-");

JLabel readMe2 = new JLabel("<html>Length and Width<br>are used to change<br>the size of<br>Rectangular
    Robot</html>", SwingConstants.CENTER);
setInput.add(readMe2);
addButton(setInput,"Length+");
addButton(setInput,"Length-");
addButton(setInput,"Width+");
addButton(setInput,"Width-");
setInput.setBorder(new EtchedBorder());


contentPanel.add(setInput,BorderLayout.EAST);

textPanel.setBorder(new EtchedBorder());

//explain how to use
content = " Robot Motion Planning! \n I will teach you how to use this application here! \n Firstly, you need to
    build the environments. " +
    "You could follow the guide which is shown as below:" +
    "\n \"Start Point\"      -- Set the start position for the robot!" +
    "\n \"End Point\"        -- Set the end psotion for the robot!"+
    "\n \"Draw Obstacles\"   -- Firtly click on the canvas to add the points and these points will used to
        build the obstacles.\n" +
    "                        -- After then, click the \"Draw Obstacles\" button to draw the obstacles." +
    "\n \"Visibility Graph\" -- Show the visibility graph of the environment"+
```

```
        "\n \"Shortest Path\" ---- Show the shortest path. The blue line is the shortest path." +
        "\n \"Clean All\" --------- Clean the canvas.";
    textArea = new JTextArea(content);
    JScrollPane stext = new JScrollPane(textArea);
    stext.setPreferredSize(new Dimension(980, 100));
    textPanel.add(stext,BorderLayout.WEST);
    contentPanel.add(textPanel,BorderLayout.SOUTH);

    frame.setSize(1200,800);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

}

private void addButton(Container panel, String button){
    JButton btn = new JButton(button);
        btn.setPreferredSize(new Dimension(100, 30));
    btn.addActionListener(this);
    btn.setFocusable(false);
    panel.add(btn);
}

public void actionPerformed(ActionEvent e) {
    String choice = e.getActionCommand();
    //draw convex hull
    if(choice == "<html>Draw<br />Obstacles</html>"){
      if(pointClicked.size() > 2){
        textArea.append("\n The obstacles have been drawn.");
        textArea.setCaretPosition(textArea.getDocument().getLength());
        drawShape();
      }else{
        textArea.append("\n Please set at least 3 vertice of the obstacles on the right canvas. ");
        textArea.setCaretPosition(textArea.getDocument().getLength());
      }
    }else if(choice == "Start Point"){
      textArea.append("\n Now you can set the start point by clicking on the right canvas. If you want to " +
          "reset the start point, just click the \"Start Point\" and reset it again.");
      textArea.setCaretPosition(textArea.getDocument().getLength());
      setStart = true;
      setEnd = false;
    }else if(choice == "End Point"){
      textArea.append("\n Now you can set the end point by clicking on the right canvas. If you want to " +
          "reset the end position, just click the \"End Point\" and reset it again");
      textArea.setCaretPosition(textArea.getDocument().getLength());
      setEnd = true;
```

```
  setStart = false;
}else if(choice == "<html>Shortest Path</html>"){
  if(startPoint.x != -1 && endPoint.x != -1){
    if(showShortestPath)showShortestPath = false;
    else{
        showShortestPath = true;
        textArea.append("\n The blue line is the shortest path.");
        textArea.setCaretPosition(textArea.getDocument().getLength());
      }
    showVG();
  }else{
    textArea.append("\n You haven't set the start point or end point yet. Please set them on the environment
        panel firstly.");
    textArea.setCaretPosition(textArea.getDocument().getLength());
  }
}else if(choice == "Clean All"){
  textArea.append("\n The canvas have been clean up.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  cleanAll();
}else if(choice == "<html>Visibility<br />Graph</html>"){
  if(startPoint.x != -1 && endPoint.x != -1){
    showVG();
    //show visibility graph
    if(!setVisibility){
      setVisibility = true;
      textArea.append("\n Visibility Graph is consist of all the black lines includes the edges of obstacles.");
      textArea.setCaretPosition(textArea.getDocument().getLength());
    }else if(setVisibility) setVisibility = false;
  }else{
    textArea.append("\n You haven't set the start point or end point yet. Please set them on the environment
        panel firstly.");
    textArea.setCaretPosition(textArea.getDocument().getLength());
  }
}else if(choice == "Point Robot"){
  textArea.append("\n Now, the robot is a point.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  rectangleRobot = false;
  circleRobot = false;
  pointRobot = true;
}else if(choice == "Circular Robot"){
  textArea.append("\n Now, the robot's shape is a point.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  circleRobot = true;
  pointRobot = false;
  rectangleRobot = false;
```

```
}else if(choice == "<html>Rectangular<br/>Robot</html>"){
  textArea.append("\n Now, the robot's shape is a rectangle.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  rectangleRobot = true;
  circleRobot = false;
  pointRobot = false;
}else if(choice == "Radius+" && circleRobot){
  textArea.append("\n The radius of the robot has been increased by 5cm.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  circleRobotR = circleRobotR + 5;
  reDrawShape();
  showVG();

}else if(choice == "Radius-" && circleRobot){
  textArea.append("\n The radius of the robot has been decreased by 5cm.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  circleRobotR = circleRobotR - 5;
  reDrawShape();
  showVG();
}else if(choice == "Length+" && rectangleRobot){
  textArea.append("\n The length of the robot has been increased by 5cm.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  recRobotl = recRobotl + 5;
  reDrawShape();
  showVG();
}else if(choice == "Length-" && rectangleRobot){
  textArea.append("\n The length of the robot has been decreased by 5cm.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  recRobotl = recRobotl- 5;
  reDrawShape();
  showVG();
}else if(choice == "Width+" && rectangleRobot){
  textArea.append("\n The width of the robot has been increased by 5cm.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  recRobotw = recRobotw + 5;
  reDrawShape();
  showVG();
}else if(choice == "Width-" && rectangleRobot){
  textArea.append("\n The width of the robot has been decreased by 5cm.");
  textArea.setCaretPosition(textArea.getDocument().getLength());
  recRobotw = recRobotw - 5;
  reDrawShape();
  showVG();
}
```

```
    robotPanel.repaint();
}

// used for user change the size of robot
private void reDrawShape(){

    ArrayList<Polygon> newPolygons = new ArrayList<Polygon>();
    newPolygons.addAll(polygons);
    polygons.clear();
    cgrowpolygons.clear();
    rgrowpolygons.clear();

    polygonNodes.clear();
    cgrowNodes.clear();
    rgrowNodes.clear();

    polygonLines.clear();
    cgrowLines.clear();
    rgrowLines.clear();

    for(Polygon p : newPolygons){
      // get all the point of each polygon
      int[] x = p.xpoints;
      int[] y = p.ypoints;
      for(int i = 0; i < x.length; i++){
        pointClicked.add(new Point(x[i],y[i]));
      }

      //for point circle
      pointClicked = DP.QuickHull(pointClicked);
      polygons.add(creatPolygon(pointClicked,polygonLines,polygonNodes));

      //for circle robot - grow the nodes
      cgrowpointClicked = DP.cgrowPolygon(pointClicked,circleRobotR);

      //get growing polygon for circle robot
      cgrowpolygons.add(creatPolygon(cgrowpointClicked,cgrowLines,cgrowNodes));


      //get grow nodes for rectangle robot
      rgrowpointClicked =
          DP.rgrowPolygon(pointClicked,recRobotw,recRobotl,creatPolygon(pointClicked,polygonLines,polygonNodes));
      rgrowpointClicked = DP.QuickHull(rgrowpointClicked);
      //get growing polygon for rectangle robot
      rgrowpolygons.add(creatPolygon(rgrowpointClicked,rgrowLines,rgrowNodes));
```

```
        pointClicked.clear();
        cgrowpointClicked.clear();
        rgrowpointClicked.clear();

      }
    }

    private void drawShape(){

      //for point circle
      pointClicked = DP.QuickHull(pointClicked);
      polygons.add(creatPolygon(pointClicked,polygonLines,polygonNodes));

      //for circle robot - grow the nodes
      cgrowpointClicked = DP.cgrowPolygon(pointClicked,circleRobotR);

      //get growing polygon for circle robot
      cgrowpolygons.add(creatPolygon(cgrowpointClicked,cgrowLines,cgrowNodes));


      //get grow nodes for rectangle robot
      rgrowpointClicked =
          DP.rgrowPolygon(pointClicked,recRobotw,recRobotl,creatPolygon(pointClicked,polygonLines,polygonNodes));
      rgrowpointClicked = DP.QuickHull(rgrowpointClicked);
      //get growing polygon for rectangle robot
      rgrowpolygons.add(creatPolygon(rgrowpointClicked,rgrowLines,rgrowNodes));

      pointClicked.clear();
      cgrowpointClicked.clear();
      rgrowpointClicked.clear();
    }


    private Polygon creatPolygon(ArrayList<Point> points, ArrayList<Line2D> lines,ArrayList<Point> nodes){
      int xPoly[] = new int[30];
      int yPoly[] = new int[30];
      for (int i = 0; i < points.size(); i++){
        //used to draw polygons
        xPoly[i] = points.get(i).x;
        yPoly[i] = points.get(i).y;
        //used to draw visibility graph
        nodes.add(points.get(i));
//        collect the edges of each polygons
        if(i == points.size()-1){
```

```
            Line2D line = new Line2D.Double();
            line.setLine(points.get(i),points.get(0));
            lines.add(line);
          }else{
            Line2D line = new Line2D.Double();
            line.setLine(points.get(i),points.get(i+1));
            lines.add(line);
          }
        }

        return new Polygon(xPoly,yPoly,points.size());
      }


      private void cleanAll(){
        setStart = false;
        setEnd = false;
        haveSolution = true;
        startPoint.x = -1;
        startPoint.y = -1;
        endPoint.y = -1;
        endPoint.x = -1;
        pointClicked.clear();
        polygons.clear();
        visibilityLines.clear();
        polygonNodes.clear();
        polygonLines.clear();
        shortestPath.clear();

        cgrowpointClicked.clear();
        cgrowpolygons.clear();
        cgrowvisibilityLines.clear();
        cgrowNodes.clear();
        cgrowLines.clear();
        cgrowshortestPath.clear();

        rgrowpointClicked.clear();
        rgrowpolygons.clear();
        rgrowvisibilityLines.clear();
        rgrowNodes.clear();
        rgrowLines.clear();
        rgrowshortestPath.clear();

//      robotPanel.repaint();
```

```
  }

  private void showVG(){

//     if(startPoint.x == -1 ){
//       JOptionPane.showMessageDialog(frame, "Please set the Start point.");
//       setVisibility = false;
//     }else if(endPoint.x == -1){
//       JOptionPane.showMessageDialog(frame, "Please set the End point.");
//       setVisibility = false;
//     }

    // for point robot
    polygonNodes.add(0, startPoint);
        polygonNodes.add(endPoint);

        cgrowNodes.add(0, startPoint);
      cgrowNodes.add(endPoint);

      rgrowNodes.add(0, startPoint);
      rgrowNodes.add(endPoint);

      // collision detection
      visibilityLines = VG.createVisibilityGraph(polygonNodes,polygonLines,polygons);
        cgrowvisibilityLines = VG.createVisibilityGraph(cgrowNodes,cgrowLines,cgrowpolygons);
        rgrowvisibilityLines = VG.createVisibilityGraph(rgrowNodes,rgrowLines,rgrowpolygons);

        //find the shortestPath
        try{
          shortestPath = SP.DijkstraAlgorithm(visibilityLines,startPoint,endPoint);
          cgrowshortestPath = SP.DijkstraAlgorithm(cgrowvisibilityLines, startPoint, endPoint);
            rgrowshortestPath = SP.DijkstraAlgorithm(rgrowvisibilityLines, startPoint, endPoint);
        }catch (Exception e){
          haveSolution = false;
          textArea.append("\n The robot cannot move to its destination! Please reset your envionment.");
      textArea.setCaretPosition(textArea.getDocument().getLength());
        }
  }

  class DrawCanvas extends JPanel{
    private static final long serialVersionUID = 1L;

    public DrawCanvas(){
      this.addMouseListener(new MouseAdapter() {
                public void mouseClicked(MouseEvent e) {
```

```
            if(setStart == true){
              startPoint.x = e.getPoint().x;
              startPoint.y = e.getPoint().y;
              setStart = false;
            }else if(setEnd == true){
              endPoint.x = e.getPoint().x;
              endPoint.y = e.getPoint().y;
              setEnd = false;
            }else {
              pointClicked.add(e.getPoint());
            }

              repaint();
          }
        });
}

@Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        //draw start point
        if(startPoint.x >= 0 || startPoint.y >= 0 ){
          String message_start = "Start";
          g.setColor(Color.RED);

          if(circleRobot){
            g.drawOval(startPoint.x-circleRobotR, startPoint.y-circleRobotR, circleRobotR*2, circleRobotR*2);
            g.drawString(message_start,startPoint.x-60,startPoint.y-15);
          }else if(rectangleRobot){
            g.drawRect(startPoint.x-recRobotl/2,startPoint.y-recRobotw/2,recRobotl,recRobotw);
            g.drawString(message_start,startPoint.x-60,startPoint.y-15);
          }else{
            g.drawString(message_start,startPoint.x-40,startPoint.y+12);
          }
            g.fillRect(startPoint.x-5,startPoint.y-5,10,10);


        }

        //draw end point
        if(endPoint.x >= 0 || endPoint.y >= 0){
          String message_end = "End";
          g.setColor(Color.RED);
          g.drawString(message_end, endPoint.x+12, endPoint.y+12);
```

```
  g.fillRect(endPoint.x-5, endPoint.y-5, 10, 10);
}

//store the clicked point to draw a polygon
if(pointClicked.size() != 0){
  g.setColor(Color.BLACK);
  for (int i = 0; i < pointClicked.size(); i++){
    g.fillRect(pointClicked.get(i).x-2, pointClicked.get(i).y-2, 4, 4);
  }
}

//draw visibility lines
if(setVisibility){
  if(haveSolution){
    if(pointRobot){
      for (Line2D l : visibilityLines){
        g.setColor(Color.BLACK);
        g.drawLine((int)l.getX1(),(int)l.getY1(),(int)l.getX2(),  (int)l.getY2());
      }
    }else if(circleRobot){
      for (Line2D l : cgrowvisibilityLines){
        g.setColor(Color.BLACK);
        g.drawLine((int)l.getX1(),(int)l.getY1(),(int)l.getX2(),  (int)l.getY2());
      }
    }else if(rectangleRobot){
      for (Line2D l : rgrowvisibilityLines){
        g.setColor(Color.BLACK);
        g.drawLine((int)l.getX1(),(int)l.getY1(),(int)l.getX2(),  (int)l.getY2());
      }
    }
  }
}

//show shortest point
if(pointRobot){
  if(showShortestPath && shortestPath.size() != 0){
    for (Line2D l : shortestPath){
      g.setColor(Color.BLUE);
      g.drawLine((int)l.getX1(),(int)l.getY1(),(int)l.getX2(),  (int)l.getY2());
    }
  }
}else if(circleRobot){
  if(showShortestPath && cgrowshortestPath.size() != 0){
    for (Line2D l : cgrowshortestPath){
      g.setColor(Color.BLUE);
```

```
      g.drawLine((int)l.getX1(),(int)l.getY1(),(int)l.getX2(),  (int)l.getY2());
    }
  }
}else if(rectangleRobot){
  if(showShortestPath && rgrowshortestPath.size() != 0){
    for (Line2D l : rgrowshortestPath){
      g.setColor(Color.BLUE);
      g.drawLine((int)l.getX1(),(int)l.getY1(),(int)l.getX2(),  (int)l.getY2());
    }
  }
}
if(circleRobot){
  if(cgrowpolygons.size() != 0){
    for (int i = 0; i < polygons.size(); i++){
      g.setColor(Color.GRAY);
      g.drawPolygon(cgrowpolygons.get(i));
      g.fillPolygon(cgrowpolygons.get(i));
    }

    for (Line2D l : polygonLines){
      g.drawOval((int)l.getX1() - circleRobotR, (int)l.getY1() -
          circleRobotR,circleRobotR*2,circleRobotR*2);
      g.fillOval((int)l.getX1() - circleRobotR, (int)l.getY1() -
          circleRobotR,circleRobotR*2,circleRobotR*2);
    }

  }
}else if(rectangleRobot){
  if(rgrowpolygons.size() != 0){
    for (int i = 0; i < polygons.size(); i++){
      g.setColor(Color.GRAY);
      g.drawPolygon(rgrowpolygons.get(i));
      g.fillPolygon(rgrowpolygons.get(i));
    }
  }
}

//draw original polygons
if(polygons.size() != 0){
  for (int i = 0; i < polygons.size(); i++){
    g.setColor(Color.BLACK);
    g.drawPolygon(polygons.get(i));
    g.fillPolygon(polygons.get(i));
  }
}
```

```
        }
    }
}
```

A.3 File: VisibilityGraph.java

```
import java.awt.Point;
import java.awt.Polygon;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;
import java.util.ArrayList;

public class VisibilityGraph {

    public ArrayList<Line2D> createVisibilityGraph(ArrayList<Point> polygonNodes, ArrayList<Line2D> polygonLines,
                            ArrayList<Polygon> polygons)
    {
            ArrayList<Line2D> vizLines = new ArrayList<Line2D>();
            for (int i = 0; i < polygonNodes.size(); i++){
              for (int j = 0; j< polygonNodes.size(); j++){
                if(i != j){
                   Line2D.Double tempLine = new Line2D.Double(polygonNodes.get(i), polygonNodes.get(j));
                   //check if this line intersects any of the grown polygon
                   //step 1: if the intersect point is not the vertex of the polygon, then the line intersect the
                       polygon
                     boolean intersects = false;
                     for(Line2D polyLine : polygonLines)
                     {
                             Point intersectP = intersection(tempLine,polyLine);
                             if(!polygonNodes.contains(intersectP) && intersectP != null){
                               intersects = true;
                               break;
                             }
                     }
                     //if this line does NOT intersect any grown polygon, add it to the visibility lines
                     if(!intersects){
                       // step 2: if the line's point1 and point2 is located at polygon vertex, then it is
                           inside the polygon
                         // so set intersect true
                         for(Polygon polygon : polygons)
                         {
                           int[] points_x = polygon.xpoints;
                           int[] points_y = polygon.ypoints;
```

```
                        ArrayList<Point2D> testPoints = new ArrayList<Point2D>();
                        for(int pn = 0; pn < points_x.length;pn++){
                          Point2D testP = new Point(points_x[pn],points_y[pn]);
                          testPoints.add(testP);
                        }

                        if(testPoints.contains(tempLine.getP1()) && testPoints.contains(tempLine.getP2())){
                          intersects = true;
                          for(Line2D l : polygonLines){
                            boolean b1 = l.getP1().equals(tempLine.getP1()) &&
                                l.getP2().equals(tempLine.getP2());
                            boolean b2 = l.getP1().equals(tempLine.getP2()) &&
                                l.getP2().equals(tempLine.getP1());
                            if(b1 || b2){
                              intersects = false;
                                  break;
                            }
                          }
                        }
                        // the points are two different polygons vertex(one polygon's vertex is inside other
                            polygons)
                        if(polygon.contains(tempLine.getP1())){
                          if(!testPoints.contains(tempLine.getP1())){
                            intersects = true;
                          }
                        }else if(polygon.contains(tempLine.getP2())){
                          if(!testPoints.contains(tempLine.getP2())){
                            intersects = true;
                          }
                        }
                      }
                    }
                    if(!intersects){vizLines.add(tempLine);}
                  }
                }
              }
              return vizLines;
  }

  //get the intersect points between two lines
public Point intersection(Line2D l1, Line2D l2) {
  int x1 = (int) l1.getX1();
  int x2 = (int) l1.getX2();
  int y1 = (int) l1.getY1();
  int y2 = (int) l1.getY2();
```

```
    int x3 = (int) l2.getX1();
    int x4 = (int) l2.getX2();
    int y3 = (int) l2.getY1();
    int y4 = (int) l2.getY2();

      int distance = (x1-x2)*(y3-y4) - (y1-y2)*(x3-x4);
      if (distance == 0){
        return null;
      }

      int xi = ((x3-x4)*(x1*y2-y1*x2)-(x1-x2)*(x3*y4-y3*x4))/distance;
      int yi = ((y3-y4)*(x1*y2-y1*x2)-(y1-y2)*(x3*y4-y3*x4))/distance;

      Point p = new Point(xi,yi);
      if (xi < Math.min(x1,x2) || xi > Math.max(x1,x2)) return null;
      if (xi < Math.min(x3,x4) || xi > Math.max(x3,x4)) return null;

      if (yi < Math.min(y1,y2) || yi > Math.max(y1,y2)) return null;
      if (yi < Math.min(y3,y4) || yi > Math.max(y3,y4)) return null;

      return p;
  }
}
```

A.4 File: FindShortestPath.java

```
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;
import java.util.ArrayList;
import java.util.Collections;


public class FindShortestPath {

   public ArrayList<Line2D> DijkstraAlgorithm(ArrayList<Line2D> vizGraph,Point2D start, Point2D goal){
            //create the initial list of graph nodes and where they can reach
            ArrayList<GraphNode> nodes = new ArrayList<GraphNode>();
            for(Line2D l1 : vizGraph){
                    Point2D p1 = l1.getP1();
                    Point2D p2 = l1.getP2();

                    //add the graph node for p1 if it has not already been created
```

```
        createGraphNode(vizGraph, nodes, p1, start, goal);

        //add the graph node for p2 if it has not already been created
        createGraphNode(vizGraph, nodes, p2, start, goal);
}

//create the 2D array of infinity distances
double[][] dijGraph = new double[nodes.size()][nodes.size()];
for(int x=0; x<nodes.size(); x++){
        for(int y=0; y<nodes.size(); y++)
        {
                dijGraph[x][y] = Double.MAX_VALUE;
        }
}

//set up the starting parameters and exit conditions
boolean goalFound = false;
int startIndex = findStartEndNode(nodes, start);
int endIndex = findStartEndNode(nodes, goal);
GraphNode curNode = nodes.get(startIndex);
dijGraph[startIndex][startIndex] = 0;
curNode.distance = 0;

//dijkstra's search loop:
while(!goalFound){
        for(GraphNode gn : nodes){
                //iterate through the list of nodes and check which nodes are reachable by the original
                if(curNode.reachables.contains(gn.point) && !gn.visited){
                        if(curNode.index == gn.index)
                                dijGraph[curNode.index][gn.index] = 0;
                        else{
                                double dist = curNode.point.distance(gn.point) + curNode.distance;
                                if(dist < dijGraph[curNode.index][gn.index]){
                                        dijGraph[curNode.index][gn.index] = dist;
                                        if(dist < gn.distance){
                                                gn.previous = curNode;
                                                gn.distance = dist;
                                        }
                                }
                        }
                }
        }
        //set the current node to visited
        curNode.visited = true;
        if(curNode.isGoal)
```

```
                        goalFound = true;

                //find the next "current node" to iterate through
                curNode = findNextCurNode(nodes);

                if(curNode == null)
                        goalFound = true;
        }

        //now that we've created the 2D array, go backwards from the goal to find the shortest path
        ArrayList<Line2D> shortestPath = new ArrayList<Line2D>();
        GraphNode cur = nodes.get(endIndex);
        boolean pathFound = false;
        while(!pathFound){
                shortestPath.add(new Line2D.Double(cur.point, cur.previous.point));
                cur = cur.previous;
                if(cur.isStart)
                        pathFound = true;
        }
        Collections.reverse(shortestPath);

        return shortestPath;
  }


private void createGraphNode(ArrayList<Line2D> vizGraph,
        ArrayList<GraphNode> nodes, Point2D p1, Point2D start, Point2D goal) {

  //make sure that the list of graph nodes does not already contain p1
  if(!graphNodeContains(nodes, p1)){
        ArrayList<Point2D> reachables = new ArrayList<Point2D>();

        //determine which points are reachable by p1
        for(Line2D l2 : vizGraph){
                //if p1 on line2, then it must reach the other point
                if(p1.equals(l2.getP1()) && !reachables.contains(l2.getP2())){
                  reachables.add(l2.getP2());
                }
                if(p1.equals(l2.getP2()) && !reachables.contains(l2.getP1())){
                      reachables.add(l2.getP1());
                }
        }

        //check if p1 is the start or the goal
        GraphNode gn = new GraphNode(p1, reachables, nodes.size());
```

```
            if(gn.point.equals(start))
                    gn.isStart = true;

            else if(gn.point.equals(goal))
                    gn.isGoal = true;

            //add p1 to the list total list of graph nodes
            nodes.add(gn);
    }
}

    private class GraphNode{
            public Point2D point;
            public ArrayList<Point2D> reachables;
            public GraphNode previous;
            public double distance;
            public int index = 0;
            public boolean visited;
            public boolean isStart;
            public boolean isGoal;

            public boolean equals(GraphNode gn){
                    return this.point.equals(gn.point);
            }

            //constructor
            public GraphNode(Point2D p, ArrayList<Point2D> r, int in){
                    point = p;
                    index = in;
                    previous = null;
                    distance = Double.MAX_VALUE;
                    visited = false;
                    isStart = false;
                    isGoal = false;
                    reachables = r;
            }
    }

    private boolean graphNodeContains(ArrayList<GraphNode> nodes, Point2D p){
            for(GraphNode n : nodes){
                    if(n.equals(p))
                            return true;
            }

            return false;
```

```
    }

    private int findStartEndNode(ArrayList<GraphNode> nodes, Point2D start_goal){
            int index = 0;
            for(GraphNode gn : nodes){
                    if(gn.point.equals(start_goal))
                            return index;

                    index++;
            }

            return -1;
    }

    private GraphNode findNextCurNode(ArrayList<GraphNode> nodes){
            double smallestDist = Double.MAX_VALUE;
            GraphNode next = null;
            for(GraphNode gn : nodes){
                    if(!gn.visited && gn.distance < smallestDist)
                    {
                            next = gn;
                            smallestDist = gn.distance;
                    }
            }

            return next;
    }

}
```

A.5 File: DrawPolygon.java

```
import java.awt.Point;
import java.awt.Polygon;
import java.awt.Rectangle;
import java.util.ArrayList;


public class DrawPolygon{

    public ArrayList<Point> QuickHull (ArrayList<Point> points) {
      ArrayList<Point> convexHull = new ArrayList<Point>();
      int minPoint = -1, maxPoint = -1;
```

```
  int minX = Integer.MAX_VALUE;
  int maxX = Integer.MIN_VALUE;

  if (points.size() <= 2){
    return (ArrayList)points.clone();
  }

  for (int i = 0; i < points.size(); i++) {
    if (points.get(i).x < minX) {
      minX = points.get(i).x;
      minPoint = i;
    }
    if (points.get(i).x > maxX) {
      maxX = points.get(i).x;
      maxPoint = i;
    }
  }

  Point A = points.get(minPoint);
  Point B = points.get(maxPoint);
convexHull.add(A);
  convexHull.add(B);
  points.remove(A);
  points.remove(B);

  ArrayList<Point> leftSet = new ArrayList<Point>();
  ArrayList<Point> rightSet = new ArrayList<Point>();

  for (int i = 0; i < points.size(); i++) {
    Point p = points.get(i);
    if (pointLocation(A,B,p) == -1)
      leftSet.add(p);
    else
      rightSet.add(p);
  }
  hullSet(A,B,rightSet,convexHull);
  hullSet(B,A,leftSet,convexHull);

  return convexHull;
}

private void hullSet(Point A, Point B, ArrayList<Point> set, ArrayList<Point> hull) {
  int insertPosition = hull.indexOf(B);
  if (set.size() == 0){
    return;
```

```
    }
    if (set.size() == 1) {
      Point p = set.get(0);
      set.remove(p);
      hull.add(insertPosition,p);
      return;
    }
    int dist = Integer.MIN_VALUE;
    int furthest_Point = -1;
    for (int i = 0; i < set.size(); i++) {
      Point p = set.get(i);
      int distance  = distance(A,B,p);
      if (distance > dist) {
        dist = distance;
        furthest_Point = i;
      }
    }
    Point P = set.get(furthest_Point);
    set.remove(furthest_Point);
    hull.add(insertPosition,P);

    ArrayList<Point> leftSetAP = new ArrayList<Point>();
    for (int i = 0; i < set.size(); i++) {
      Point M = set.get(i);
      if (pointLocation(A,P,M)==1) {
        leftSetAP.add(M);
      }
    }

    ArrayList<Point> leftSetPB = new ArrayList<Point>();
    for (int i = 0; i < set.size(); i++) {
      Point M = set.get(i);
      if (pointLocation(P,B,M)==1) {
        leftSetPB.add(M);
      }
    }
    hullSet(A,P,leftSetAP,hull);
    hullSet(P,B,leftSetPB,hull);
}

  private int pointLocation(Point A, Point B, Point P) {
      int pointPosition = (B.x-A.x)*(P.y-A.y) - (B.y-A.y)*(P.x-A.x);
      if(pointPosition > 0){
        return 1;
      }else{
```

```
            return -1;
        }
    }


    private int distance(Point A, Point B, Point C) {
        int ABx = B.x-A.x;
        int ABy = B.y-A.y;
        int num = ABx*(A.y-C.y)-ABy*(A.x-C.x);
        if (num < 0){
          num = -num;
        }
        return num;
    }


// get the grow points - circular robot
    public ArrayList<Point> cgrowPolygon(ArrayList<Point> Points, int circleRobotR){
      ArrayList<Point> growPoints = new ArrayList<Point>();
      for(int i = 0; i < Points.size()-1;i++){
        growPoints.add(cgetGrowPoint(Points.get(i),Points.get(i+1),circleRobotR,1));
        growPoints.add(cgetGrowPoint(Points.get(i+1),Points.get(i),circleRobotR,-1));
      }
      growPoints.add(cgetGrowPoint(Points.get(Points.size()-1),Points.get(0),circleRobotR,1));
      growPoints.add(cgetGrowPoint(Points.get(0),Points.get(Points.size()-1),circleRobotR,-1));

      return growPoints;
    }

    private Point cgetGrowPoint(Point p1, Point p2,int r,int testSide){
      Point growP = null;
      Boolean Perpendicular;
      int x1 = p1.x;
      int x2 = p2.x;
      int y1 = p1.y;
      int y2 = p2.y;
      //p1 - p2 distance
      Double distance = Math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
      double sin = Math.abs(x1-x2)/ distance;
      double cos = Math.abs(y1-y2)/ distance;

      Double newX,newY;
      newX = (x1-r*cos);
      newY = (y1+r*sin);
```

```
  Perpendicular = Math.abs((newX - x2)*(newX - x2)+(newY - y2)*(newY - y2) -
      (newX-x1)*(newX-x1) - (newY-y1)*(newY-y1) - (x2-x1)*(x2-x1)-(y2-y1)*(y2-y1)) <= 1;
  // testSide used to test which side the point is, -1 is at right of the line, 1 is left of the line
  //perpendicular is used to test if the the point line is perpendicular of the line
  if(pointLocation(p1,p2,new Point(newX.intValue(),newY.intValue())) == testSide && Perpendicular)
    growP = new Point(newX.intValue(),newY.intValue());

  newX = (x1+r*cos);
  newY = (y1+r*sin);
  Perpendicular = Math.abs((newX - x2)*(newX - x2)+(newY - y2)*(newY - y2) -
      (newX-x1)*(newX-x1) - (newY-y1)*(newY-y1) - (x2-x1)*(x2-x1)-(y2-y1)*(y2-y1)) <= 1;
  if(pointLocation(p1,p2,new Point(newX.intValue(),newY.intValue())) == testSide && Perpendicular)
    growP = new Point(newX.intValue(),newY.intValue());

  newX = (x1+r*cos);
  newY = (y1-r*sin);
  Perpendicular = Math.abs((newX - x2)*(newX - x2)+(newY - y2)*(newY - y2) -
      (newX-x1)*(newX-x1) - (newY-y1)*(newY-y1) - (x2-x1)*(x2-x1)-(y2-y1)*(y2-y1)) <= 1;
  if(pointLocation(p1,p2,new Point(newX.intValue(),newY.intValue())) == testSide && Perpendicular)
    growP = new Point(newX.intValue(),newY.intValue());

  newX = (x1-r*cos);
  newY = (y1-r*sin);
  Perpendicular = Math.abs((newX - x2)*(newX - x2)+(newY - y2)*(newY - y2) -
      (newX-x1)*(newX-x1) - (newY-y1)*(newY-y1) - (x2-x1)*(x2-x1)-(y2-y1)*(y2-y1)) <= 1;
  if(pointLocation(p1,p2,new Point(newX.intValue(),newY.intValue())) == testSide && Perpendicular)
    growP = new Point(newX.intValue(),newY.intValue());

  return growP;
}



// get grow points for rectangular robot
public ArrayList<Point> rgrowPolygon(ArrayList<Point> Nodes, int width, int length, Polygon polygon){
  int l = length;
  int w = width;
  int x,y;
  ArrayList<Point> growPoints = new ArrayList<Point>();

  for (Point n : Nodes){
    x = (int) n.getX();
    y = (int) n.getY();
    if(!polygon.intersects(x-l, y-w, l, w)){
      growPoints.add(new Point((x-l/2),(y-w/2)));
```

```
        }
        if(!polygon.intersects(x-l, y, l, w)){
          growPoints.add(new Point((x-l/2),(y+w/2)));
        }
        if(!polygon.intersects(x, y-w, l, w)){
          growPoints.add(new Point((x+l/2),(y-w/2)));
        }
        if(!polygon.intersects(x, y, l, w)){
          growPoints.add(new Point(x+l/2,(y+w/2)));
        }
      }

      return growPoints;
    }

}
```