Development of a shortest path searching software in 2D

Bing Zhao Tan

Bachelor of Science in Computer Science with Honours

The University of Bath

May 2009
This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:
Abstract

Robot motion planning is a topic that has been discussed for several years. The algorithms behind the motion planning have been widely used in our life. For example, GPS gives a short path for cars. This project presents a software that could be used to find out a shortest path for travelling. In the software, users can model the obstacles between two points. In the project, we have described the background algorithms for achieving the objective in the literature review. The design and implementation of this software are presented in this document, and the further development should be made to the software in the conclusion.
Development of a shortest path searching software in 2D

Submitted by: Bingzhao Tan

COPYRIGHT
Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

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:
# Contents

List of Figures

Chapter 1 - Introduction

1.1 Problem Description ...........................................2  
1.2 Aim and Objectives ...........................................2  
1.3 Dissertation Structure ........................................3

Chapter 2 - Literature Review

2.1 What is Motion Planning? ........................................4  
   2.1.1 Motion-planning Problem Classification .......................5  
   2.1.2 Complexity of Motion-planning .............................6  
   2.1.3 Motion-planning Algorithm Classification .................6  
2.2 What is the Configuration Space? .............................7  
2.3 Polytopes Modeling ..........................................8  
   2.3.1 What is Convex Hull? ..................................9  
   2.3.2 Convex Hull Algorithm ................................10  
2.4 Free Space Searching .......................................11  
2.5 Shortest-path Graph Search Algorithm .........................13  
   2.5.1 Dijkstra’s algorithm ................................13  
   2.5.2 A* algorithm ......................................16  
2.6 Visibility Graph ...........................................18  
2.7 Summary ................................................19

Chapter 3 - Requirements

3.1 Introduction ................................................20  
3.2 Requirements Analysis ......................................20  
3.3 Requirements Specification .................................21  
   3.3.1 Functional Requirement .................................21  
   3.3.2 Non-Functional Requirement ............................24  
3.4 Project Constraints .........................................25  
   3.4.1 Time ...............................................25  
   3.4.2 Software resource ....................................25
Appendix B – Screenshots ........................................56
Appendix C – Testing Table ........................................59
Appendix D – Parts of Source Code ..............................61
List of Figures

Figure 2.1: a point robot among static obstacles ............5
Figure 2.2: the forbidden space and the free space ............7
Figure 2.3: convex subsets .................9
Figure 2.4: example of convex hull ...........10
Figure 2.5: Minkowskie Sum of two triangles ..............12
Figure 2.6: polygon shortest path spanning tree ..........14
Figure 2.7: an example search by A* algorithm .............17
Figure 2.8: visibility graph ................18
Figure 4.1: the repository model of RobotPathSearch ..........27
Figure 4.2: the layered model of algorithm system ..........28
Figure 4.3: the manager model of graph constructor ..........29
Figure 4.4: block diagram ................30
Figure 4.5: event-processing model ................31
Figure 4.6: user interface prototype .................34
Figure 5.1: upper convex hull and lower convex hull ..........42
Figure 5.2: user interface ..................46
Figure 5.3: edit menu ......................46
Figure 5.4: special case for visibility graph ..............47
Figure A.1 object oriented design diagram ..............55
Figure B.1 input obstacles and the robot ..............56
Figure B.2 the forbidden space on canvas ..............57
Figure B.3 the visibility graph on canvas ..............57
Figure B.4 the shortest path and distance ..............58
Acknowledgements

I would like to thank my supervisor, Professor Nicolai Vorobjov for his help throughout the project. I would also like to thank Alan Ip who has given some suggestions for my implementation.
Chapter 1

Introduction

1.1 Problem Description

The movement of objects is a widely discussed topic in the world of technology, especially robotics. Every second, there are millions of different kinds of arbitrary objects moving around in the world, but in most cases we try to look for the shortest path to the destination. In Euclidean space, this is fairly straightforward to calculate; however, an issue arises when there are obstacles between the start point and the destination point: how do we circumnavigate, or manoeuvre around, those obstacles but still follow the next shortest path? What even if the obstacles are of different shapes? We can resolve the above problems by building a computer model of the situation using model-based manipulation systems. Then, we need to consider an algorithm for planning the optimized path that a polyhedral object should follow for moving amongst a set of known obstacles. Collision detection is the first main question of the project; it is a basic tool to achieve efficiency in many robotics and computer graphics application, such as obstacle avoidance, motion planning and physical object dynamics simulation. Hence, we should first disquisition an algorithm for avoiding collisions. Conversely, if we are just catering for scenarios with a single obstacle or few obstacles, each proposed path provides only local collision information and so it would be much more practical to find out the worse case length of the iterated path. The global view is an important side of this project; therefore, this project is based on investigating algorithms of collision avoidance and optimization path. Determining the most efficient collision-free length of paths is an important subject in the computer modeling of physical object motion, such as robot motion planning.

1.2 Aim and Objectives

The main aim of this project now is to develop software for searching the shortest path among
Development of a shortest path searching software in 2D

some convex polygons (obstacles). The software should allow users modeling graphically.

In order to achieve the aim of this project, the objectives have been identified:

- Research and study a selection of motion planning algorithms to find out the pros and cons of each algorithm.
- Describe the motion planning algorithm in two dimensions and implement it.
- Improve the algorithms to three dimensions
- Develop a program which performs an object moving under above algorithms.
- Model the moving object as a point, moving among known polyhedral obstacles using the algorithms and describe how the algorithms is worked in the process.
- Test for remaining problem or possible improvement to the motion planning algorithm.

1.3 Dissertation Structure

The next chapter of the dissertation is the literature review; providing the background into the problem and reviewing the algorithms needed to solve the problem. Chapter 3 identifies the requirements for the project. From these requirements, high level design decisions are made in the chapter 4. Chapter 5 is including the implementation and testing for the project, the implementation is based on the design mentioned in the previous chapter. The last chapter has discussed the conclusion of the project and further development of the software.
Chapter 2

Literature Review

2.1 What is Motion Planning?

A robot is a versatile mechanical device, performing tasks by executing motions in the workspace. This workspace is interacting with physical objects and is subject to the law of nature. Usually, the robot motion is executed in order to achieve a task through some static spatial physical objects automatically. This means we need to program a feasible plan for the robot motion first, such that the need for planning motions will become more significant, so we will consider the problem with motion planning. Thus motion planning computes motion to achieve a goal of a spatial arrangement of physical objects from an initial arrangement; otherwise motion planning is primarily used to collision detection [1]. We classify the Motion planning in the following session. As a conclusion, the basic problem of motion planning is collision-free path planning for one rigid object (robot) among static obstacles which are polytopes. E.g. Figure 2.1

Inputs:
- geometric description of the obstacles and the robot
- kinematics’ and dynamic properties of the robot
- initial and goal positions of the robot

Outputs:
- continuous sequence of collision-free paths from the initial position to goal positions
2.1.1 Motion-planning Problem Classification

Motion planning (defined by Yong K.Hwang and Narendra Ahyja[2]) can be classified as either gross-motion planning or fine-motion planning. Gross-motion planning is concerned with the problems involving free space wider than that the objects’ sizes plus the positional error of the robot. Fine-motion planning is concerned with the problem of moving objects when the space is so narrow that the required accuracy of motion exceeds a robot’s positional accuracy. Path planning and trajectory planning are included in motion planning. Path planning typically consists of only a kinematical geometric specification of the positions and directions of the robot, but trajectory planning is also concerned with the linear and angular velocities. We always devise path first and then assign the velocity, but sometimes the trajectory planning is unimportant. Motion planning can be static or dynamic which is depending on the obstacle information. In a static problem, all the information about the obstacles is known at the beginning, and the motion of the robot is devised from the known information, whereas in the dynamic problem, only partial information about the obstacles is available, and so the robot plans a path based on the partial information to achieve a given goal. By following the originally designed path, the robot discovers more information about obstacles, and then it needs to redesign a path from the updated information until it reaches the goal. If the obstacles are stationary, it is a time invariant problem; otherwise it is a time variant problem. Furthermore, there are conformable problems and unconformable problems depending on whether the robot can change its shape or not. Motion planning is either constrained or unconstrained depending on whether there are constraints on the motion of the robot.
2.1.2 Complexity of Motion-planning

We use big $O$ notation to analyze algorithms in computer science and describe complexity of problems. E.g. $n^2 + 3n + 4$ is $O(n^2)$. Complexity of a problem is analyzed by giving an upper bound on the number of elementary computations or a lower bound on the size of memory space required. We denote $O(u(n))$ as an upper bound is gained by showing an algorithm for the problem having a complexity bounded above by some function of the input size. We denote $O(1(n))$ as a lower bound is gained by showing a group of example problems whose only solutions are bounded below by some functions of the size of the example. If a problem has an upper bound is matching a lower bound, i.e. $u(n) = l(n)$, then we can say that the problem has complexity $u(n)$.

The complexity of a motion planning problem depends on the degrees of freedom of a given robot, denoted $d$, and is concerned with the number of obstacles in an environment (workspace), denoted $n$. Degrees of freedom is the number of parameters specifying the configuration of an object. The configuration of an object of a particular shape is a set of independent parameters that configure the position of the object. The obstacles are polytopes, which are polygons in two dimensions (2D) and polyhedra in three dimensions (3D). All of these that we need to consider in a configuration space (see section 1.2).

2.1.3 Motion-planning Algorithm Classification

We can separate the motion-planning algorithm in two aspects, one is the completeness algorithm, and the other is the scope algorithm. The completeness algorithm includes exact algorithm and heuristic algorithm. Exact algorithms either find a solution or prove that no solution exists, and they are expensive for computation. Heuristic algorithms have got a lower solving time to generate a solution, but the disadvantages of them are possible failure to find a solution for difficult problems, or to find a poor solution. The scope algorithms contain global algorithms and local algorithms. Global algorithms take into account all the information in the environment and plan a solution for motion from the start position to the goal position. Local algorithms provide a solution for motion by the local information in the environment; they will possibly change the solution when they collect further information. The lack of the global view makes the local algorithms may find out a worse solution and search expensively, but they are useful for the further unknown environments which are not present in the model.
between start and goal configuration. Also the local algorithm can be used a component of global algorithm.

2.2 What is the Configuration Space?

In section 1.1.2, we have roughly introduced the degree of freedom (DOF) of a robot. In general, the number of DOF corresponds to the number of parameters specifying the placement of the robot. In two dimensions, this number is three for planning robots that can translate and rotate; it is six in three dimensions. Thus the configuration space is basically the parameter space of the robots and obstacles, which corresponds to workspace. The configuration space of a translating robot on a plane is the 2 dimensional Euclidean space, but a translating and rotating robot needs to be represented in the configuration space like $\mathbb{R}^2 \times [0:360^\circ]$. Then we can specify a placement of the robot with a reference point in the configuration space, but not all parts of the configuration space are possible. We denote the forbidden space as the part of the configuration space consisting of the points corresponding to placement where the robot intersects obstacles in workspace. In other words, the forbidden space is the space where the robot is not allowed to translate through. The rest of the configuration space is called free space, which consists of the points corresponding to placements where the robot does not intersect any obstacle, e.g. figure 2.2. Therefore we can model the environment on computer by using a configuration space and to plan the robot motions, thus the collision-free paths are curves in the free space.

![Figure 2.2: the grey area is the forbidden space, and the space between the grey areas is the free space.](image-url)
2.3 Polytopes Modeling

Before planning the robot motion, we need to model the workspace on computer using a model-based manipulation system, to best approximate how to model the robot and the obstacles which are polytopes. There are two alternatives that are usually suggested for geometric modeling, which are a boundary representation and a solid representation. Boundary representation is used to define an equation of a polytope coinciding with the polytope’s surface. Solid representation is describing the set of all points that are contained in the polytope or not. Both of them are useful in our motion planning in the computer model.

Firstly, we define the workspace $W$ for two possible circumstances: 1. 2D workspace, in which $W = \mathbb{R}^{2}$, and 2. 3D workspace, in which $W = \mathbb{R}^{3}$. There two elements in the workspace $c$, what are obstacles and robots. We denote the obstacle region $O$ as the set of all points that lie within an obstacle in $W$, hence $O \subseteq W$. Also the robots will be defined in this similar way.

The solid representation of $O$ is developed in terms of a combination of primitives. In a computer, each primitive represents a subset of $W$, which is easy to manipulate. A complicated obstacle region can be represented in finite and Boolean combinations of primitives. For convex polygons in 2D workspace, $W = \mathbb{R}^{2}$, we define it as for any pair of points $x_{1}, x_{2} \in X$, $X$ a subset called convex, and $\lambda \in [0,1]$, then there exists

$$\lambda \cdot x_{1} + (1-\lambda) \cdot x_{2} \in X$$

A boundary representation of $O$ is an $m$-sided polygon that can be described using vertices and edges, which are referred to as the corners and line segments of the polygon respectively. As a result, a sequence $(x_{1}, y_{1}), (x_{2}, y_{2}), \ldots, (x_{m}, y_{m})$ of $m$ points in $\mathbb{R}^{2}$ can be used to specify the polygon in a counterclockwise order. An edge of the polygon are specified by two points (vertices), using the function describing the line passes through the points $(x_{i}, y_{i}), (x_{j}, y_{j})$. We can determine an equation of the form $a \cdot x + b \cdot y + c = 0$ from $x_{i}, y_{i}, x_{j}, y_{j}$, where $a, b, c \in \mathbb{R}$. Then we will have the function $f(x, y) = a \cdot x + b \cdot y + c$ for the line corresponding to the edge.

In detail, a solid representation of $O$ can be stated as the intersection $m$ half-planes. Each half-plane corresponds to the set of points lying to one side of a polygon edge. From above, we can assume that $f(x, y) < 0$ on one side of the edge, and $f(x, y) > 0$ on the other side, otherwise $f(x, y) = 0$, which is on the edge. Hence we define a half-plane $P_{i}$ as:
Development of a shortest path searching software in 2D

\[ P_i = \{(x, y) \in W \mid f_i(x, y) \leq 0\} \]

where \( f_i(x, y) \) is the line corresponding to the edge from \((x_i, y_i), (x_{i+1}, y_{i+1})\).

Thus, the \( n \)-sided convex polygonal obstacle region \( O \) can be presented as:

\[ O = H_1 \cap H_2 \cap \cdots \cap H_m \]

In general, the polyhedral models are similar to the 2D cases. The 3D model replaces the half-plane primitives with half-space primitives. The 2D is defined by 2 features, whereas 3D case is defined by 3 features which are vertices, edges and faces. (Steven M LaValle[3])

2.3.1 What is Convex Hull

In this project, we just consider the convex polygons mentioned in previous sections. A polygon, which is a subset \( S \) of the plane, is called convex (as Figure 2.3) if and only if for any pair of points in the polygon, \( p, q \in S \). The line segment \( pq \) is completely contained in \( S \). Hence the convex hull \( CH(S) \) of a set \( S \) is the smallest convex set that contains \( S \).

![Convex subset](image)

Assume there is a finite set \( P \) of \( n \) points in the plane, the convex hull of \( P \) is a convex polygon. An easy way to represent a polygon is by listing its vertices in clockwise order, beginning with an arbitrary one. The problem is considering as:

Input = set of points : \( p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8 \)

Output = representation of the convex hull

For further representing the convex hull, we say if all points of \( P \) except \( p, q \) lie to the one side of the straight line through \( p \) and \( q \), then the segment \( pq \) is an edge of \( CH(P) \). An example of a set of points and its convex hull is provided in Figure 2.4.
2.3.2 Convex Hull Algorithm

There have been a number of algorithms developed for finding 2D-convex hulls. E.g Gift Wrapping Algorithm, Quick Hull Algorithm, Graham’s Algorithm, Incremental Algorithm and so on.

Gift Wrapping Algorithm

Gift Wrapping Algorithm is described by Jarvis [4] in 1973. The algorithm begins by locating lowest (right-most) points, and then finds the points with the smallest from this point. So a hull edge must be through these two points, and then use this established hull edge to find another point with the smallest angle in counter-clockwise by setting the second point to be a pivot point. The process is doing a wrapping the line around the set of points until the beginning point is attacked again. The run time of Gift Wrapping Algorithm is $O(nh)$ where $n$ is number of points and $h$ is number of hull edges.

Quick Hull Algorithm

Quick Hull Algorithm is presented by O’Rourke [5]. The algorithm is similar to Gift Wrapping Algorithm. But it defined four extreme points which are highest, right-most, lowest, left-most in two dimensions, such that the four points are connecting to form a quadrilateral. The algorithm is doing selection of the four corners formed by the quadrilateral, e.g. finding an arbitrary point in the top-right corner to construct a new polygon. Then use the Quick Hull function recursively for further computation until all points contained in a polygon. The complexity of this algorithm is $O(n \log n)$ average and the worst case is $O(n^2)$. 

Figure 2.4: Example of a set of points and its convex hull.<3>
Graham’s Algorithm

Graham’s Algorithm is described by Graham [6] in 1972 and it has been updated by O’Rourke [16]. The first step is to identify the lowest (right-most) point, defined a point 0, it is obvious on the hull. The next step is to sort all of the points angularly in an anti-clockwise direction about this point, and then it is pushing the final point found to have the greatest angle on a stack; the follow point is point 0. Beginning from the point with the lowest angle, named point 1, other points are then process in turn until the final point reached. When we walked around a polygon in anti-clockwise, the turn can be both a right turn and a left turn for an arbitrary polygon, but for a convex polygon the turn must be a left turn. The process of this algorithm tests for a strict left turn from the point at top-1 in the stack. The running time of Graham’s Algorithm is \( O(n \log n) \).

Incremental Algorithm

Incremental Algorithm described by Berg, Kreveld, Overmars, and Schwarzkopf [1] is similar to Graham’s Algorithm, but it tests the points (vertices) in clockwise order and separates the hull to upper hull and lower hull by using left-most point and right-most point. The process involves a test for a right turn. For the upper hull, it tests by left-to-right order. Otherwise, it tests by right-to-left order for the lower hull. But the algorithm is not mentioned if there are two points have the same x-coordinate and if there are three points on a straight line make a right turn. This algorithm also has \( O(n \log n) \) time complexity.

2.4 Free Space Searching

From section 1.2, we know that a collision-free path for a robot in the configuration space is a curve in free space, so how can we find the free space? For a point robot with polygon obstacles in the configuration space, we just need to use the trapezoidal map algorithm [1] to compute the free space. But as the robot is a polygon, the problem will be more complicated because of the forbidden space is concerned with the shape of the robot. This problem is considered as the obstacles in configuration space are no longer the same as in workspace. Consequently, we apply the Minkowski Sums (Figure 2.5) to model the “new” obstacles. Lozano and Wesley [7] have represented the grown obstacles by using Minkowski Sums.
The Minkowski sum of two sets $S_1 \subseteq \mathbb{R}^2$ and $S_2 \subseteq \mathbb{R}^2$, denoted $S_1 \oplus S_2$, which means:

$$S_1 \oplus S_2 := \{p + q : p \in S_1, q \in S_2\},$$

Where $p + q$ is the vector sum of the vectors $p$ and $q$, i.e., if $p = (p_x, p_y)$ and $q = (q_x, q_y)$ then we have $p + q := (p_x + q_x, p_y + q_y)$.

In our objectives, because the robot and the obstacles are convex polygons in a plane, the Minkowski sums can be applied.

We assume the robot $T$ is a convex polygon and so is an obstacle $P$. We use $T(x, y)$ to denote the position of the robot with its reference point at $(x, y)$. Then the grown obstacle is $P' = P \oplus (-T(x, y))$, where $-T(x, y)$ is reflection of $T(x, y)$ about the origin in configuration space (2D Euclidean space).

The algorithm for calculating the Minkowski sum can be expressed as follow:

Inputs: A convex polygon $T$ with vertices $v_1, \cdots, v_n$ and a convex polygon $P$ with vertices $w_1, \cdots, w_m$. Assume vertices are in counterclockwise order in the lists.

Outputs: The Minkowski sum $T \oplus P$

1. $i = 1; j = 1$

2. while $i \neq n + 1$ and $j \neq m + 1$

3. Add $v_i + w_j$ as a vertex to $T \oplus P$

4. if $\text{angle}(v_{i+1}v_i) < \text{angle}(w_{j+1}w_j)$
5. \( i = i + 1; \)

6. \( \text{else if } \text{angle}(v_{i,1}) > \text{angle}(w_{j,1}) \)

7. \( j = j + 1; \)

8. \( \text{else } i = i + 1 \text{ and } j = j + 1 \)

Explanation: For each pair \( v \) and \( w \) of vertices of \( T \) and \( P \), compute \( v + w \), and then compute the convex hull (section 2.3.1) of these sums. We denote \( \text{angle}(xy) \) is the angle between the vector \( xy \) and the positive \( x \)-axis. The running time of the algorithm of two convex polygon is \( O(n+m) \).

2.5 Shortest-path Graph Search Algorithm

There are many different graph search algorithms for different shortest path problems. In a graphical model of the obstacles and the robot, we need some algorithm to help us to find out the shortest collision-free paths among the obstacles for the robot. The problem here concerns with the paths between two points which are named start position and goal position, so we investigate shortest path algorithms which will find an optimal path to the problem, such as Dijkstra’s algorithm and A* algorithm.

2.5.1 Dijkstra’s algorithm

Dijkstra’s algorithm was first described by E.W. Dijkstra[8]. It is one of many important graph search algorithms and uses greedy strategy. It solves the single-source shortest path problem, which it looks for the shortest paths from a source vertex to all other vertices in the graph, on a weighted graph with all positive edge weighted path costs, outputting a shortest path tree. A shortest path tree is a subgraph of a given graph constructed so that the distance between a selected root node and all other nodes is minimal. The running time of Dijkstra’s algorithm is lower than that of the Bellman-Ford algorithm (see [9]), but the Dijkstra’s algorithm is restricted by non-negative edges and the Bellman-Ford algorithm solves the single-source shortest path problem with the edges that may be negative. Therefore the Bellman-Ford algorithm is usually used only when there are negative edge weights, whereas Dijkstra’s algorithm is always using in routing, e.g. shortest path searching. Figure 2.6 shows a shortest
In our project objective, all of the edge weights are non-negative, so that we prefer to use Dijkstra’s Algorithm with the lower running time. We construct a directed graph $G = (V, E)$ for problems, where $V$ is a set with elements of nodes or vertices and $E$ is a set of edges. By the description of the Dijkstra’s algorithm above, all of the edges should be non-negative, so we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$, where $w$ is the weight function of a particular edge and $u, v \in V$ are the nodes (vertices). Dijkstra’s algorithm maintains a set $S$ of vertices whose shortest distances from the source have already been determined, and we use a set $Q$ of min-priority queue of vertices. Then we can implement the algorithm by the following pseudo code.
Development of a shortest path searching software in 2D

As shown in the pseudo code, the Dijkstra’s algorithm repeatedly selects the vertex \( u \in V - S \) with the minimum distance estimate, adds \( u \) to \( S \), and relaxes all edges leaving \( u \). Thus finally we get a set \( S \) of vertices which are solved and an array of shortest distances between all vertices and the source.

The running time of Dijkstra’s algorithm on a graph with edges \( E \) and vertices \( V \) depends on how the min-priority queue is implemented. Consider the pseudo code, which we maintain the min-priority queue of the vertices being numbered 1 to \( |V| \) in an ordinary linked list and
operation \( \min(Q) \) is simply a linear search through all vertices in Line 9. The linear search takes \( O(V^2) \) time after comparing all the vertices \( v \) in \( Q \), which \( Q = V \) in the beginning. Since the total number of edges is \( |E| \), there is a total of \( |E| \) repeat of loops from Line 11 to 14, such that the running time of Dijkstra’s algorithm is \( O(|V|^2 + |E|) = O(|V|^2) \) in this instance. Otherwise, if the graph is quite sparse, we can look for the \( \min(Q) \) in Line 9 more efficiently by using binary heap [9], the running time will be reduced to be \( O\left((|E| + |V|) \cdot \log V\right) \) which is assuming all vertices could be connected to source, i.e. \( |E| \geq |V| - 1 \). Also we can achieve Fibonacci heap [9] to implement the min-priority queue, in which case we will have \( O\left(|E| + |V| \cdot \log |V|\right) \).

### 2.5.2 A* algorithm

The A* algorithm is a graph search algorithm that is used to find a least-cost path from a given start node to one goal node. The A* algorithm is popular in using because it estimates the least cost of reaching the goal with heuristic (evaluation) function, such that it usually finds an optimal path. A heuristic approach typically uses special domain knowledge about the represented problem to improve the computation. The cost estimate is made up of two parts; one part is the actual known cost between the start node and the current node, the other part is the estimated cost of reaching the goal node.

Therefore we can construct the heuristic function \( \hat{f}(n) \) as following:

\[
\hat{f}(n) = g(n) + \hat{h}(n)
\]

where \( g(n) \) is the actual cost from start node to current node, \( \hat{h}(n) \) is the estimated cost from current node to goal node. \( \hat{h}(n) \) is significantly equal or less than the actual cost, whereas the optimal path is neglected possibility. In general shortest path search, the estimated cost from current node to goal node always is the weight of the direct Euclidean distance (straight line).

For the description of the A* algorithm, we need to mention that a successor of a node \( p \) is
Development of a shortest path searching software in 2D

another node \( q \) which is the node of an arc from \( p \) to \( q \). The basic idea of the A* algorithm(Javris [10]) is following; Outputs of the algorithm are the least cost path from start node to goal node. E.g. Figure 2.7.

1. Starting with the initial (start) node, construct an open list of nodes which are maybe traversed with their values \( \hat{f} \).

2. From the open list, it takes the node \( n \) with the least cost (the lowest value \( \hat{f} \)), and places the node on a list designated as closed.

3. Calculate \( \hat{f} \) for every successor of the last node \( n \) placed on the closed list, and then place the successors not are on the closed list with their new values \( \hat{f} \) on the open list.

   Replace any node \( n' \) which is a successor of \( n \) on the closed list and for which \( \hat{f}(n') \) is smaller now than past on the open list.

4. Repeat from step 2 until the successor of \( n \) is the goal node. The search is terminated.

   ![Figure 2.7: An example search by using A* algorithm, the estimated distance \( \hat{h} \) here is the Euclidean distance<3>](image)

At last, the A* algorithm is admissible and optimal (given by Hart, Nilsson, and Raphael [11]). We can use A* algorithm in a shortest path searching problem with the obstacles, by considering the start and goal points and all obstacle vertices as the nodes.
2.6 Visibility Graph

Furthermore, to use the shortest path algorithm above, we need to compute the visibility graph of the graph first. A visibility graph is a kind of graph, where every edge incident to a vertex shows the direction of the other visible vertices (two vertices $v$ and $w$ are mutually visible if the segment of $vw$ does not intersect the interior of any obstacle, and the segment is called the visibility edge), as shown in Figure 2.8. There are two elements contained in visibility graph, which are vertices and edges. The vertices of the visibility graph are the vertices of polygons and the start and goal points. The edges are the segments of pairs of mutually visible vertices, in other words, the edge set of the visibility graph is a subset of all possible edges between the vertices that do not intersect edges of given polygons at precisely one point.

![Visibility Graph](image)

Figure 2.8: Visibility Graph

There are many methods for computing visibility graphs, Lee [12] gave a algorithm with running time $O(n^2 \log n)$, Welzl [13] described an $O(n^2)$ algorithm. Recently, the optimal $O(m + n \log n)$ algorithm was given by Ghosh and Mount [14].

An ordinary idea to compute the visibility graph is following:

**Input:** A set $S$ of disjoint polygonal obstacles.

**Output:** The visibility graph $G_{vis}(S)$

1. Initialize a graph $G = (V, E)$ where $V$ is the set of all vertices in $S$ and $E = \emptyset$.
2. For all vertices $v \in V$, do check other vertices $w \in V$ except $v$, if $w$ is visible from $v$, then add the arc $(v, w)$ to $E$. 
2.7 Summary

Following the literature review, we have found developing a perfect and optimal path searching software is constrained with the running time of several algorithms. They are convex hull algorithm, visibility graph algorithm and graph search algorithm. Otherwise, implementing this software in three dimensions has more constraints needed to be considered. Such that the objectives of this project have been modified:

- Researching the algorithms needed in this project, we find out the nearest optimal and simple algorithms to implement.
- The software developed is implementing in two dimensions.
- The software allows users modeling the convex polygons, but not all types of polygons.
- The software shall display the data graphically.
Chapter 3

Requirement

3.1 Introduction

Usually the requirements are the descriptions of the services provided by the software and some constraints for operating, and they help to the further design and implementing for the software (Sommerville [15]). In software development, we should try to design the software satisfied these requirements, in order for the software to be implemented efficiently and correctly. Requirements are typically classified to functional requirements and non-function requirements, and description of each are explaining in more detail in later sections. The following section describes the requirements analysis and specification according to the literature review and the objectives of this project.

3.2 Requirements Analysis

The first step of requirements analysis is to discover the requirements and to give some sort of justification for these requirements for software design. We should be pointed out that the literature review does not cover any existed system related to the project; the requirements hereby will be focused on the main aim of this project, but also based upon human oriented design and knowledge gained from the literature review. The requirements are classified with different priorities.

Priority: This is telling users what the requirements level is, with options of high, medium and low. High means the statement is most important in this project to achieve the objectives. Low means the less important.

Rationale: The justification of a particular statement for the service of the software.

The next step is to organize the requirements into groups of related requirements. For this project, the requirements have been categorized into functional requirements and
Development of a shortest path searching software in 2D

non-functional requirements. The statements of functional requirements describe the services provided by this software, the reaction for some particular inputs and the particular behaviors. Further, the functional requirements may also describe the software should not do. Non-functional requirements are requirements not directly concerned with the functions or methodologies used in the software, but are related to the features of the software, such as usability, reliability, efficiency and so on.

Following the analysis, requirements were created as a specification, as shown in the next section. In order to let users to understand the functionality provided without any professional knowledge, the language used in specification is simple and concise.

3.3 Requirements Specification

3.3.1 Functional Requirements

(1) The software should be able to display the input obstacles on the canvas (plane).
Priority: high
Rationale: Users can view the obstacles globally on the computer screen and know what they have input in the software. This is an important feature of the software that provides the users with a general overview of the current state.

(2) The software should be able to allow users to input the obstacles on some specify position on the canvas.
Priority: high
Rationale: Obstacles in a real world setting may be found in different positions. Users will want to model the obstacles on the canvas and how they are configured in the real world.

(3) The software should allow user to identify the start and goal position for the mobile object.
Priority: high
Rationale: Different start positions and different goal positions will be produced different paths for the mobile object.

(4) The software should calculate the shortest path distance between the start and goal position.
Development of a shortest path searching software in 2D

Priority: medium
Rationale: This allows users to know the precise route through which the mobile object travels.

(5) The software should automatically define the interior point of the mobile object.
Priority: high
Rationale: When the software finishes constructing the forbidden space (in Chapter 2), the polygon mobile object will be considered to be a point mobile object. The interior point is useful when defining the point mobile object.

(6) The software should allow users to modify the obstacles. E.g. deleting the existed obstacles or moving some particular obstacles.
Priority: high
Rationale: Sometimes users want to change their original decision for the placement of obstacles; a deletion method would help users to rectify such problem.

(7) The software should be able to input some different shapes of obstacles.
Priority: high
Rationale: There are many different types of convex polygons. In this project, the obstacles will be modeled as these polygons.

(8) The software should be able to define different color for obstacles before input.
Priority: medium
Rationale: Different colors of obstacles give a good global perspective of the setting. Users may want to associate particular obstacles with a special meaning; they can use different color codes to group these related obstacles together.

(9) The software should be able to change the canvas size
Priority: medium
Rationale: In some case, there may be a lot of obstacles modeled on the canvas simultaneously. A larger canvas allows more space to input these obstacles, while maintaining a faithful representation of the scene for users. Besides, a bigger canvas allows users to see a more detailed rendering of obstacles.

(10) The software should allow users clear the canvas.
Priority: low
Rationale: Sometimes users may want to delete lots of obstacles on the canvas, so a simple clear method would let users clean the canvas easily.
(11) The software should be able to build a new canvas.
*Priority:* high
*Rationale:* All the parameter in the software will be initialized on building a new canvas. Users can also use it to make a new project without restarting the software, saving time.

(12) The software should allow users to open and save file.
*Priority:* medium
*Rationale:* If users cannot finish their projects in one sitting, they can save their current work as a particular file. Users also can open their previous saved work to resume, and can also review their finished work.

(13) The software should be able to find the shortest path for the mobile object.
*Priority:* high
*Rationale:* Using a graph search algorithm, the software produces a shortest path on the visibility graph. This requirement is the most important feature of the software and the primary objective of this project.

(14) The software should display the visibility graph between the start position and the goal position with bright color on the canvas.
*Priority:* high
*Rationale:* This requirement allows users to see the path generated by the software clearly on screen and lets users know what the exact path is.

(15) The software should be able to construct the forbidden space by using the convex hull algorithm and Minkowski sum for vertices of obstacles.
*Priority:* high
*Rationale:* From the literature review, we know that we should create the forbidden space and the free space before looking for a shortest path. So it is necessary that the software can be applied to build the forbidden space necessarily.

(16) The software should display the coordinates of cursor on the canvas.
*Priority:* high
*Rationale:* Displaying the coordinates on the interface would allow users to easily find out the obstacle's position on the canvas, and would help users to draw particular obstacles. Moreover, the coordinates of each vertex of the obstacles would be used to further calculations.
3.3.2 Non-Functional Requirements

(17) The software should have an understandable, clear and usable interface.
Rationale: A clear, unobtrusive interface lets users identify the buttons, the menus and the drawing area immediately, which minimizes the time wasted navigating around the software and allows the user to focus on the task in hand.

(18) The interface of the software should make use of tooltips for each of the menus and buttons, together with plenty of prompts.
Rationale: Using supplementary text for describing the operations on menus and buttons significantly reduces the time needed to train users. Carefully placed prompts would help users to understand how to use the software only as and when appropriate.

(19) The system environment should support memory-intensive tasks.
Rationale: When users input a lot of obstacles, the software will record every obstacle’s data, which means using lots of memory to repaint on the canvas.

(20) Users should be able to use the software confidently after reading the documentation.
Rationale: A good introduction should not require too much time for users to understand usage of the software.

(21) After drawing an obstacle, the application should respond promptly, in order for the user to carry out other tasks.
Rationale: This requirement hinges on the hardware environment as well as on the efficiency of the rendering algorithm and calculations. The outcome will be to allow users to draw lots of obstacles without prolonged waiting times.

(22) The software should be able to construct every algorithm’s result within 10 seconds.
Rationale: Less waiting time let the software work economically and efficiently, same as above.

(23) The software should be able to handle input error and display error warning message.
Rationale: Decent error handling allows the software to detect when a fault has arisen, and should prevent the system from crashing. Warning messages and validation should halt further execution and notify users with the relevant information.
3.4 Project Constraints

3.4.1 Time

The project should be finished before 27th April 2009. Due to other subjects’ coursework, about 25 hours will be spent on the project every week. Before submitting, the project needs to be formatted; this also costs time.

3.4.2 Software resource

For the implementation of this project, we are going to use programming development software. The software is not variable in the BUCS computer, so we need to use our computer with this.

3.4.3 Hardware resource

Mouse and keyboard are needed and a computer with the Java platform accessible.
Chapter 4

Design

4.1 Introduction

The requirements have been completed in last chapter, such that the design could be created. In this chapter, we develop the design from the requirements specified. The system in this project is named RobotPathSearch. The names and usage of design models or styles used in this chapter are gained from Sommerville [15].

4.2 System Organization

The organization of a system is the basic strategy to structure a system. There are many organizational styles to use, such as shared data repository style, shared services and server style and layered style. Usually, a large system can be separated into some related sub-systems. In this project, I used the repository model and the layered model for the whole system and sub-systems.

The repository model is such that sub-systems contained in a system must exchange information to work together. We could design in a way so that all sub-systems can access all shared data held in a central database. The benefits of this model include allowing data generated by one sub-system to be used by another sub-system, and that data are organized around a database. This is suited to my system, because data modified by one sub-system affects other sub-systems.
Figure 4.1: The repository model of the RobotPathSearch system. The system has a shared central database which is named Main Canvas. All sub-systems can access the data held in Main Canvas and input data into it. The implementation of Main Canvas would meet requirements (11) and (14).

**Saving System** Designing for requirement (12), this sub-system is used to save data and open previous data. Users can use the sub-system to back-up their current projects and review finished projects.

**Tools System** This sub-system is contains tools and buttons which are used to perform arbitrary actions available. Handling of user events with the tools and buttons is also included in the system.

**Graph Constructor** This sub-system is used to create particular convex polygons.

**Menu System** This sub-system generates a menu bar for the system, and handles users’ actions as well.

**Graph Modifier** This sub-system is used to modify existing polygon data in Main Canvas. After editing a particular polygon, the sub-system outputs the details of the changes into the Main Canvas, and then the Main Canvas asks the Graph Constructor to re-generate the polygon. E.g. delete, undo.

**Color Generator** This sub-system generates any color given a user’s input and constructs a color panel in Main Canvas. The system was designed to meet requirement (8).

**Algorithm System** This sub-system contains all of the algorithms used to calculate data in Main Canvas.
After the system organization model has been chosen, we would decompose the sub-systems into modules. We used object-oriented decomposition (Appendix A) to illustrate the modules. The object-oriented decomposition decomposed a system into a set of relative objects. In the diagram of the object-oriented decomposition, we illustrate the object classes, attributes and operations; these are gained from the Literature Review and the Requirements.

### 4.2.1 Algorithm System

We also used the layered model to organize the Algorithm System. Each layer in the layered model provides a set of services. As a layer is completed, services provided can be available to users. The layers are changeable and portable, as the interface is unchanged, a layer can be replaced by another equivalent layer.

![Figure 4.2: The layered model of Algorithm System](image)

Figure 4.2 shows the layered model of Algorithm System. For this project, taking from the literature review and requirements (4), (13) and (15), we need to apply several algorithms to achieve the final results. The algorithms should be used step by step in the system, so the Algorithm System in this project should run from top to bottom in the figure, but data could be disposed by current completed algorithms without remaining which are not completed. Moreover, algorithms in each layer are changeable, for example we can replace them with many alternative algorithms that have been published as described in the literature review.

### 4.2.2 Graph Constructor

The Graph Constructor is a sub-system that creates particular polygons. Many types of convex polygons will be used in modeling obstacles; we need to group these convex polygons into several classes for organizing conveniently. Each class could be considered to a sub-system in Graph Constructor. Besides this, the polygon data will be used for other
Development of a shortest path searching software in 2D

calculations, so that accessing data from the sub-system should be efficient and reliable. For these reasons, this sub-system was designed in a centralized control model.

The Control model is one of a few design architecture styles and it is used to illustrate the control flow between sub-systems. The Centralized control model is a commonly used control style and is usually applied to applications that specify one sub-system as responsible for controlling other sub-systems.

Furthermore, in a centralized control model, one sub-system is designed as the system controller and is in charge of executing other sub-systems. In this project, the sub-systems (polygon classes) were executed in parallel, this meant that creating a triangle was independent of creating a rectangle. The centralized control model selected for the Graph Constructor is the manager model. This model was chosen so that a process of a sub-system could execute in parallel with other processes and one component of the system could control other sub-system processes.

Figure 4.3: the manager model of Graph Constructor

From the requirement (7) described in the last chapter, we needed to draw different types of polygons on the canvas. As shown in the manager model in figure 4.3, the Graph2D class was designed as a header system which contains and controls common data of all polygons. All different types of polygons were considered as the sub-systems. Constructing a polygon in the system must be done through the Graph2D class to invoke the particular polygon class. Lines, triangles and rectangles were the most common shapes to use, so I created individual classes for each of them. The class named Polygon was used to create all non-regular convex polygons. The Point class was designed to meet requirements (3) and (5). The start and goal positions and the interior points would be displayed as points on the canvas.
4.3 Block Diagram

After organizing the system, we can classify the sub-systems in groups. This Block Diagram illustrates the outline of the whole system. As shown from figure 4.4, the system has been grouped into 4 sections. The arrows in the figure indicate the connections and interactions between the sub-systems mentioned in the last section and the Main Canvas. The interaction between Graph Constructor, Graph Modifier and Main Canvas meets requirements (1), (2), (6) and (10). Menu System, Tools System and Color Generator are arranged together, because they are all related with the user interface.

4.4 Event –processing Model

The system has been designed by using the repository model, the layer model and one centralized control model, but also the architectural model for the system is mainly based on the event-driven model, as the system responds to events through the user interface. The user interface design will be described in a later section.
Main Canvas – This is responsible for interfacing with external API, for creating graphs, running algorithms and storing data.

Saving System – This is responsible for saving and loading data.

Ancillary Data – This object handles any other commands not directly related to Main Canvas.

Command – This object handles the commands generated by various events, and calls the methods in Main Canvas.

Event – This object creates commands from the specific events generated through the user interface.

Display – This object interfaces with the Screen and calls the refresh method when the display needs to be updated.

Screen – Responsible for capturing events and refreshing.

Figure 4.5: the event-processing model
4.5 User Interface Design

User interface design is concerned with the usability and execution of interactions between users and the system. The way chosen to design the user interface reflects the requirements which are outlined in chapter 3 of this project. For example the requirement (17) specifies a clear and usable interface. The interface achieves this by clearly separating the control area and the drawing area that is displayed. User interaction and information presentation are two important factors in interface design. They will be described in this section first, and we will introduce user analysis and user interface prototyping later.

4.5.1 User Interaction and Information Presentation

User interaction means releasing commands and related data to the system. Three styles of interaction will be used in this project.

- **Direct manipulation** This style basically means the users interact directly with objects on the screen. Usually, the direct manipulation indicates the action and the object to be manipulated with a pointing device (a mouse or a finger on touchscreen). The speed, intuition and learnability are the advantages of this interaction, but implementation is hard and it is constrained by external hardware (visual equipment).

- **Menu selection** Users can invoke a command from a menu. The command operates on a selected object on the screen. The advantages are avoiding user-related errors and not needing as much typing. The disadvantages are that the manipulation can become complex as the menu options accumulate.

- **Form fill-in** Users type in data in the field of a form, and some fields are associated with a menu or with action buttons. The disadvantages are that users may cause problems or errors when they type in information.

In this project, direct manipulation is mainly used for interaction between users and the system. For drawing polygons on the canvas and using action buttons, a mouse is used for the pointing device. Besides satisfying requirement (3), using a mouse is easier than typing in data. This is the primary manipulation method through the interface. Menu selection and form fill-in styles will be assisting, for example the delete method is invoked in the menu selection, reflecting requirement (6). Form fill-in styles will be combined with menu selection, for example the method for changing the size of the canvas is invoked using a menu, but also needs users to input the particular size in a field.
I will present information to users on the screen with a graphical representation. The reason for using this representation is that this project revolves around computing graphical data. Displaying obstacles and forbidden spaces, constructing visibility graphs and displaying a shortest path are all based on a graph.

4.5.2 User Analysis

The user analysis does not generate very specific interface requirements, but they help to understand the needs of system users. If we take users’ concerns and users’ constraints into account, the interface design will be more acceptable to users. This can reduce the time cost in design modification.

Users may have no technical knowledge of the computation algorithms that are used to build shortest paths. Because of this, the interface must avoid using technical algorithm terms. Maybe users do not have time to read through the whole documentation to know how to use the system. The system should provide prompts for each action button and each menu displayed in the interface, reflecting requirement (18), otherwise, errors will be easily generated through input mistakes. Error messages should be provided in the interface, which is a useful way to prevent these erroneous inputs or mistakes happening (usually to the same users) and is also an efficient way to stop the error immediately to ask the user to modify the input, instead of allowing the input data to propagate through further execution.

Action buttons and menus are the tools for user to invoke methods in the system. For usability reasons, the action buttons and menus should be rendered proportionately in the user interface. With this, users will be able to see the positions of the buttons and menus distinctly and can manipulate them using the mouse easily. Moreover, decent spacing and separation between buttons is good, because users may accidentally use the wrong button or menu unintentionally. Disabling particular buttons and menus is a method for avoid this problem. For example, in the menu, ‘undo’ is set to be disabled before deleting a polygon, so then users cannot use the ‘undo’ method without deleting first.

When users input obstacles, they may want to know the coordinates on the canvas. Coordinates are used to help visualize the obstacles at a particular position, so the user interface should be able to display the current coordinates of the cursor. This meets requirement (16).
4.5.3 User Interface Prototyping

Figure 4.6 shows a prototype of the user interface of this system. According to the design above, I plan to have three parts in the interface. Canvas would be the component that displays the input data and results. For example obstacles, the forbidden space and the shortest path would be displayed on this canvas. Action buttons will be used to initiate the methods of the graph constructor and the algorithm system. Users will be able to use these buttons easily with mouse clicking. Saving system state and graph modifier will be contained in the menus component.

4.6 Programming Language Selection

Through the design and literature review, Java was selected as the programming language for developing the system. The reasons for choosing Java are:

- Java is an object-oriented programming language with a built-in application programming interface (API) that can handle graphics and user interfaces. The graphics manipulation is the fundamental objective of this project, so the API is very useful to the implementation. The user interfaces of this project can be easily implemented by using the swing toolkit. Due to the plentiful API functions in Java, the system development is simple.
For safety and stability, Java incorporates an exception handling process. The process protects an execution from crashing the application with errors and exceptions. In this project, events generated by users may have exception or errors. With exception handling, the system can be more usable and reliable.

Portability: once the system is compiled, the code produced can be read and executed by any platform that can run Java.

4.7 Summary

The high level design of this project has been already completed in this chapter. The design has met the majority of the requirements, and the remainder will be reflected upon in later implementation. The implementation described in later chapters may be different from the design, because actual implementation will be affected by the algorithms we choose and the constraints of the API in Java.
Chapter 5
Implementation and Testing

5.1 Introduction

Last chapter, we have designed the system using high-level design architectural models and styles. In this chapter, we are going to implement the system following the design mainly and detail the design. We mainly concerns with the key sub-systems in the following session to achieve the aim and objectives of this project. From the requirements and the design, main canvas, graph constructor, algorithm system and user interface would be essential parts in the implementation. Because of the time constraint, we have not done the implementation of saving system. The implementing for the saving system will be on the further work. The source code of several important classes is in Appendix D, and the full source code will be in the disc.

5.2 Main Canvas

This sub-system is the most important in the system, which displays the obstacles, the robot and output from the algorithm system. In section 2.2, we have given the description and applications of the configuration space. We could think of the main canvas as the configuration space. From the repository model shown in the design, main canvas contains all of the data of input obstacles and robot. For the further manipulation, the data in main canvas should be able to access. In the field of the main canvas class, we should initialize all the collections for storing data.

In the design, we mentioned three styles of interaction between users and the system. The first usable one is direct manipulation (see section 4.5.1) in this sub-system. Assuming an instance of main canvas is a white canvas, we extended this class to JPanel that is in API. Otherwise, for performing a mouse to be a point device, we use MouseListerner and
Development of a shortest path searching software in 2D

MouseMotionListener interface.

```java
public class MainCanvas extends JPanel implements MouseListener, MouseMotionListener
{
    ……
}
```

The functions of these interface used in the system are clicked, dragged, released and move. We implement these methods following the requirements and the design. Requirement (3) said users could be able to define the start point and goal point for the robot, we can reflect this by using clicked function. When users request to define the start point on user interface, basing on the event-processing, the common will send to the main canvas class. With that, the main canvas will revive data of the start point by user clicking on the canvas and display a red point. We use this same principle for the goal point as well, but for reorganizations that the goal point would display with color blue. Moreover, this principle is also useful to input the non-regular polygons. From the literature review, we just performed convex polygons in our system. Users defined the vertices of a particular polygon with left click using mouse, and then right click performed a finished common to main canvas. After that, main canvas applied convex hull algorithm (section 2.3.2) from the algorithm system to construct a convex polygon.

For reflecting the requirement (6), users should be able to delete a particular obstacle. In the implementation, combining clicked method with move method of MouseMoveListener, the system would test the cursor whether is on an obstacle. If the cursor was on an obstacle, the type of the cursor would change, and then click mouse would select this temporal obstacle, so that we defined a testCursorInShape method in the class. By testing the data of input obstacles, the method returned true when the cursor on obstacles and false for not. If there is no any obstacles on the canvas, the method return false directly.

```java
public boolean testCursorInShape(int nowX, int nowY) {
    if(shapes.size() == 0) {
        return false;
    }
    if(isDragging) {
        return true;
    }
    for(int i = shapes.size() - 1; i >= 0; i--){
        Graph2D temp = shapes.get(i);
        if(temp.shape.contains(nowX, nowY)) {
            graph2D = temp;
        }
    }
    return true;
}
```
Development of a shortest path searching software in 2D

```java
return true;
}
}
return false;
}
```

After every user action, the system should scan the data once and display them. The reason is that the system kept updating information for users, in order to let users know what they have done currently. This is reasonable in the event-processing model. In order to achieve this purpose, we define a paint(Graphics2D g2d) function in the class. In all the event handle function, which are mouseClicked, mouseDragged, mouseRelease and mouseMove, we called paint(Graphics2D g2d) method. The paint(Graphics2D g2d) scan lists of data by using for loop, and then draw them on the canvas with special requests.

The relation and communication of the classes have been shown in the block diagram in last chapter. Some of these classes would be relative with this main canvas class, such that we should initialize the classes in the field for further manipulation. But there is a different between our implementation and the design. We were not building the Graph Modifier as a single separate class, and the methods used in Graph Modifier would just in the Main Canvas.

### 5.3 Graph Constructor

Firstly, we will discuss the graph constructor. All the obstacles and the robot modeling are basic on this sub-system, and this sub-system is also important to construct the outcome from algorithm system. Lots of the requirements mentioned are required on this sub-system. Following on the design in last chapter, we used a centralize control model to structure the graph constructor. We have seen the diagram of the control model above. The header class name Graph2D should contain common data for the different types of polygons. For example, the color used to draw, the stroke width and the vertices of the polygon. Because of these, we can build this class as an abstract class. The common data should be in the field of this class and the common method draw and setVerticesData.

```java
public abstract class Graph2D {
    public Color color;
    public float strokeWidth;
    public VerticesData verticesData;

    public void draw(Graphics2D g2d)
```
public void setVerticesData()
......
}

For each polygon building in the system, we are going to construct a little vertices database for them. Because the vertices data are mapped and would be used in the algorithm calculation, we built another class name VerticesData first. The data of vertices of each polygon would be stored in lists by order. In java API, there are many types of collections classes used for storing list of object references, for example ArrayList and LinkedList. We are using LinkedList class in our implementation. The type of the LinkedList used is Point which represents the vertex.

```java
public class VerticesData {
    public LinkedList<Point> vertices = new LinkedList<Point>();
    public VerticesData(){
    }
    public void addVertex(Point p){
        vertices.add(p);
    }
    ......}
```

Normally, the regular convex polygon in the system can be constructed applying the Rectangle class in API, for example square and rectangle. There is a little problem here, as shown in the design model, there is a Triangle class inherited Graph2D, so how can we construct a triangle? We could use Polygon class to solve this problem, and also Polygon class could be used to in the polygons class of the system. All of the polygon obstacles data would be gained from the MainCanvas class. The shape of the robot used in the system was a rectangle.

### 5.4 Algorithm System

The algorithm system has been design in a layered model and contained four sub-systems, which are MinkowskiSum, ConvexHull, VisibilityGraph and Graph Search. In the following section, we will describe the algorithms used in the system and how can we implement them.
5.4.1 MinkowskiSum

Following the definition of minkowski sum in the literature review (section 2.4), we simply implement this just adding two sets of points. The sets of points could be considered as two polygons. Actually in our system, we do not need all of the points inside the polygons. The reason of constructing minkowski sum of two polygons is that we used the output to calculate the grown obstacles (section 2.4). Because of this, we just input the vertices data of obstacles and the robot and added the vertices of robot to the vertices of obstacles. We scanned the collection of vertices of a particular obstacle by for loop. For each vertex of the obstacle, we scanned each vertex of the robot and added the reflection about the origin in the canvas to the vertex of the obstacle.

```java
public void calculateSum(){
    int x1,y1,x2,y2;
    int temp_x,temp_y;
    for(int j=0;j<obstacle.size();j++){
        for(int i=0;i<robot.size();i++){
            x1 = robot.get(i).x;
            y1 = robot.get(i).y;
            x2 = obstacle.get(j).x;
            y2 = obstacle.get(j).y;
            temp_x = x1+x2;
            temp_y = y1+y2;
            Point p = new Point(temp_x,temp_y);
            sum.add(p);
        }
    }
}
```

5.4.2 Convex Hull Algorithm

In the literature review section 2.3.1 and 2.3.2, we have described the problem to represent convex hull and some convex hull algorithms. Looking back in section 2.3.2, we have introduced four convex hull algorithms, there are two algorithms have the lower running time $O(n \log n)$ . The two algorithms are Graham’s algorithm and incremental algorithm. They are similar, but which one is the best for our system? Firstly, we need to know a basic mathematical method in java API, which is Math.atan2(double y, double x). This method converts rectangular coordinates (x,y) to polar. Both of the algorithms needed to sort all of the
Development of a shortest path searching software in 2D

points angularly in anti-clockwise direction, such that we could use Math.atan2 method to obtain angle between two given points. But return of the method is a polar angle, so that we should convert the polar angle to a pre-tilt angle.

```java
public double angle(Point v,Point w){
    int x1 = v.x;
    int y1 = v.y;
    int x2 = w.x;
    int y2 = w.y;
    
    double temp;
    temp = Math.atan2(x1-x2, y1-y2);
    temp = temp*180/Math.PI;
    return temp;
}
```

We could use the function above to compare two points’ angle with a base point, and then we have arranged the points angularly in order. Secondly, we needed to find out a point on which side of a given straight line. Why we should do this? Because of in the definition of convex hull, all the given points should be inside the convex hull. Easily gained from the figure 2.4 in section 2.3.1, all of the points should on one side of the straight line constructed with two points that are on the convex hull. We could use this rule to find out all of the points on convex hull. In the implementation, we achieved this object by using a geometry mathematical method. We knew a straight line function is \( a \times x + b \times y = 0 \), which x and y is the coordinate of a point that on the line, and the function of a straight line could be gained from two given points on the line.

```java
public void getLineFunction(Point p1,Point p2){
    int x1 = p1.x;
    int y1 = p1.y;
    int x2 = p2.x;
    int y2 = p2.y;
    double temp_y = (double)y1-y2;
    double temp_x = (double)x1-x2;
    a = temp_y/temp_x;
    double temp_b = (double)a*x1;
    b = (double)y1-temp_b;
}
```
In the Euclidean space, if a point has coordinate \((x, y)\), we can test this point’s position roughly by the line function. The rules are following:

1. If \(a \times x + b > y\), then the point is on the left side of the line.
2. If \(a \times x + b < y\), then the point is on the right side of the line.
3. If \(a \times x + b = y\), then the point is on the line.

These three rules are not definite, because of the point on the left side of right side of a straight line is depend on the direction of this line. With this reason, we could be confident to pick up a suitable algorithm for our system. Graham’s algorithm just used one boundary point and kept left turn to find the convex hull. This way was hard to know when we should define a point in the convex hull should be on left side or right side of an edge of the convex hull. Incremental algorithm used two boundary points, that are left-most and right-most, to separate the convex hull to upper and lower, as shown in figure 5.1.

![Figure 5.1: Upper convex hull and lower convex hull](image)

In a lower convex hull, the points in the convex hull should be on the left side of the edges, but the points should on the right side of the edges in the upper convex hull. In the implementation, we found the left-most (lowest) point and the right-most (highest) point by comparing their coordinates. Because the origin of the canvas is on the left-top corner, the left-most (lowest) point should have the biggest y-coordinate and least x-coordinate. We could use the same idea for looking the right-most (highest) point.

```java
public void lowest()
{
    lowest = new Point(10000,-1);
    int y, int x;
}
```
for(int i=0;i<vertices.size();i++){
    y = vertices.get(i).y;
    x = vertices.get(i).x;
    if(y >= lowest.y && x <= lowest.x){
        lowest = vertices.get(i);
    }
}
origin = lowest;
lowerConvexHull.add(lowest);
this.removeVertices(lowest);
}

After getting the lowest point and the highest point, we rearranged the points about the lowest boundary point angularly in anti-clockwise wise with comparing angles. This means the point has lowest angle would be stored as the first element of a list. Then we could get the points on the convex hull by sorting the list of points.

5.4.3 Visibility Graph Method

Before getting the shortest path from the start point to the goal point, we should build the visibility graph (section 2.6). We constructed the visibility graph based on removing the invisible lines from all of the lines. In the implementing, we connected all lines between the start point, the goal point and the vertices of grown obstacles firstly and then followed the below steps to remove the invisible lines.

1. First, we removed the lines that all of the points on the lines were contained in the grown obstacles (the forbidden space).
2. Second step is to remove the invisible lines between the start point and the vertices and between the goal point and the vertices. We were easy to get build the edges of the grown obstacles with the points got from the incremental algorithm above. If a line between start point and a vertex was intersected over two edges, we would remove the line. Because of that if the vertex is visible, the line between it to the start point has just two intersections with the edges. We could also use this rule for the goal point and the vertices.
3. At last, we started to remove the invisible lines between the vertices each other by the following condition. One vertex is visible to another should be satisfied that the line between them has no more four intersection edges of all edges.
These three steps were just for that the grown obstacles have no intersection between each other. If there is a grown obstacle A intersected with a grown obstacle B, we could just remove the vertices that are inside B from A before building the visibility graph.

### 5.4.4 Graph Search Algorithm

From the literature review section 2.5, we knew the Dijkstra’ algorithm is restricted by non-negative edges. That is difficult to define the direction of all the lines (edges) in the visibility graph in implementing. Because of this, we used A* algorithm in our system. According to the four steps of A* algorithm in section 2.5.2, we needed to create two lists, named openList and closedList. The usage of these two lists is to store the temporal vertices. The estimate cost of a vertex in our system is the straight distance between the current vertex to the goal point. For storing the heuristic cost, the actual cost and the estimate cost of a particular vertex, we would use HashMap() class to perform the mapping cost and the vertex. We constructed three HashMap() instances, named f_score, g_score and h_score. The code of A* algorithm is in the Appendix D, GraphSearchAlgorithm.class.

The return of A* algorithm in the code is a HashMap() of points. They were stored in the HashMap() with no order, such that we need to build a extra function to arrange the points to get the shortest path. We scanned the map from the goal point until the path reached the start (source) point and added to a list.

```java
public void getShortestPath(){
    ....
    Point2D point2d = goal;
    Point2D current = new Point2D.Double();

    LinkedList<Point2D> sorted = new LinkedList<Point2D>();
    sorted.add(goal);

    while(!point2d.equals(source)){
        this.findPath(listOfStart, point2d);
        current = listOfEnd.get(listOfStart.indexOf(point2d));
        sorted.add(current);
        point2d = current;
    }
}
```
development of a shortest path searching software in 2D

\[
\text{public void findPath(LinkedList<Point2D> list, Point2D current)\{}
\text{\hspace{1em}for(int i = 0;i<list.size();i++){}
\text{\hspace{2em}if(current.equals(list.get(i))){
\text{\hspace{3em}point2d = list.get(i);}
\text{\hspace{2em}}}
\text{\}}
\text{\}}
\]

5.5 User Interface

Figure 5.2 has shown the interface of the system. Following the design of the user interface, we should achieve the three styles interaction in our interface. The direct manipulation has been mentioned in above section. The menu selection style was implemented by using the JMenuBar in Java API and the form-fill in was implemented by JOptionPane. For handling the wrong actions from users, the buttons and menu set to be disable before users achieved some tasks. For example, the buttons for constructing forbidden space, visibility graph and shortest path, were disable in the figure 5.2. We have mentioned the ‘undo’ and ‘delete’ problem in the design of user interface. Figure 5.3 has shown the edit menu, we set the ‘undo’ menu button to be disable before deleting obstacles.

The color generator in the interface should provide colors selection for users. In the interface, we have defined 10 colors on the color panel; users could use a particular color by clicking the square of the color. If user wanted to perform the obstacle with other special colors to be more colorful, there is color chooser windows provided in API launching with double click the color squares.

Before deleting a particular obstacle, users should select the selector button. If the selector button has been selected, the cursor would be changed style when it on the obstacles. The finished button on the tools panel was used to check whether users have defined the start point and the goal point. If users clicked the finished button before setting the start and goal point, the system would display a warning message to remind users.
Development of a shortest path searching software in 2D

Figure 5.2: User Interface

Figure 5.3: Edit Menu
5.6 Testing

Testing of a system is to aid in validation of system functionality, and to see how well the system meets the requirements mentioned in the requirements specification. Although testing has been carried out throughout the implementation of the project, system testing after completion is necessary to protect errors generated in the final system. When a new functionality added to the system, new errors would be introduces as well. Sometimes the errors were generated by the constraint between a new functionality and a previous functionality of the system.

The system testing would be based on the user interface, because all the functionalities of the system could be accessed from the user interface. The test cases for the system are derived from the requirements specification. After the testing, we would discuss requirements validation.

To achieve the main aim of this project, we should do the following test:
1. Input some obstacles on the plane (canvas).
2. Identify the start point and the goal point.
3. Construct the forbidden space (grown obstacles) for a particular robot
4. Build the visibility graph for the start point and goal point defined before.
5. Find the shortest path for the robot.

The testing table would be in Appendix C and some screenshot in Appendix B. The testing table contains the input, expected outcome and actual outcome.

In the system testing, we have found out there is a problem with building the visibility graph method. When there is an obstacles intersected with another obstacle in a special case, likes figure 5.4. The outcome of the method would have error. This case was that we missed in the implementation.

![Figure 5.4: Special case for two obstacles](image.png)
Otherwise, there was another problem with drawing a non-regular polygon. Sometime the polygon displayed on the canvas was not built with the particular vertices. This was limited users to select the vertices slowly and precisely. Maybe the reason of this happened is implementing of the convex hull algorithm, but the system constructed the forbidden space correctly with the algorithm, so we needed to do more test in the further to find out the error in the system.

5.6.1 Requirements Validation

“The software should be able to display the input obstacles on the canvas (plane).”
“The software should be able to allow users to input the obstacles on some specify position on the canvas.”
“The software should allow user to identify the start and goal position for the mobile object.”
“The software should be able to input some different shapes of obstacles.”

These requirements have all been achieved. The users could input the particular obstacles and define the start and goal point with a mouse.

“The software should calculate the shortest path distance between the start and goal position.”
“The software should be able to find the shortest path for the mobile object.”
“The software should display the visibility graph between the start position and the goal position with bright color on the canvas.”
“The software should be able to construct the forbidden space by using the convex hull algorithm and Minkowski sum for vertices of obstacles.”

These requirements have been all achieved. Users could press the buttons on the tools system to invoke these methods.

“The software should automatically define the interior point of the mobile object.”

This requirement has been achieved by defining the start point and the goal point for the robot. The start point and the goal point are the interior points of this robot.

“The software should allow users to modify the obstacles. E.g. deleting the existed obstacles or moving some particular obstacles.”

We have achieved ‘undo’ and ‘delete’ methods in the system, but not ‘move’ method.
“The software should be able to define different color for obstacles before input.”

This requirement has been achieved. The color generator provided the changing color service for users.

“The software should be able to change the canvas size.”
“The software should allow users clear the canvas.”
“The software should be able to build a new canvas.”

These requirements have all been achieved. Users could invoke these methods on the menu.

“The software should display the coordinates of cursor on the canvas.”

This requirement has been achieved. The interface displays the current position’s coordinate of the cursor on the bottom.

“The software should allow users to open and save file.”
“Users should be able to use the software confidently after reading the documentation.”

These requirements have not been achieved. Due to the time constraint, we have not implemented the saving system and have not wrote the introduction documentation for the system, but this would be on the further work.

“The software should have an understandable, clear and usable interface.”
“The software should be able to handle input error and display error warning message.”
“The interface of the software should make use of tooltips for each of the menus and buttons, together with plenty of prompts.”
“After drawing an obstacle, the application should respond promptly, in order for the user to carry out other tasks.”

These requirements have been achieved in the user interface. The warning message will be display on the interface when there is error occurred.

“The system environment should support memory-intensive tasks.”
“The software should be able to construct every algorithm’s result within 10 seconds.”

These requirements are hard to validate. Because of it is difficult to input plenty of obstacles.
Chapter 6

Conclusion

6.1 Overview

The objectives set in the introduction have been modified after the literature review. Because the initial objectives were established just following the problem description, they were not perfect in the project. In the literature review, the problems and the constraints of this project has been indicated definitely in detail. In addition the several main algorithms have been described and discussed in the chapter. The information gained from the literature review was very useful for the further implantations.

The requirements mentioned in chapter 3 were just according to objectives and the literature review, such that the requirements may be not comprehensive.

The decisions made in the design were based on the requirements specification. There were several models used in the high level design for the system. User interface design is a key part in the project. Because of the software was mainly using direct manipulation interaction with users. The user interface should be designed and implemented more careful.

During the implementation, when a new functionality added to the system, that would occur some new errors. This cost much time to modify the errors and to check the functionality many times. On other hand, due to the time constraint, the implementation of the saving system has been not achieved. This was to cause the system did not meet the requirement (12), but the requirement was independent with the objectives of the project. In the testing, there were mentioned that two errors generated in the completed system; one is concerned with the visibility graph method, the other is concerned with the convex hull algorithm.

The objectives of the project have basically achieved and the project has attained the main aim roughly. Because of there were some errors occurred by using two primary algorithms,
the achievements of the project are not excellent:

- This project presents the software, named RobotPathSearch, which can be used by users with limited technical knowledge in programming and geometry.
- A* algorithm has been implemented in the project. This algorithm is a famous path search algorithm. The usages of the algorithm are various, for example in the GPS of cars.
- The software can model convex polygon obstacles and create a shortest path among some no intersection obstacles graphically.
- The interface is simple to use, and is understandable. This will allow new users to know how to use the software quickly.

### 6.2 Further work

As explained in the chapter 5, the visibility graph constructed is not correct on a special case and the modeling of non-regular polygons has leak. These should be fixed firstly.

There was no time to implement the saving system, as described in chapter 5. The idea was that the saving system could save the current data of the canvas into a file and could load the previous canvas. The data of all of the polygons have been managed in some collection, so that the saving system can be easy to implement from the current software.

Currently, the robot in the software was just created in a rectangle shape, for the usability, the robot should be able to define in any particular shapes. The software can provide a simple service that allows users identify the shape of the robot. One idea is to build an extra canvas on the interface, and this canvas is used to model the shape of the robot.

Moreover, the software can be improved to provide more methods to modify obstacles. For examples, moving the obstacles on the canvas and rotating the obstacles.

In the literature review, we have mentioned the rotation question in the motion planning. In the further development, the software can be designed to test a shortest path for a rotatable robot. The path found will be shorter than the path of a not rotatable robot with same shape.

Finally, the software is usable on the two dimensions modeling. In the further work, improving the software on three dimensions is possible, due to the algorithms used in three dimensions are similar.
Development of a shortest path searching software in 2D

Bibliography


Development of a shortest path searching software in 2D

Pictures Derivation

<1> http://www.cs.tau.ac.il/~ophirset/robotics/ex2/Minkowski_sum_2/Minkowski_sum_2/Chapter_main.html


<4> http://alienryderflex.com/shortest_path/
Appendix A

Figure A.1: Object-Oriented Design Diagram
Appendix B

Screenshots

Figure B.1: Obstacles input and the start, goal position of the rectangle robot
Development of a shortest path searching software in 2D

Figure B.2: The forbidden space

Figure B.3: The visibility graph
Figure B.4: The shortest path and the distance
## Appendix C

### Testing Table

<table>
<thead>
<tr>
<th>No.</th>
<th>Input</th>
<th>Expected Outcome</th>
<th>Actual Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Draw a blue rectangle</td>
<td>Display a blue rectangle on the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>2</td>
<td>Draw a triangle</td>
<td>Display a triangle on the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>3</td>
<td>Draw a particular non-regular polygon</td>
<td>Display a non-regular polygon</td>
<td>Basically as expected, but sometime the output polygon is non constructed by the identify vertices</td>
</tr>
<tr>
<td>4</td>
<td>Delete a selected polygon</td>
<td>The selected polygon will be removed form the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>5</td>
<td>Undo the deleting</td>
<td>The deleted polygon will be back on the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>6</td>
<td>Select the delete menu before selecting a polygon</td>
<td>Not delete any polygons on the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>7</td>
<td>Set the canvas size with width 800, height 1000</td>
<td>The canvas will change to the particular size</td>
<td>As expected</td>
</tr>
<tr>
<td>8</td>
<td>Clear the canvas</td>
<td>There is nothing on the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>9</td>
<td>Define the start point</td>
<td>Display a red point on the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>10</td>
<td>Define the goal point</td>
<td>Display a blue point on the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>11</td>
<td>Define the start point twice</td>
<td>Display just one red point on the canvas</td>
<td>As expected</td>
</tr>
<tr>
<td>12</td>
<td>Press the finished button before define the start point and goal point</td>
<td>The system will display a warning message</td>
<td>As expected</td>
</tr>
<tr>
<td></td>
<td>Press the forbidden space button</td>
<td>Display the grown obstacles and change the robot to be a point</td>
<td>As expected</td>
</tr>
<tr>
<td>---</td>
<td>---------------------------------</td>
<td>-------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>14</td>
<td>Press the visibility graph button</td>
<td>Display the visibility graph with blue color</td>
<td>Basically as expected</td>
</tr>
<tr>
<td>15</td>
<td>Press the shortest path button</td>
<td>Display the shortest path with red color and distance</td>
<td>As expected</td>
</tr>
<tr>
<td>16</td>
<td>Select the new menu</td>
<td>Give a new canvs</td>
<td>As expected</td>
</tr>
<tr>
<td>17</td>
<td>Select the about menu</td>
<td>Display a message window</td>
<td>As expected</td>
</tr>
<tr>
<td>18</td>
<td>Move the cursor on the canvas</td>
<td>On the bottom of the interface will display the coordinate of the cursor on current position</td>
<td>As expected</td>
</tr>
<tr>
<td>19</td>
<td>Press the rectangle button</td>
<td>On the bottom of the interface will dispaly the status : rectangle</td>
<td>As expected</td>
</tr>
<tr>
<td>20</td>
<td>Input non integer to change the canvas size</td>
<td>Display an error message and does not change the canvas</td>
<td>As expected</td>
</tr>
</tbody>
</table>
Appendix D

Parts of Source code

The following pages contain source code of the algorithms and main canvas class.
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;
import java.util.HashMap;
import java.util.LinkedList;

public class GraphSearchAlgorithm{

    public LinkedList<Line2D> visibleLine = new LinkedList<Line2D>();
    public Point2D source;
    public Point2D goal;
    public LinkedList<Graph2D> initialvertices = new LinkedList<Graph2D>();
    public LinkedList<Point2D> vertices = new LinkedList<Point2D>();
    public LinkedList<Line2D> test = new LinkedList<Line2D>();
    public LinkedList<Point2D> openList;
    public LinkedList<Point2D> closedList;
    public Graph2D points;
    public Point2D vertex;

    public GraphSearchAlgorithm(LinkedList<LinkedList<Line2D>> set,
                                  LinkedList<Graph2D> vertices){
        for(int i=0;i<set.size();i++){
            LinkedList<Line2D> temp = set.get(i);
            for(int j=0;j<temp.size();j++){
                visibleLine.add(temp.get(j));
            }
        }
        this.initialvertices = vertices;
    }

    public void getSourceAndGoad(){
        double x;
        double y;
        double source_x;
        double source_y;

        points = initialvertices.getFirst();
        source_x = (double)points.getX1();
        source_y = (double)points.getY1();
        source = new Point2D.Double(source_x, source_y);
    }
}
Development of a shortest path searching software in 2D

```java
points = initialvertices.getLast();
x = (double)points.getX1();
y = (double)points.getY1();
goal = new Point2D.Double(x, y);

for(int i=1;i<initialvertices.size();i++) {
    points = initialvertices.get(i);
    x = (double)points.getX1();
    y = (double)points.getY1();
    vertex = new Point2D.Double(x, y);
    vertices.add(vertex);
}
}

public void setEdges()
{
    boolean exist = false;
    test.add(visibleLine.getFirst());
    for(int i = 0;i<visibleLine.size();i++){
        Line2D line = visibleLine.get(i);
        Point2D p1 = line.getP1();
        Point2D p2 = line.getP2();
        if(i!=0){
            for(int j = 0;j<test.size();j++){
                Line2D temp = test.get(j);
                Point2D p1x = temp.getP1();
                Point2D p1y = temp.getP2();
                if(p1.equals(p1y) && p2.equals(p1x)){
                    exist = true;
                }
                else if(p1.equals(p1x) && p2.equals(p1y)){
                    exist = true;
                }
            }
            if(!exist){
                test.add(line);
                exist = true;
            }
        }
    }
    exist = false;
}
```
Development of a shortest path searching software in 2D

```java
public HashMap sort() throws Exception{
    closedList = new LinkedList<Point2D>();
    openList = new LinkedList<Point2D>();

    openList.add(source);

    HashMap g_score = new HashMap();
    HashMap h_score = new HashMap();
    HashMap f_score = new HashMap();
    HashMap came_from = new HashMap();
    LinkedList<Point2D> neighbour = new LinkedList<Point2D>();

    g_score.put(source, Double.valueOf(0));
    h_score.put(source, this.estimate(source, goal));
    f_score.put(source, this.estimate(source, goal));

    double currentDistance = 0;

    while(openList.size()>0){
        // Calculate node in openset with minimum f_score value.
        Point2D x = source;
        double shortestDistance = 10000;
        for(int i = 0;i<openList.size();i++){
            currentDistance = this.getDouble(f_score.get(openList.get(i)));
            if (currentDistance < shortestDistance){
                shortestDistance = currentDistance;
                x = openList.get(i);
            }
        }
        // finish
        if (x.equals(goal)){
            return came_from;
        }
    }
    // finish
}
```
openList.remove(x);
closedList.add(x);

//Get neighbour vertex

for(int i=0;i<vertices.size();i++){
    Point2D point = vertices.get(i);
    for(int j=0;j<test.size();j++){
        Line2D line = test.get(j);
        Point2D p1 = line.getP1();
        Point2D p2 = line.getP2();

        if(x.equals(p1) && point.equals(p2)){
            neighbour.add(point);
        }

        if(point.equals(p1) && x.equals(p2)){
            neighbour.add(point);
        }
    }
}

for(int i = 0;i<neighbour.size();i++){
    Point2D y = neighbour.get(i);

    if(closedList.contains(y)){
        continue;
    }

    double initial_g_score = this.getDouble(g_score.get(x))
        + x.distance(y);
    boolean better = false;

    if(!openList.contains(y)){
        openList.add(y);
        h_score.put(y, this.estimate(y, goal));
        better = true;
    }
Development of a shortest path searching software in 2D

```java
else if(initial_g_score < this.getDouble(g_score.get(y))){
    better = true;
}

if(better){
    came_from.put(y, x);
    g_score.put(y, Double.valueOf(initial_g_score));
    double temp = this.getDouble(g_score.get(y))
    +this.getDouble(h_score.get(y));
    f_score.put(y,Double.valueOf(temp));
}

neighbour.clear();
throw new Exception();
}

public Double estimate(Point2D start, Point2D goal){
    return Double.valueOf(start.distance(goal));
}

public double getDouble(Object o){
    return Double.valueOf(o.toString()).doubleValue();
}
```
Development of a shortest path searching software in 2D

```java
import java.util.*;
import java.awt.*;
public class IncrementalAlgorithm {

    public LinkedList<Point> vertices;
    public LinkedList<Point> sortedVertices;
    public LinkedList<Point> upperConvexHull;
    public LinkedList<Point> lowerConvexHull;
    public Point lowest;
    public Point highest;
    public Point origin;
    public double a;
    public double b;
    public boolean leftSide;
    public boolean onLine;
    public int l;
    public int h;
    public int initializedLower = 0;
    public int initializedUpper = 0;

    public IncrementalAlgorithm(LinkedList<Point> vertices){
        this.vertices = vertices;
        sortedVertices = new LinkedList<Point>();
        upperConvexHull = new LinkedList<Point>();
        lowerConvexHull = new LinkedList<Point>();
    }

    public void lowest(){
        lowest = new Point(10000,-1);
        int y;
        int x;

        for(int i=0;i<vertices.size();i++){
            y = vertices.get(i).y;
            x = vertices.get(i).x;
            if(y >= lowest.y && x <= lowest.x){
                lowest = vertices.get(i);
            }
        }
    }
}
```
Development of a shortest path searching software in 2D

}  
origin = lowest;  
lowerConvexHull.add(lowest);  
this.removeVertices(lowest);

}

public void highest(){
    highest = new Point(-1,10000);  
    int y;
    int x;

    for(int i=0;i<vertices.size();i++){
        y = vertices.get(i).y;  
        x = vertices.get(i).x;  
        if(y <= highest.y){
            highest = vertices.get(i);
            
        }  
    }

    upperConvexHull.add(highest);
}

public void removeVertices(Point p){
    for(int i=0;i<vertices.size();i++){
        if(vertices.get(i) == p){
            vertices.remove(i);
            break;
        }
    }
}

public void sortInOrder(){
    while(vertices.size()!=0){
        sortedVertices.add(sortLowerAngle());
        removeVertices(sortLowerAngle());
    }
}

}
public Point sortLowerAngle()
{
    Point p = vertices.getFirst();

    for(int i = 0; i < vertices.size(); i++)
    {
        Point temp = vertices.get(i);
        if(angle(p, origin) > angle(temp, origin))
        {
            p = temp;
        }
        else if(angle(p, origin) == angle(temp, origin))
        {
            int x1 = p.x;
            int y1 = p.y;
            int x2 = temp.x;
            int y2 = temp.y;

            if(y1 == y2 && x2 > x1)
            {
                p = temp;
            }
            else if(x1 == x2 && y1 > y2)
            {
                p = temp;
            }
            else if(y2 < y1 && x2 > x1)
            {
                p = temp;
            }
        }
    }

    return p;
}

public double angle(Point v, Point w)
{
    int x1 = v.x;
    int y1 = v.y;
    int x2 = w.x;
    int y2 = w.y;

    double temp;
    temp = Math.atan2(x1 - x2, y1 - y2);
    temp = temp*180/Math.PI;
return temp;
}

/*public void getLineFunction(Point p1,Point p2){
   int x1 = p1.x;
   int y1 = p1.y;
   int x2 = p2.x;
   int y2 = p2.y;

   double temp_y = (double)y1-y2;
   double temp_x = (double)x1-x2;

   a = temp_y/temp_x;

   double temp_b = (double)a*x1;

   b = (double)y1-temp_b;
}

public void testLeft(Point p){
   double x = (double) p.x;
   double y = (double) p.y;

   if(y < a*x+b){
       leftSide = true;
       onLine = false;
   }
   if(y > a*x+b){
       leftSide = false;
       onLine = false;
   }
   if(y == a*x+b){
       onLine = true;
   }
}
*/

public void getVerticesOfConvexHull{
for(int i = 0; i < sortedVertices.size(); i++){
    if(sortedVertices.get(i) == lowest){
        l = i;
        break;
    }
}

for(int i = 0; i < sortedVertices.size(); i++){
    if(sortedVertices.get(i) == highest){
        h = i;
        break;
    }
}

for(int i = 0; i < h; i++){
    lowest = lowerConvexHull.getLast();
    Point p1 = sortedVertices.get(i);
    for(int j = i; j < h; j++){
        Point p2 = sortedVertices.get(j+1);
        if(angle(p1, lowest) > 0 && angle(p2, lowest) > 0){
            if(angle(p1, lowest) > angle(p2, lowest))
                p1 = p2;
        }
        if(angle(p1, lowest) < 0 && angle(p2, lowest) > 0){
            p1 = p2;
        }
        if(angle(p1, lowest) < 0 && angle(p2, lowest) < 0){
            if(Math.abs(this.angle(p1, lowest)) <
                Math.abs(this.angle(p2, lowest))){
                p1 = p2;
            }
        }
    }
}

if(angle(p1, lowest) == angle(p2, lowest)){
    int x1 = p1.x;
    int y1 = p1.y;
    int x2 = p2.x;
    int y2 = p2.y;
}
Development of a shortest path searching software in 2D

```java
int x2 = p2.x;
int y2 = p2.y;

if(x1 == x2 && y2<y1){
    p1 = p2;
}
else if(y1==y2 && x1<x2){
    p1 = p2;
}
else if(y2<y1 && x1<x2){
    p1 = p2;
}
}
}
i = sortedVertices.indexOf(p1);
if(p1!=highest){
    lowerConvexHull.add(p1);
}
}

for(int i = h+1;i<sortedVertices.size();i++){
    Point q1 = sortedVertices.get(i);
    highest = upperConvexHull.getLast();

    for(int j=i;j<sortedVertices.size()-1;j++){
        Point q2 = sortedVertices.get(j+1);

        if(angle(q1,highest)>0 && angle(q1,highest)>0){
            if(angle(q1,highest)>angle(q1,highest))
                q1 = q2;
        }

        if(angle(q1,highest)<0 && angle(q2,highest)>0){
            q1 = q2;
        }
    }
}
```
if(angle(q1,highest)<0 & angle(q2,highest)<0){
    if(Math.abs(this.angle(q1,highest)) < Math.abs(this.angle(q2,highest))){
        q1 = q2;
    }
}
if(angle(q1,highest)==angle(q2,highest)){
    int x1 = q1.x;
    int y1 = q1.y;
    int x2 = q2.x;
    int y2 = q2.y;
    if(x1 == x2 & y1<y2){
        q1 = q2;
    } else if(y1==y2 & x2<x1){
        q1 = q2;
    } else if(y1<y2 & x2<x1){
        q1 = q2;
    }
    i = sortedVertices.indexOf(q1);
    upperConvexHull.add(q1);
}
}

public void testLast(){
    int size = upperConvexHull.size();
    Point p1 = upperConvexHull.get(size-1);
    Point p2 = upperConvexHull.get(size-2);
    if(angle(p1,origin)==angle(p2,origin)){
        upperConvexHull.remove(size-1);
    }
}
Development of a shortest path searching software in 2D
import java.awt.*;
import java.util.LinkedList;

/**
 * @author Bingzhao Tan
 */
public class MinkowskiSum {
    public LinkedList<Point> robot;
    public LinkedList<Point> obstacle;
    public LinkedList<Point> sum;

    public MinkowskiSum(LinkedList robot,LinkedList obstacle){
        this.robot = robot;
        this.obstacle = obstacle;
        sum = new LinkedList<Point>();
    }

    public void calculateSum(){
        int x1,y1,x2,y2;
        int temp_x,temp_y;
        for(int j=0;j<obstacle.size();j++){
            for(int i=0;i<robot.size();i++){
                x1 = robot.get(i).x;
                y1 = robot.get(i).y;
                x2 = obstacle.get(j).x;
                y2 = obstacle.get(j).y;

                temp_x = x1+x2;
                temp_y = y1+y2;

                Point p = new Point(temp_x,temp_y);
                sum.add(p);
            }
        }
    }
}
import java.awt.*;
import java.util.LinkedList;

/**
 * @author Bingzhao Tan
 */
public class MyPolygon extends Graph2D{

    public boolean finalized;
    public int length;

    public MyPolygon(Color c,Point p) {
        sets = new LinkedList<Point>();
        verticesData = new VerticesData();
        color = c;
        this.addPoint(p);
        finalized = false;
    }

    @Override
    public void addPoint(Point p){
        sets.add(p);
    }

    public void setSets(LinkedList<Point> set){
        sets = set;
    }

    public void setFinalized(boolean finalized){
        this.finalized = finalized;
    }

    public void draw(Graphics2D g2d) {
        int size = sets.size();
        int [][] points = new int[2][size];
        for(int i = 0;i < sets.size();i++){
Development of a shortest path searching software in 2D

points[0][i] = sets.get(i).x;
points[1][i] = sets.get(i).y;
}
shape = new Polygon(points[0],points[1],points[0].length);
g2d.setStroke(new BasicStroke(this.strokeWidth));
g2d.setColor(color);
g2d.fill(shape);

public void setVerticesData() {
    for(int i=0;i<sets.size();i++){
        Point p = sets.get(i);
        this.verticesData.addVertex(p);
    }
}
}
import java.awt.*;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;
import java.util.LinkedList;

/**
 * @author Bingzhao Tan
 */
public class VisibilityGraph {
    public LinkedList<Line2D> visibleOfLine;
    public LinkedList<LinkedList<Line2D>> setOfVisible =
        new LinkedList<LinkedList<Line2D>>();
    public LinkedList<Point> initial = new LinkedList<Point>();
    public LinkedList<Line2D> segementS = new LinkedList<Line2D>();
    public LinkedList<Line2D> temp_seg = new LinkedList<Line2D>();
    public Line2D segement;

    public VisibilityGraph(LinkedList<Point> input){
        this.initial = input;
    }

    public void getSegementS(){
        for(int i = 0;i<initial.size()-1;i++){
            Point p1 = initial.get(i);
            for(int j = i+1;j<initial.size();j++){
                Point p2 = initial.get(j);
                double x1 = (double) p1.x;
                double y1 = (double) p1.y;
                double x2 = (double) p2.x;
                double y2 = (double) p2.y;
                segement = new Line2D.Double(x1,y1,x2,y2);
                segementS.add(segement);
            }
        }
    }

    public void ignoreLineInShape(LinkedList<Line2D> list){
        for(int i = 0;i<segementS().size();i++){
            for(int j = 0;j<list.size();j++){
                double x1 = (double) p1.x;
                double y1 = (double) p1.y;
                double x2 = (double) p2.x;
                double y2 = (double) p2.y;
                segement = new Line2D.Double(x1,y1,x2,y2);
                segementS.add(segement);
            }
        }
    }
}
Line2D seg = segmentS.get(i);
Line2D inSeg = list.get(j);
if(seg.getP1().equals(inSeg.getP2())
    && seg.getP2().equals(inSeg.getP1())){
    segmentS.remove(i);
}
else if(seg.getP2().equals(inSeg.getP2())
    && seg.getP1().equals(inSeg.getP1())){
    segmentS.remove(i);
}
}
public void getVisible(LinkedList<Line2D> list){
    int count_interset = 0;
    for(int k = 0;k<initial.size()-1;k++){  
        visibleOfLine = new LinkedList<Line2D>();
        Point origin = initial.get(k);
        double origin_x = (double)origin.x;
        double origin_y = (double)origin.y;
        Point2D origin2D = new Point2D.Double(origin_x,origin_y);
        Point2D origin2D = new Point2D.Double(origin_x,origin_y);
        for(int i = 0;i<segmentS.size();i++){  
            Line2D seg = segmentS.get(i);
            Point2D startP = seg.getP1();
            Point2D endP = seg.getP2();

            if(startP.equals(origin2D)){
                temp_seg.add(seg);
            }
        }
        for(int l = 0;l<temp_seg.size();l++){
            for(int j = 0;j<list.size();j++){  
                Line2D inSeg = list.get(j);
                if(temp_seg.get(l).intersectsLine(inSeg)){
                    count_interset = count_interset+1;
                }
            }
        }
    }
if(!temp_seg.get(l).getP2().equals(initial.getLast())){
    if(count_intersect < 3 && k==0){
        visibleOfLine.add(temp_seg.get(l));
    }
    if(count_intersect < 5 && k>0){
        visibleOfLine.add(temp_seg.get(l));
    }
}
else{
    if(count_intersect < 3){
        visibleOfLine.add(temp_seg.get(l));
    }
}
}

count_intersect = 0;
}
if(k==0){
    boolean existP1 = false;
    boolean existP2 = false;
    for(int j = 0;j<list.size();j++){
        Line2D inSeg = list.get(j);
        Point2D inSeg_p1 = inSeg.getP1();
        Point2D inSeg_p2 = inSeg.getP2();
        for(int z = 0;z<initial.size();z++){
            if(initial.get(z).equals(inSeg_p1)){
                existP1 = true;
                break;
            }
        }
        for(int z = 0;z<initial.size();z++){
            if(initial.get(z).equals(inSeg_p2)){
                existP2 = true;
                break;
            }
        }
        if(existP1 && existP2) {
            visibleOfLine.add(inSeg);
        }
    }
}
existP1 = false;
existP2 = false;

Point start = initial.getFirst();
Point goal = initial.getLast();
Line2D temp = new Line2D.Double(start.x,start.y,goal.x,goal.y);
if(temp.intersectsLine(inSeg)){
    for(int v=0;v<visibleOfLine.size();v++)
    {
        Point2D p1_temp = temp.getP1();
        Point2D p2_temp = temp.getP2();

        Line2D visi = visibleOfLine.get(v);

        Point2D p1_vi = visi.getP1();
        Point2D p2_vi = visi.getP2();

        if(p1_temp.equals(p1_vi) && p2_temp.equals(p2_vi))
        {
            visibleOfLine.remove(v);
        }

        if(p1_temp.equals(p2_vi) && p2_temp.equals(p1_vi))
        {
            visibleOfLine.remove(v);
        }
    }
}
}

setOfVisible.add(visibleOfLine);
temp_seg.clear();
}