COMPARISON OF PATH PLANNING IN SIMULATED ROBOT

By

CH’NG CHEE YU’NG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology
(Kampar Campus)

JAN 2020

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: Comparison of Path Planning in Simulated Robot

Academic Session: JAN 2020

I CH'NG CHEE YU'NG
(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

[illegible] [illegible]

(Author's signature) (Supervisor's signature)

Address:
No. 16, Jalan Bawal 7, Taman Mutiara,
42800, Tanjong Sepat,
Selangor.

CHANG JING JING
Supervisor's name

Date: 4/23/2020 **Date**: 4/22/2020

COMPARISON OF PATH PLANNING IN SIMULATED ROBOT

By

CH’NG CHEE YU’NG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology

(Kampar Campus)

JAN 2020

DECLARATION OF ORIGINALITY

I declare that this report entitled "**COMPARISON OF PATH PLANNING IN SIMULATED ROBOT**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature	:	[illegible]
Name	:	CH'NG CHEE YU'NG
Date	:	4/23/2020

ACKNOWLEDGEMENT

I would like to thank my supervisor Dr. Chang Jing Jing for her valuable and constructive suggestions during the planning and development of the project. In addition, thank you for giving me this bright opportunity to participate in projects related to Robot Operating System (ROS) and path planning algorithms. One million thank you.

Finally, I must thank my parents and family for their support and encouragement throughout the Final Year Project.

ABSTRACT

The development of technology is getting faster and faster, and the robotics revolution is also expanding rapidly. This is because robots can make a huge contribution to humans. The flexibility of robots has made it being capable of performing a diversity of tasks automatically (Benefits of Using Robotics, 2019). They can replace humans to do dangerous jobs, and complete the jobs more efficiently and accurately than humans. A basic task of a mobile robot is to move to a targeted point in order to perform the specified tasks. Hence, path planning algorithm such as A* algorithm and Dijkstra’s algorithm are essential for the robot to navigate efficiently from one place to the targeted place. However, different robots will work in different environments. If all the robots use the same path planning algorithm, it is possible that they may not bring the greatest benefit. This is because each path planning algorithm has its own applicable domain, performance, advantages and disadvantages in various situations. Thus, a comparison of path planning algorithm in simulated robot will be conducted in this study. There are 4 path planning algorithms will be compared, which are Dijkstra’s algorithm, A* algorithm, Rapid-Exploring Random Tree (RRT) algorithm, and the last one is an algorithm modified from A*. The concept of the modified A* is to search the path in two directions. As a result of this paper, Dijkstra's algorithm suitable for finding the shortest distance. If you want to speed up the search time and there are fewer obstacles on the map, it is recommended to use A*, modified A* or RRT. However, if there are many obstacles on the map, RRT will not be suitable for searching. Robot Operating System (ROS) and Simple Two Dimensional Robot (STDR) simulator will be used in this project.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREIVATIONS

RRT	Rapidly-exploring Random Tree
ROS	Robot Operating System
STDR	Simple Two Dimensional Robot

Chapter 1: Introduction

1.1 Problem statement and motivation

With the rapid development of science and technology, robots are considered as a significant element in society. Robot is a biologically-like machine that can move independently and perform complex actions (Definition of ROBOT, 2019). Robot can be controlled by using an external control device, or the controls can be embedded in the robot so it can operate automatically without the manual control. What is the reason why humans want to develop robots? This is because robots can do what humans cannot do. Also, robot can accomplish certain tasks are more precise and efficient than humans.

One of the benefits of robotics is that they can work in any environment (Benefits of Using Robotics, 2019). For example, humans cannot fly, but robots can. A drone is a flying robot. It can use software-controlled flight plan to fly automatically or remotely controlled (What is a Drone? - Definition from WhatIs.com, 2019). Drones are now widely used in civilian applications, including search and rescue, surveillance, weather monitoring, traffic monitoring and so on. In addition, there is a robot fish that can take video images and pictures underwater (Soffar, 2019). It can gathers information and collects artifacts about the water conditions. Even on land, there are many robots that can help humans perform certain tasks. For example, a robotic vacuum cleaner, it is a floor cleaning robot and it can clean autonomously without human control. In other words, robots reduce risky work for humans because they are capable of working in dangerous environment. Moreover, they can handle repetitive tasks, lifting heavy loads, and toxic substances. This has helped humans avoid many accidents, also saving money and time.

According to the examples above, we can find that most of the robots are mobile robot. They are capable of moving around in their environment and are not fixed to one physical location. Therefore, navigation is the foundation and highlights of mobile robots. Navigation in robotics is indivisible and essential. Navigation is the movement of a robot to a specific point. A robot needs to use sensors to perceive the environment and build or update its environment map (Domestic Environment - an overview | ScienceDirect Topics, 2017). Apart from that, if the robot's navigation system leads to a poor routes or spends a lot of time to find the route, then achieving the goal will waste a lot of energy and time. Hence, path planning algorithm is a significant issue in robots.

Path planning is a process of searching an optimal path for robot to move from source to destination. From a human point of view, the path planning from location X to location Y, simultaneous obstacles, avoidance and response to environmental changes are simple (Path Planning - an overview | ScienceDirect Topics, 2019). However, if this happens in mobile robots, it becomes challenging. Therefore, various path planning algorithms have been invented today in order to help mobile robots calculate the optimal path. For instance, A* algorithm, Potential field, D*, RRT (Rapidly-exploring Random Tree) and so on. In addition, each of the path planning algorithms has its own applicable domain, performance, advantages and disadvantages in various situations. Therefore, the path planning algorithm of the robot should be wisely selected in order to save the energy and time for the robot to perform certain tasks.

However, if we do not know the properties of the path planning algorithms, then how do we choose the appropriate path planning algorithm? Hence, this paper compared various path planning algorithms. These path planning algorithms included Dijkstra’s, Rapidly-exploring Random Tree (RRT), A*, and the last one is an algorithm modified from A*. Also, the comparisons were made in a simulated robot. This is because various situation can be simply simulated in the simulator. Apart from that, it also saves the cost of the robot and the time to build the real environment.

1.2 Project Scope

The scope of this project is stated as follows:

1. 4 path planning algorithms which are Dijkstra's, A*, RRT and modified A* algorithms are focused in this project.
2. Only using Simple Two Dimensional Robot (STDR) simulator to simulate a robot instead of using a real robot.
3. ROS Robot Programming is used in this project.
4. Python is used in this project.

1.3 Project Objectives

The project objectives are as follows:

- To review existing path planning algorithms.
- To modify an existing path planning algorithm.
- To simulate robot that is implemented by various path planning algorithms.
- To compare various path planning algorithms.

1.4 Contribution

Under different circumstances, different path planning algorithms may be preferred. Through this study, the robot developer will be able to observe and understand the differences in path planning algorithms in simulated robot without wasting a lot of time to understand the properties of various path planning algorithms. This study is expected to provide better evidence for the developer to make an informed choice of path planning algorithm.

1.5 Background information

Robot Operating System (ROS)

ROS is a meta-operating system and it is an open-source (ROS/Introduction - ROS Wiki, 2019). It provides the services containing hardware abstraction, package management, implementation of commonly-used function, message-passing between processes and so on. In addition, it is a robot software platform and it provides a variety of development environments dedicated to developing robot applications.

The ROS has 3 main characteristics. The first characteristic is the reusability of program. Users can focus on the features they want to develop without having to worry about the remaining functions. This is because they can download the corresponding package from ROS. Moreover, they can share their own programs so that others can reuse them. The second characteristic is that ROS is a communication-based program. Each program and feature is programmed in the form of the smallest units of executable processes, and each process runs independently and exchanges data systematically. Hence, this is very useful for finding errors, because programs that are divided into minimum functions can be debugged separately. Thirdly, the ROS supports a variety of development tools. The ROS provides 2D visualization tool, 3D visualization tool, 3D simulator, debugging tool and so on. These software tools necessary for robot development, which take full advantage of the convenience of development.

ROS Master, nodes, publishers, subscriber, and topics are the 5 components of the ROS architecture (Wilcher et al, 2019).

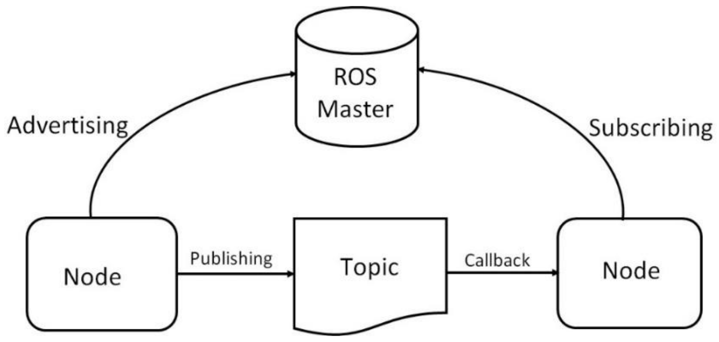


Figure 1.1 shows the system components of a typical ROS model

The job of the ROS Master is to manage the names and registration services of nodes in the ROS system. Apart from that, ROS Master monitors publishers and subscribers to ensure that relevant themes and services are available in the robotic system. Moreover, positioning and communication between nodes in a robotic system is enabled by the ROS Master. In order to initiate the node communication function, the ROS Masters usually use a command called ‘roscore’. Node is an executable file within the ROS system that allows communication among another node. Publisher is a message transmitted by a node or topic in a ROS system. Instead, a message received by node or topic in the ROS system is called subscriber. Topic is the specific name type for the message of publishing and subscribing.

1.6 Report Organization

The report is divided into 7 chapters. The first chapter is about the problem statement, project scope, motivation, project objectives, contributions and background information. The second chapter is the literature review. Four path planning algorithms were reviewed, namely Dijkstra algorithm, A* algorithm, RRT algorithm and D * Lite algorithm. The third chapter is system design. The system design describes in detail the development method of the project. The next chapter is the proposed method / approach. In this chapter, methods, tools, implementation issues and challenges, and timelines will be introduced. Chapter 5 introduces system implementation and testing. In Chapter 6, some experiments will be conducted to compare path planning algorithms. The last chapter is the conclusion of the whole project and some future work of the system.

Chapter 2: Literature Review

2.1 Dijkstra's algorithm

2.1.1 Introduction

This algorithm is a graph search based algorithm. It is used to solve shortest path problem for a directed or undirected graph with nonnegative edge costs, resulting in a shortest path tree. Dutch computer scientist Edsger Dijkstra created this algorithm in 1959 (Venkat, 2014).

The algorithm finds the lowest cost path between the point and each of the other points. It can also be used to search the shortest path from a point to a target point by stopping the algorithm after determining the shortest path to the target point.

2.1.2 How does a Dijkstra's algorithm work?

In the Dijkstra's algorithm, a graph G is considered with *n* nodes via edges *e*. Edge $e_{n1,n2}$ has the cost $c_{n1,n2}$ from node *n1* to node *n2*.

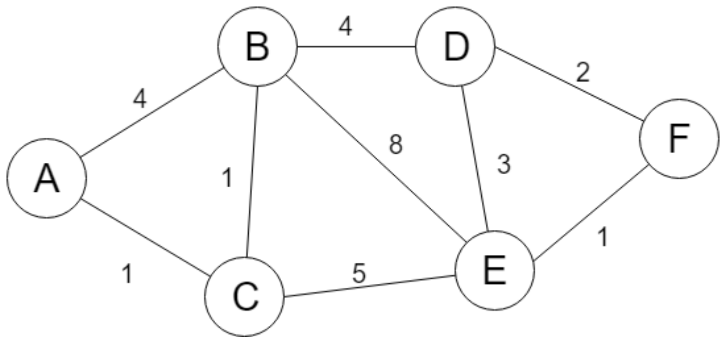


Figure 2.1 shows the weighted graph

First of all, a starting point, *s* is chosen and added in an unvisited list, *U*. The $c_{s,s}$ = 0 because $e_{s,s}$ is not exist. In addition, the other nodes are added to *U* and set the $c_{s,n_{1...k}} = \infty$, where the *k* is the nodes count in the graph. The *U* is a min-priority queue. In other words, the node which has the minimum cost is chosen in the *U*.

Therefore, the *s* is chosen from the *U* since the $c_{s,s}$ is the lowest. After that, the costs from the *s* to its neighbors *x* are updated. The formula for updating the cost for *x* is shown below (Kangutkar, 2017).

$$c_{s,x_{updated}} = min(\, c_{s,x_{current}} , c_{s,x-1} + c_{x-1,x})$$

The $c_{s,x_{current}}$ refers to *x* cost in *U* and the *x-1* is the current node being considered. The selected node is removed from *U* and placed in *V* after updating the costs to each neighbor. The same steps are repeated until the goal node g is added to *V* or if the minimum cost in *U* is ∞.

2.1.3 Advantages and Disadvantages

The advantages of Dijkstra's algorithm are it has an *O*(n^2). Therefore, it is sufficiently effective to use it for a relatively enormous problem. Besides that, it is simple as compare to another path planning algorithm. However, the disadvantages of Dijkstra's algorithm are it performs blind searches. Thus, a lot of time will be wasted to search for useless resource. Also, if Dijkstra's algorithm handles negative weights, it will cause this algorithm to produce incorrect results. (Huang et.al., 2009).

2.2 A* algorithm

2.2.1 Introduction

A* algorithm is a graph search algorithm. It is used to find a path from a starting node to a target node (Gotal et.al. 2014). The algorithm was developed in 1968 by Hart et al. A* combines the information used by the Dijkstra algorithm with the information used by Best-First-Search. Compared to Dijkstra, A* achieves better performance by using heuristic. It is an approximation of distance from the current location to the target location (Dorin, n.d.). In the process of searching for the shortest route, each cell of the grid is evaluated based on an evaluation function given by

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

(n) is the evaluation function, where $h(n)$ is the heuristic cost of the minimum path to reach the goal node g. Besides that, the $g(n)$ refer to the accumulated cost from the starting point s to the current point n. Figure 2.2 shows how the estimate is determined (Choset, n.d.).

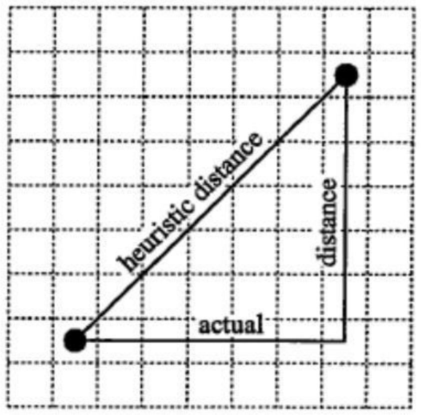


Figure 2.2 shows the A heuristic h(n) (using Euclidean distance)*

According to the grid cell structure shown in Figure 2.3 (Kangutkar, 2017). The diagonal edges have a cost *c* of 1.4, while horizontal and vertical edges have a cost *c* of 1. For instance, $c_{x1,x5}$ = 1.4 and $c_{x1,x2}$ = 1. If x_9 and x_4 are obstacles, then $c_{x1,x9}$ = ∞ and $c_{x1,x4}$ = ∞.

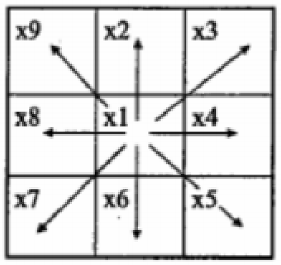


Figure 2.3 shows the possible moves between grid cells

2.2.2 How does an A* work?

This algorithm requires 2 lists to store information about nodes which are open list *O* and closed list *C*. The *O* stores nodes for expansions and the *C* stores nodes that have been explored. Initially, the starting node *s* is added to the *O* for expansions. In *O*, the minimum (n) is chosen which called (n_{best}). n_{best} is a node with the minimum $f(n)$. After that, the n_{best} is remove from *O* and placed into *C*. If the n_{best} is not the goal node g and each neighbor of n_{best} is not included in *C*, then the *x* in *O* will update *g(n)*, and the *x* not in the *O* will be placed into *O*. The same steps are repeated until the n_{best} = g or the *O* is empty.

2.2.3 Advantages and Disadvantages

The A* algorithm is complete and optimal. Apart from that, the time complexity of this algorithm is $O\ (n\ log\ n)$, it can be used to solve very complex problems. However, the accuracy of the heuristic algorithm that is used to compute $h(n)$ has a greatly depends to a large extent on the execution speed of A*.

2.3 D* lite algorithm

2.3.1 Introduction

Sven Koenig and Maxim Likhachev are the persons who designed the D* lite algorithm. This algorithm is an incremental heuristic search algorithm. In the grid, each of the cell, x has a g(u) and rhs(u) value. If g(u) = rhs(u) then, u is said to be consistent, else it is inconsistent. Inconsistent cells are added to a priority queue U. A cell is termed as under-consistent if g(u) < rhs(u) and over-consistent if g(u) > rhs(u). A term inconsistent is used to detect the changes in the graph created by dynamic obstacles, under-consistent means c(current, targeted) increase and over-consistent describes a situation, when c(current, targeted) decrease. The formula of priority, k and rhs(u), is shown below (Kangutkar, 2017).

$$k = \min(g(u), rhs(u) + h(u))$$

$$rhs(u) = min_{n\in\text{Neighbors}}\,(g(n) + c(u, n))$$

The function UPDATE_VERTEX() is called when u is adding to U. If u ∈ U, it is removed from U and if the neighbor is inconsistent, it is added to the queue along with its key k.

2.3.2 How does a D* lite work?

First of all, the rhs(goal) is set to 0 and then added to U. Besides, all g(deg) and rhs(deg) are set to ∞. After initialized, COMPUTE_SHORTEST_PATH() is called. In this function, while $k_{top}$of U is less than the k_{start} or rhs(start)! = g(start) the function runs. The top most element, u is dequeued. If u is over-consistent, g(u) = rhs(u) and the UPDATE_VERTEX() is called for all neighbors. If u is consistent, then g(u) = ∞ and UPDATE_VERTEX() is called on u, itself, as well as its neighbors.

When the path is calculated, the robot move according to the calculated path. If any inconsistency occurs, and it is not possible to travel to cell v, then the edge cost c(u, v) is set to ∞ and UPDATE_VERTEX() is called on u. Likewise, for every member in U, the cost is updated with rhs(u). After that, COMPUTE_SHORTEST_PATH() is called again.

2.3.3 Advantages and Disadvantages

The advantage of D* lite is it is more efficient than other brute-force replanner in expansive and complex environments. However, D* lite algorithm is too complex as compare to other algorithms such as A* algorithm.

2.4 Rapid-Exploring Random Tree (RRT)

2.4.1 Introduction

Rapid-Exploring Random Tree (RRT) was created by LaValle et al. A non-convex high-dimensional spaces is suitable for this algorithm. The main idea is to explore the unexplored part by sampling the points, and gradually "pull" the search tree near to them (LaValle et.al, n.d.).

2.4.2 How does a standard RRT work?

First of all, a starting point, s and a goal point, g attempt to connect directly. If there is no collision, the path is found. Otherwise, a new point needs to be created randomly in the specific area, repeat this step if the new point is in the obstacle. Next, the new point tries to connect with the closest point of the tree. Also, a new point is randomly created if the new point cannot be connected to the point of tree. On the other hand, the new node become the node of the tree if new node is successfully connected. Lastly, check if the node can connect to the target node. If a collision occurs, a new point will be randomly created. If the nodes can connect, it means that the path has been found.

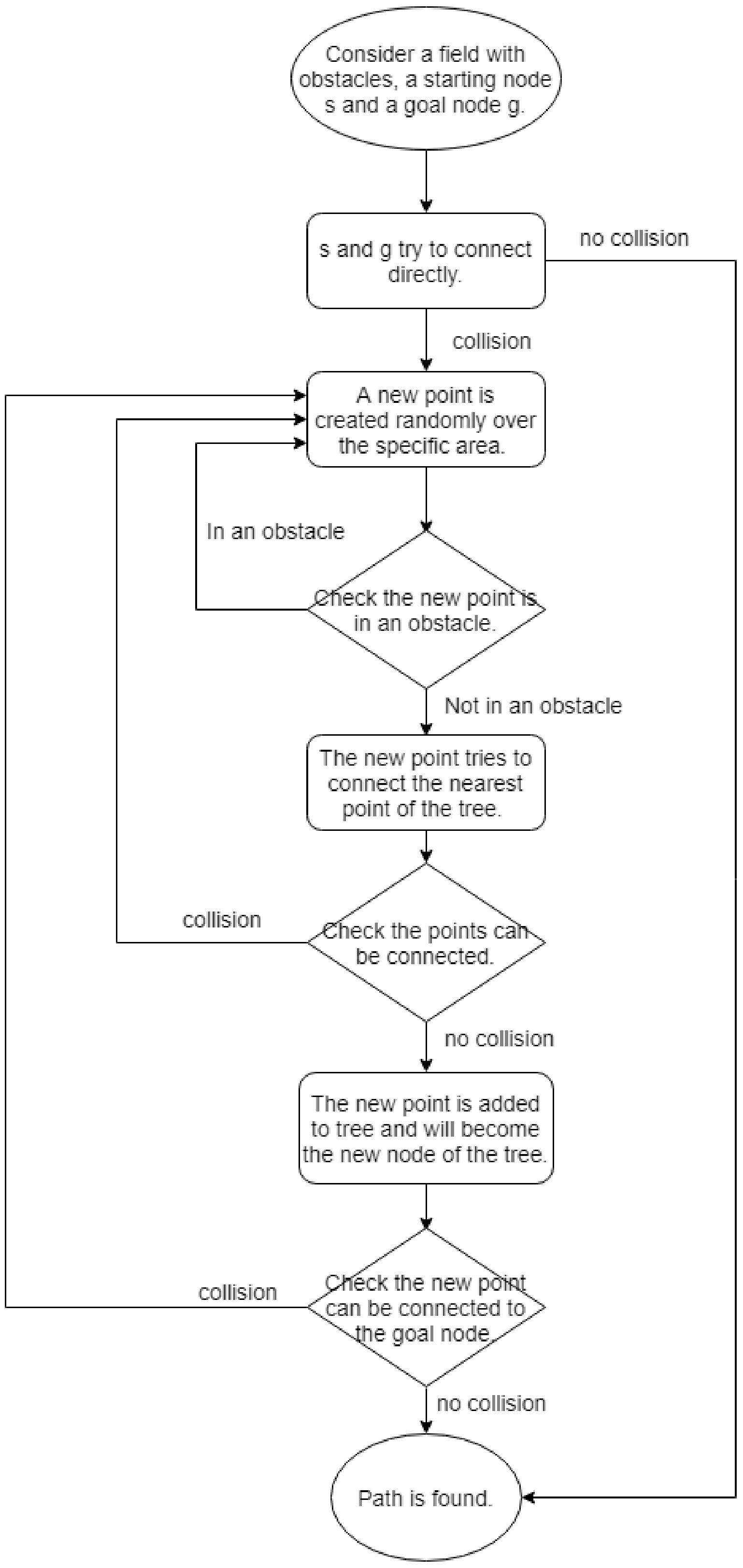

Consider a field with obstacles, a starting node s and a goal node g.
s and g try to connect directly.
no collision
collision
A new point is created randomly over the specific area.
In an obstacle
Check the new point is in an obstacle.
Not in an obstacle
The new point tries to connect the nearest point of the tree.
collision
Check the points can be connected.
no collision
The new point is added to tree and will become the new node of the tree.
collision
Check the new point can be connected to the goal node.
no collision
Path is found.

Figure 2.4 shows the flow chart of the standard RRT

2.4.3 Advantages and Disadvantages

The advantage of the RRT is that the algorithm is very simple to program. Besides that, a tree rapidly explores the entire area, instead of 'staying' in the near the starting node. However, the algorithm is not deterministic. This means that with the same starting point, it will have some different paths. Apart from that, it is difficult to connect a node to a node if there are a lot of obstacle in state space. Moreover, some of the found paths may not be the path chosen by humans.

Chapter 3: System Design

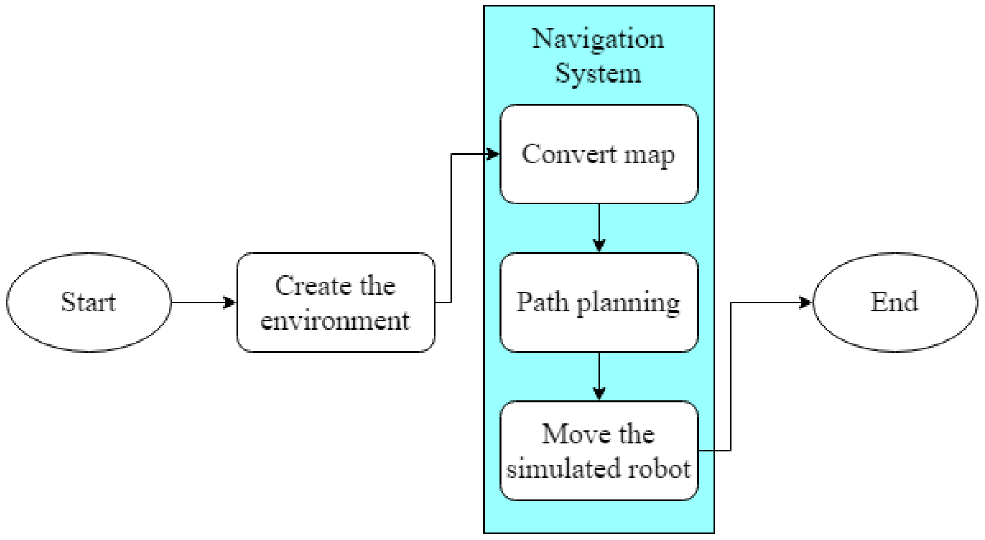


Figure 3.1 shows the system design

3.1 Create the environment

Since the purpose of this project is to analyze and compare path planning algorithms in simulated robot, it is necessary to conduct experiments with a simulated robot and environments. Therefore, a simulated robot and map must be created. In this project, Simple Two Dimensional Robot (STDR) simulator will be used to simulate the movement of the robot in a two-dimensional environment. The STDR simulator already contains some created maps and robots.

3.1.1 Map

The components of the map are image file containing the occupancy data and YAML file. Name of image file and map meta-data need to be included in the YAML file. For example, the Figure 3.2 shows the example of image file and Figure 3.3 shows the format of the map YAML file.

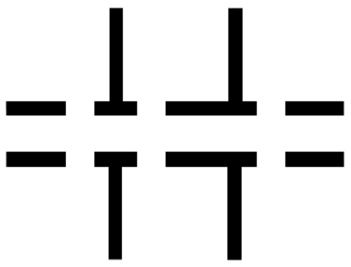
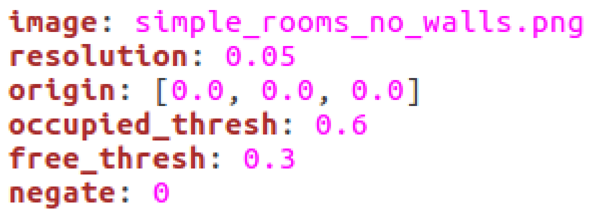


Figure 3.2 shows the simple_rooms_no_walls.png



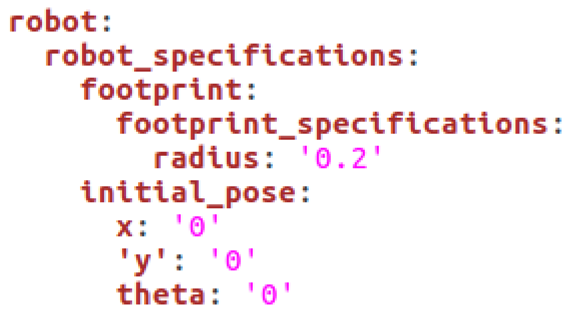
```
image: simple_rooms_no_walls.png
resolution: 0.05
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.6
free_thresh: 0.3
negate: 0
```

Figure 3.3 shows a map's YAML file

In this case, the simple_rooms_no_walls.png is the path to image file. The map resolution is 0.05 (m / pixel). In other words, 1 pixel represents 0.05 meters. According to the origin (x, y, yaw), x-axis equal to 0, y-axis equal to 0, and the yaw as counterclockwise rotation equal to 0. The occupied_thresh of 0.6 means that if the pixel's occupancy probability is greater than 0.6, the pixel is considered to be fully occupied. Conversely, free_thresh of 0.3 means that if the occupancy probability of a pixel is less than 0.3, the pixel is considered completely free. The negate of 0 means that whether the white/black free/occupancy semantics will not be reversed.

3.1.2 Robot

A YAML or XML file needs to be created according to the requirement of the robot.



```
robot:
  robot_specifications:
    footprint:
      footprint_specifications:
        radius: '0.2'
    initial_pose:
      x: '0'
      'y': '0'
      theta: '0'
```

Figure 3.4 shows the YAML file of a simple robot

In this robot’s YAML file, it explains the robot specifications, such as the robot footprint and initial pose. The example shows that the radius of the robot is 0.2m, and the initial posture of the robot is x-axis = 0, y-axis = 0 and theta = 0.

3.1.3 Load the Map and Robot

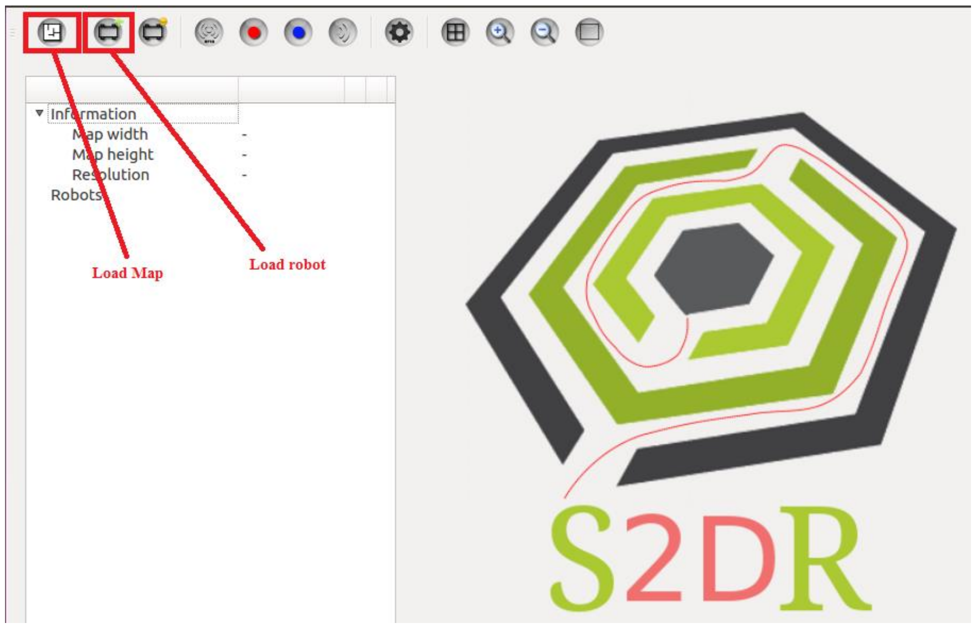


Figure 3.5 shows the “Load Map” and “Load robot” buttons in STDR simulator

Click the "Load Map" button and select the map’s YAML file to load the map. In the same way, click the "Load Robot" button and then select the robot’s YAML file to load the robot.

3.2 Navigation System

In order to move a robot from one point to another point without collision obstacles, the map in PNG format needs to be converted into a map that can be calculated. After that, path planning algorithms are used to calculate the route where the robot can reach the specified point without collision obstacles. After obtaining the path, the robot is moved by using the calculated path.

3.2.1 Convert the map

In the map, white indicates no obstacles, and other colors indicates obstacles. Each pixel of the map contains a decimal code (R, G, B) representing the color of the pixel. For example, (255,0,0) is represent red, (0,255,0) is represent green, and (0,0,255) is represent blue. Therefore, an occupancy map can be constructed by reading each pixel. If the pixel is white (255,255,255), append True to a list otherwise append False. In other words, true means no obstacles, false means obstacles.

3.2.2 Path planning

In this project, Dijkstra's algorithm, A* algorithm, Rapid-exploration Random Tree (RRT) algorithm, and an algorithm modified from A * will be coded. The following sections will show the main functional flowcharts of these algorithms.

3.2.2.1 Dijkstra's and A* algorithms

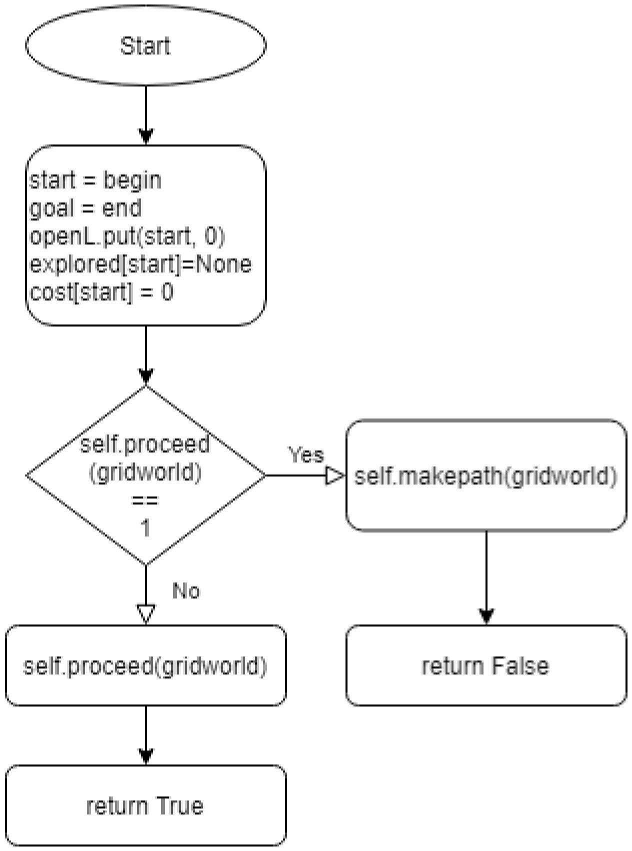


Figure 3.6 shows flowchart of search() function

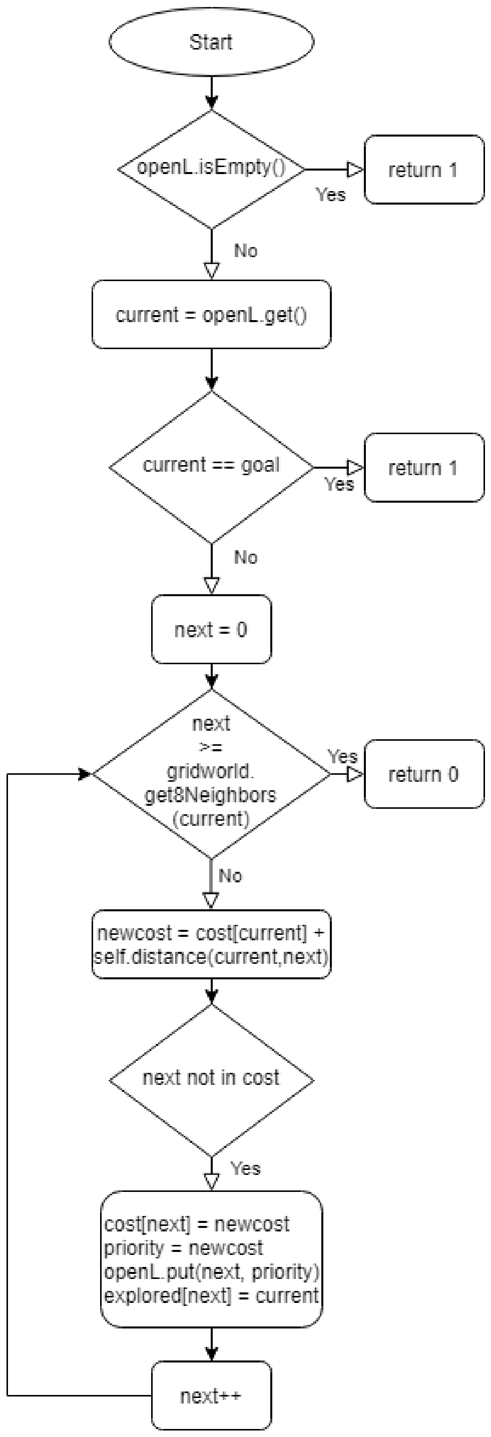


Figure 3.7 shows the flowchart of proceed() function of Dijkstra

Both of the codes are almost the same. The main difference is that in the proceed() function of A*, priority = newcost + self.heuristic(next, goal) instead of just priority = newcost.

3.2.2.2 RRT algorithm

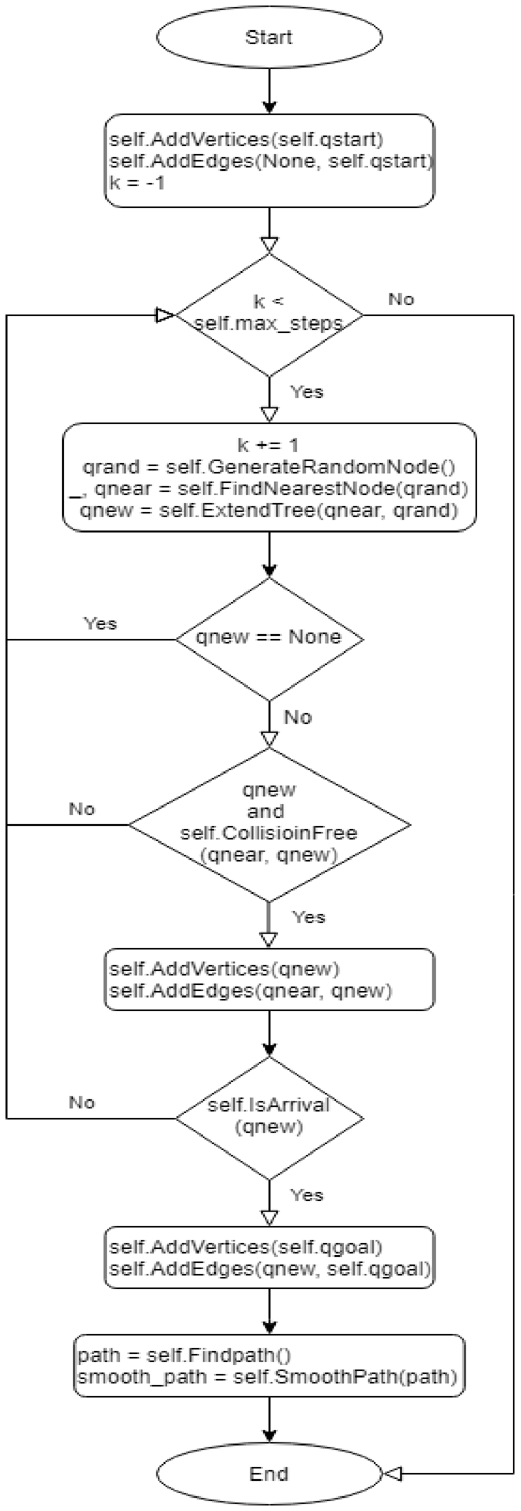

Start
self.AddVertices(self.qstart)
self.AddEdges(None, self.qstart)
k = -1
k < self.max_steps
No
Yes
k += 1
qrand = self.GenerateRandomNode()
_, qnear = self.FindNearestNode(qrand)
qnew = self.ExtendTree(qnear, qrand)
qnew == None
Yes
No
qnew and self.CollisioinFree (qnear, qnew)
No
Yes
self.AddVertices(qnew)
self.AddEdges(qnear, qnew)
self.IsArrival (qnew)
No
Yes
self.AddVertices(self.qgoal)
self.AddEdges(qnew, self.qgoal)
path = self.Findpath()
smooth_path = self.SmoothPath(path)
End

Figure 3.8 shows the flowchart of Planning() of RRT

3.2.2.3 Modified A* algorithm (bidirectional A* search)

The A* algorithm finds the path from the starting point to the goal point. In this modified A* algorithm, it will find the path by looking in both directions. In other words, the algorithm will search from the starting point to the goal point, and from the goal point to the starting point. When one side (forward/backward) of the current neighbor node is on the other side of explored node, the searching process can be stopped. For example, if the current's neighbor node of forward searching meet the explored node of backward searching, the path can be obtained from the start point to the current node of forward searching plus from the explored node of backward searching to goal.

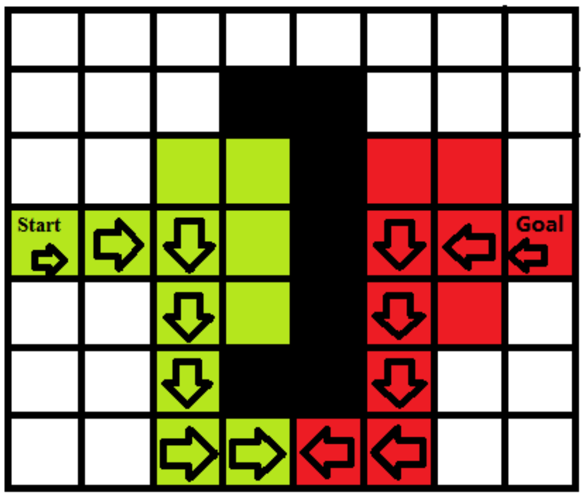


*Figure 3.9 shows the example of bidirectional A**

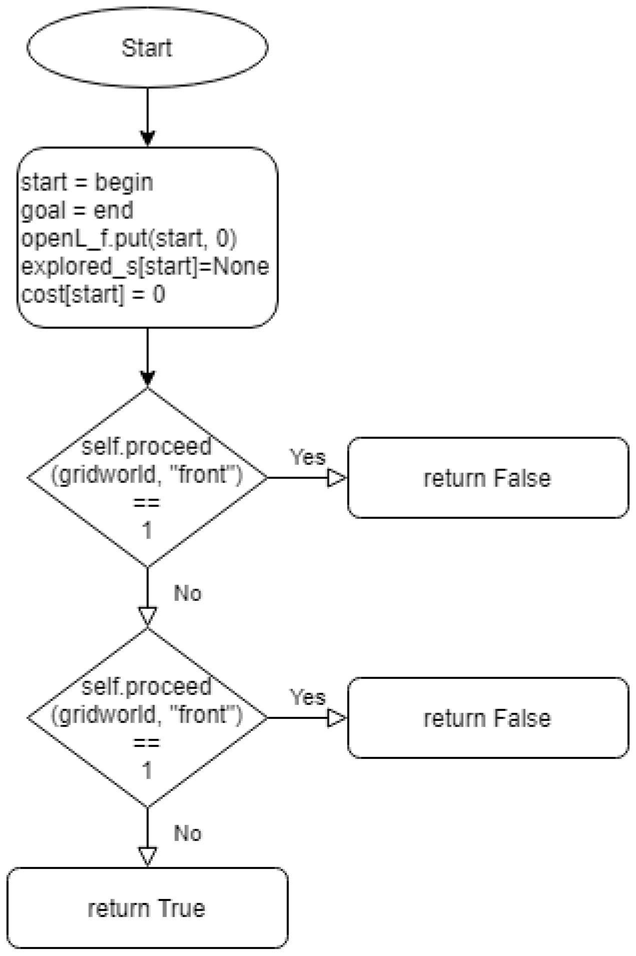


Figure 3.10 shows the flowchart of search_forward()

The search_backward() is almost the same with search_forward(). The difference are:

1. openL_f.put(start,0) change to openL_b.put(goal, 0).
2. explored_s[start] = None –change to explored_g[goal]=None
3. cost[start] = 0 change to cost[goal] = 0
4. self.proceed(gridworld, “front”) change to self.proceed(gridworld, “back”)

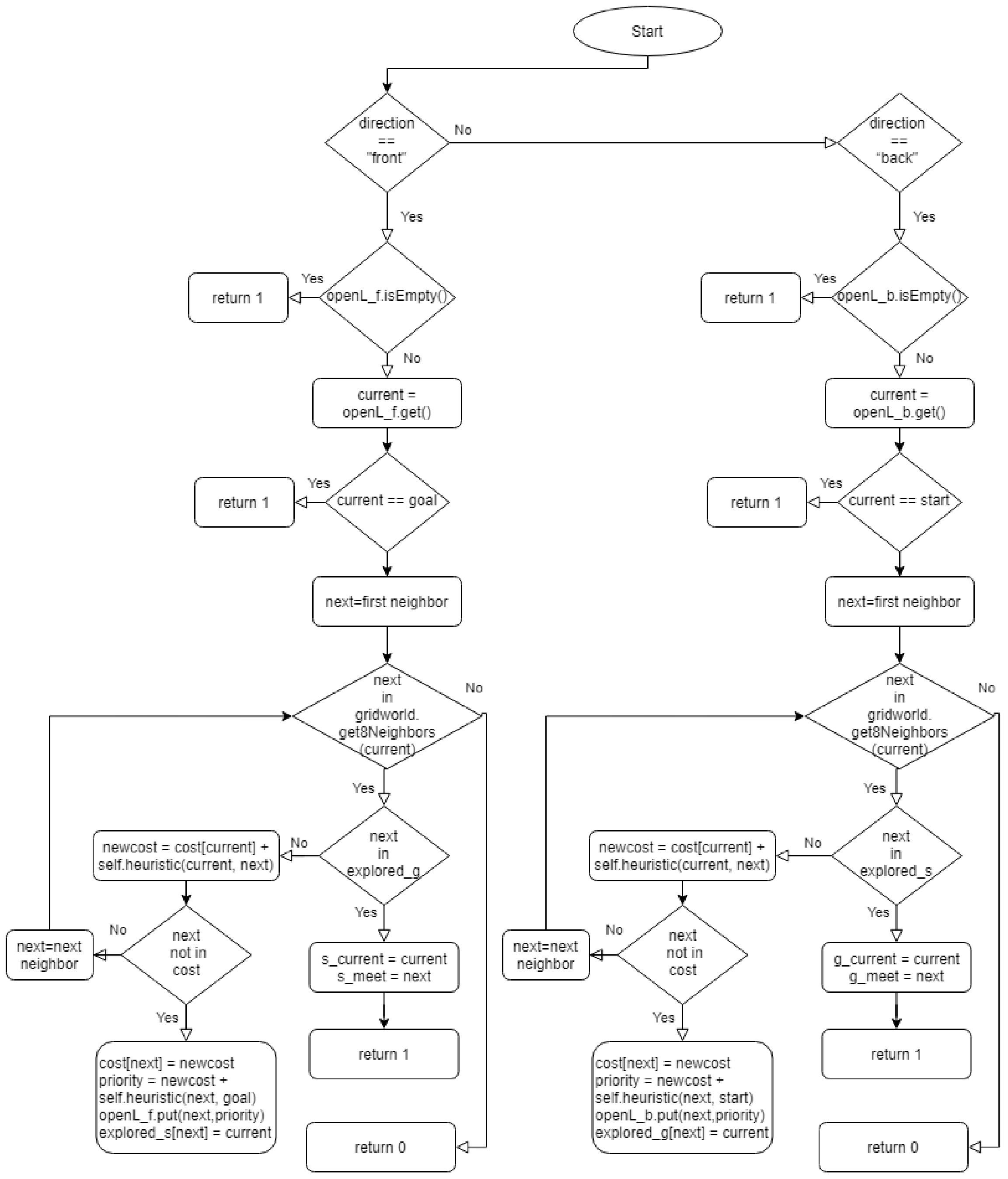
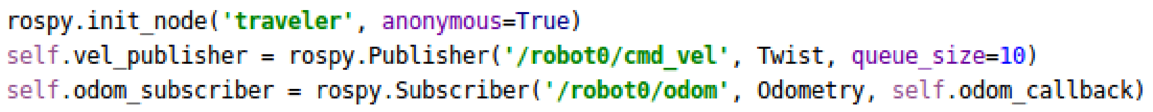

Start
direction == "front"
No
direction == "back"
Yes
return 1
openL_f.isEmpty()
openL_b.isEmpty()
current = openL_f.get()
current = openL_b.get()
current == goal
current == start
next=first neighbor
next in gridworld. get8Neighbors (current)
newcost = cost[current] + self.heuristic(current, next)
next in explored_g
next in explored_s
next=next neighbor
next not in cost
s_current = current s_meet = next
g_current = current g_meet = next
cost[next] = newcost priority = newcost + self.heuristic(next, goal) openL_f.put(next,priority) explored_s[next] = current
cost[next] = newcost priority = newcost + self.heuristic(next, start) openL_b.put(next,priority) explored_g[next] = current
return 0

Figure 3.11 shows the flowchart of proceed()

3.2.3 Move the simulated robot

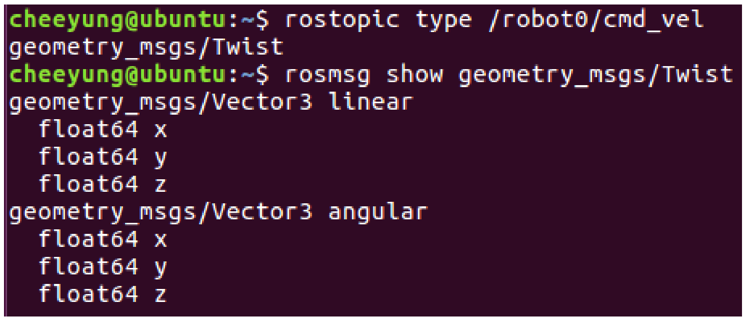
After obtaining the calculated path, the next step is to move the simulated robot. In order to move the simulated robot, a node must be initialize.



```
rospy.init_node('traveler', anonymous=True)
self.vel_publisher = rospy.Publisher('/robot0/cmd_vel', Twist, queue_size=10)
self.odom_subscriber = rospy.Subscriber('/robot0/odom', Odometry, self.odom_callback)
```

Figure 3.12 shows the code to initialize the node in python

A node called ‘traveler’ is created. This node is published to a topic called ‘/robot0/cml_vel’ through the “Twist” message class in order to control the speed and direction of the robot. When the robot publishes a Twist message, it will move according to the message.



```
cheeyung@ubuntu:~$ rostopic type /robot0/cmd_vel
geometry_msgs/Twist
cheeyung@ubuntu:~$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Figure 3.13 shows the details of the message

The rostopic type will return the message type of any topic being published. In this case, “geometry_msgs/Twist” is returned. The message contains "geometry_msgs/Vector3 linear" and "geometry_msgs /Vector3 angular", which are used to control the movement speed and rotation speed, respectively.

In addition, the "traveler" node also subscribes to the "/robot0/odom" topic with the "Odometry" message class to obtain the current location of the robot. Apart from that, ‘self.odom_callback is the callback function for the odemetry subscriber, which will continuously update the current position of the robot.

After the current position and calculated path of the robot are obtained, they can be used to calculate the distance and the rotation angle. The distance can be calculated using Euclidean distance and the rotation angle can be calculated using math.atan2(y,x).

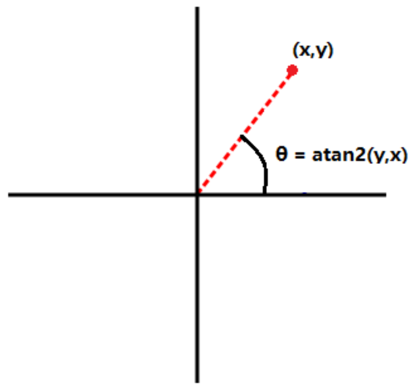


Figure 3.14 shows the example of atan2(y,x)

Therefore, the robot can move to the goal by controlling the rotation angle and linear velocity.

Chapter 4: Proposed method / Approach

4.1 Design Specifications

4.1.1 Methodologies and General Work Procedures

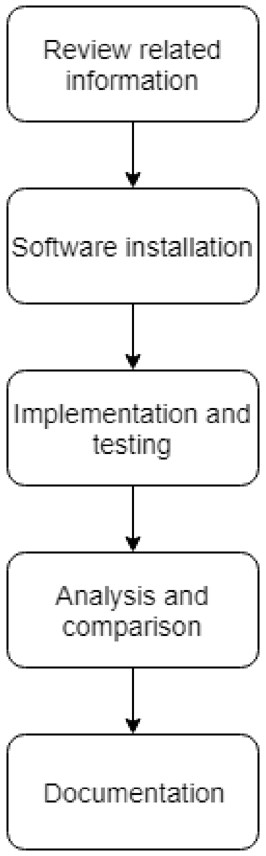


Figure 4.1 shows the project methodology

In the beginning, some relevant information will be studied. In order to simulate robots, the robotic operating system (ROS) plays an important role. ROS has many software libraries and tools to build a robots. Besides, if we want to simulate a robot, then a simulator is needed. Therefore, information about the STDR simulator need to be studied. Apart from that, several existing path planning algorithms also need to be reviewed.

After reviewing the relevant information, we can start to install the software that we need in order to do the implementation. The only platform that ROS currently run is Unix-based platform. Also, most of the ROS software is established on Ubuntu and Max OS X systems. Therefore, VMware Workstation 15 Player and Ubuntu 16.04 Xenial Xerus (LTS) operating system are chosen to be installed. After install the Ubuntu, the ROS development environment need to be configured. Besides, a ROS operation test is configured by using a Turtlesim package provided by ROS.

At the implementation phase, 4 path planning algorithms need to be written which are Dijkstra’s, A*, RRT, and modified A*. After that, those path planning algorithms are used to do a simulation in STDR simulator. If the path planning algorithms in simulated robot can perform correctly, then the result of the simulations will be collected.

Once the results are collected, the results can be analyzed and compared. Finally, the information obtained from the analysis will be used to do a documentation.

4.1.2 Tool to use

Hardware:

Laptop:

Processor	Intel Core i5 @ 2.30GHz
Graphics	2047MB NVIDIA GeForce 930M
Installed RAM	12.0GB
System Type	Windows 10 Home 64-bit

Table 4.1 shows the specification of laptop

Software:

1. VMware Workstation 15 Player:

   It is an ideal utility for running a single virtual machine on a Window or Linux PC. The purpose of installing this software is that an Ubuntu operating system is needed in this project.
2. Simple Two Dimensional Robot (STDR) simulator:

   It is a 2D simulator that can create a robot in a 2D scene and simulate the robot's movement on the computer.

4.2 Implementation issues and challenges

Learning ROS takes a lot of time

ROS is a fairly large system that includes many packages and libraries. These packages are bundled with message-passing framework. In order to understand the concept of ROS, a lot of information need to be studied.

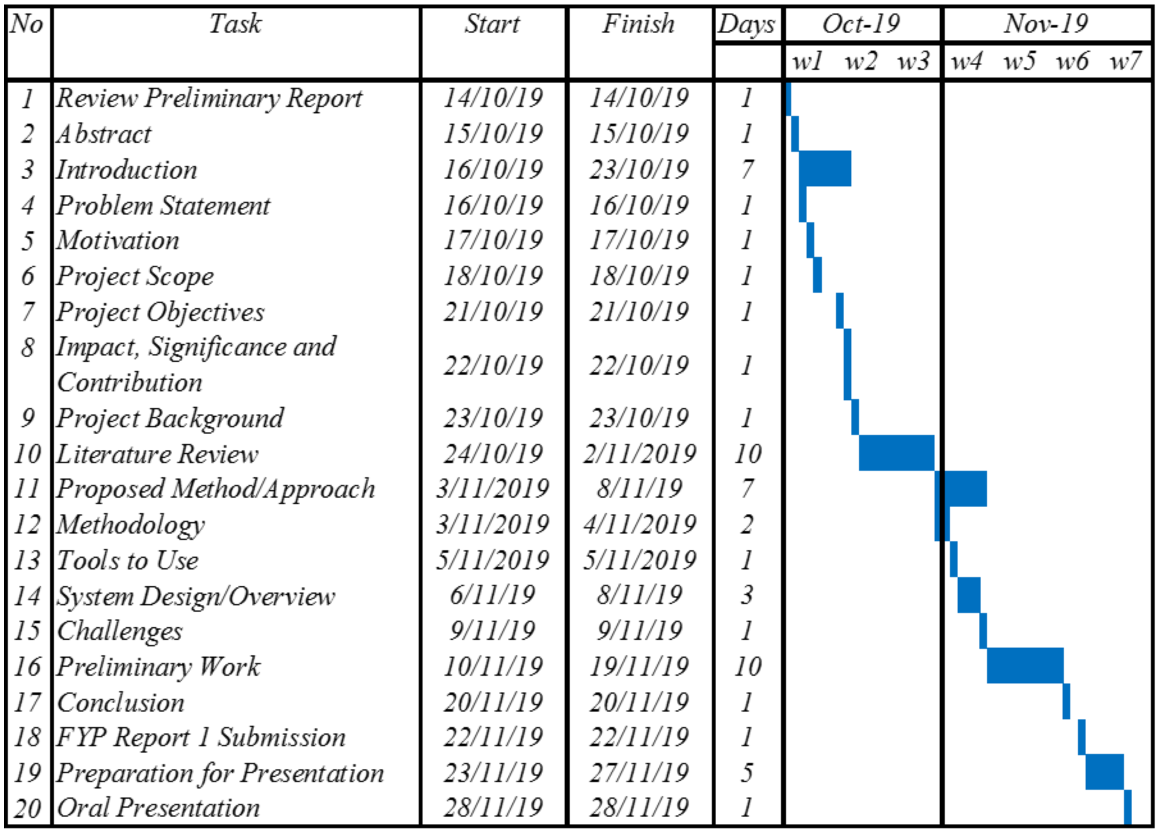
Difficult to set up the software environment

An Ubuntu 16.04 Xenial Xerus (LTS) operating system needs to be installed. This is because most of the ROS software is established on Ubuntu and Max OS X systems. Besides that, the ROS and TurtleBot3 development environment need to be configured by using Linux command. To ensure proper ROS configuration, a Turtlesim package is used to do a ROS operation test. In other words, a lot of time will be wasted in order to build an environment for implementation.

A lot of computing power are needed

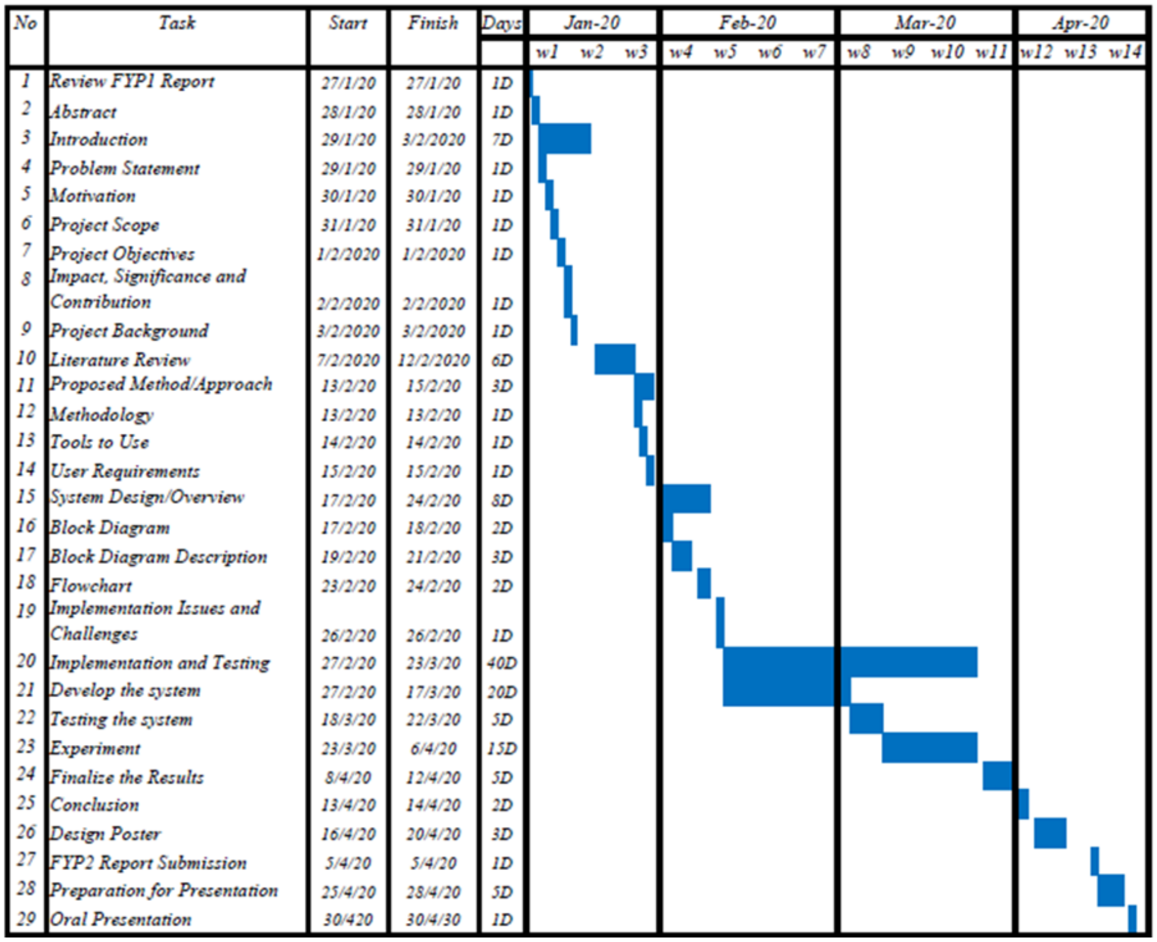
Although the Gazebo is an excellent tool for simulating robots. However, it requires a lot of computing power and may not work well on the laptop. In additionally, the virtual machine is also running. This will lead to an increase in computing power.

4.3 Timeline



No	Task	Start	Finish	Days
1	Review Preliminary Report	14/10/19	14/10/19	1
2	Abstract	15/10/19	15/10/19	1
3	Introduction	16/10/19	23/10/19	7
4	Problem Statement	16/10/19	16/10/19	1
5	Motivation	17/10/19	17/10/19	1
6	Project Scope	18/10/19	18/10/19	1
7	Project Objectives	21/10/19	21/10/19	1
8	Impact, Significance and Contribution	22/10/19	22/10/19	1
9	Project Background	23/10/19	23/10/19	1
10	Literature Review	24/10/19	2/11/2019	10
11	Proposed Method/Approach	3/11/2019	8/11/19	7
12	Methodology	3/11/2019	4/11/2019	2
13	Tools to Use	5/11/2019	5/11/2019	1
14	System Design/Overview	6/11/19	8/11/19	3
15	Challenges	9/11/19	9/11/19	1
16	Preliminary Work	10/11/19	19/11/19	10
17	Conclusion	20/11/19	20/11/19	1
18	FYP Report 1 Submission	22/11/19	22/11/19	1
19	Preparation for Presentation	23/11/19	27/11/19	5
20	Oral Presentation	28/11/19	28/11/19	1

Figure 4.2 shows the Timeline for FYP1



No	Task	Start	Finish	Days
1	Review FYP1 Report	27/1/20	27/1/20	1D
2	Abstract	28/1/20	28/1/20	1D
3	Introduction	29/1/20	3/2/2020	7D
4	Problem Statement	29/1/20	29/1/20	1D
5	Motivation	30/1/20	30/1/20	1D
6	Project Scope	31/1/20	31/1/20	1D
7	Project Objectives	1/2/2020	1/2/2020	1D
8	Impact, Significance and Contribution	2/2/2020	2/2/2020	1D
9	Project Background	3/2/2020	3/2/2020	1D
10	Literature Review	7/2/2020	12/2/2020	6D
11	Proposed Method/Approach	13/2/20	15/2/20	3D
12	Methodology	13/2/20	13/2/20	1D
13	Tools to Use	14/2/20	14/2/20	1D
14	User Requirements	15/2/20	15/2/20	1D
15	System Design/Overview	17/2/20	24/2/20	8D
16	Block Diagram	17/2/20	18/2/20	2D
17	Block Diagram Description	19/2/20	21/2/20	3D
18	Flowchart	23/2/20	24/2/20	2D
19	Implementation Issues and Challenges	26/2/20	26/2/20	1D
20	Implementation and Testing	27/2/20	23/3/20	40D
21	Develop the system	27/2/20	17/3/20	20D
22	Testing the system	18/3/20	22/3/20	5D
23	Experiment	23/3/20	6/4/20	15D
24	Finalize the Results	8/4/20	12/4/20	5D
25	Conclusion	13/4/20	14/4/20	2D
26	Design Poster	16/4/20	20/4/20	3D
27	FYP2 Report Submission	5/4/20	5/4/20	1D
28	Preparation for Presentation	25/4/20	28/4/20	5D
29	Oral Presentation	30/420	30/4/30	1D

Figure 4.3 shows the Timeline for FYP2

Chapter 5: System Implementation and Testing

5.1 Installation and environment setup

In order to run an Ubuntu operating system on Windows, a virtual machine was required. Therefore, VMware Workstation 15 Player was installed. Once the virtual machine was installed, Ubuntu 16.04.6 LTS (Xenial Xerus) was created by using ISO image for Ubuntu in the virtual machine.

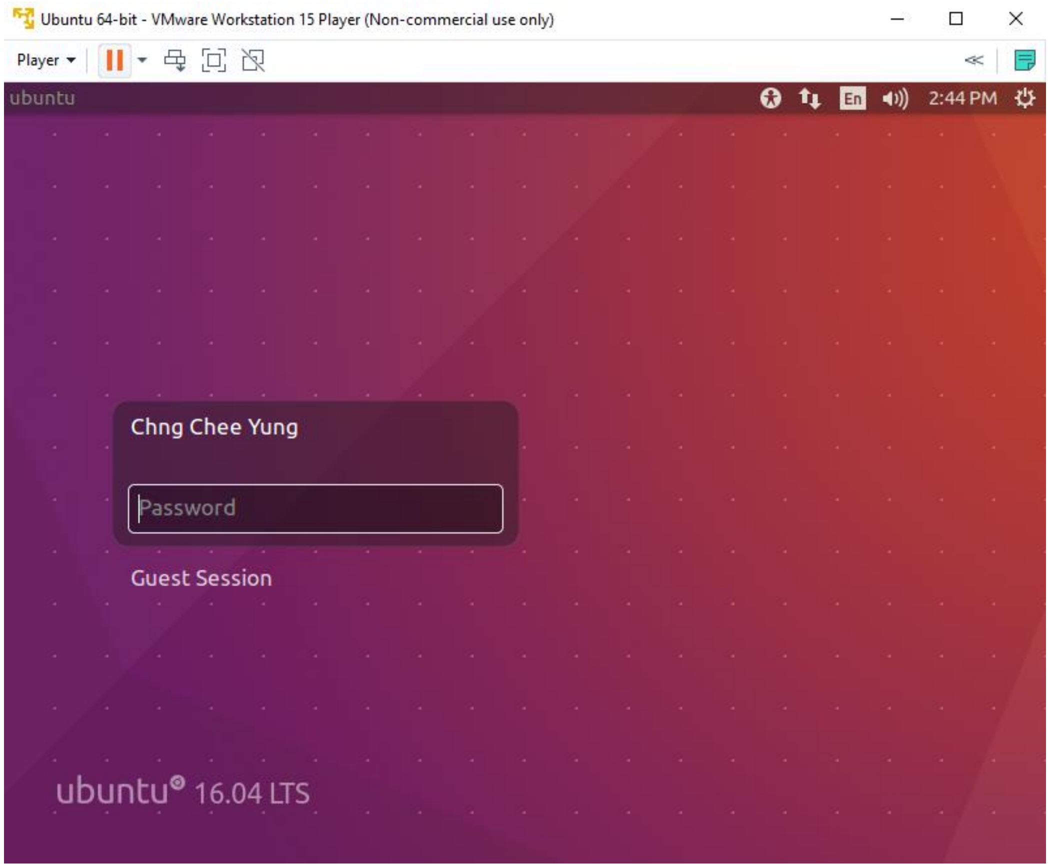
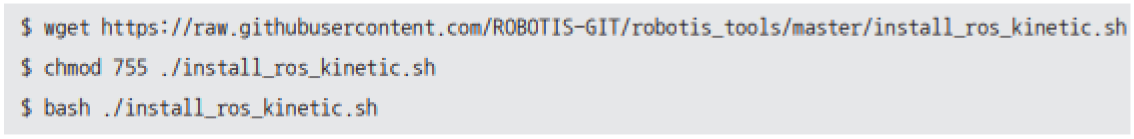


Figure 5.1 shows the Ubuntu opens in the virtual machine

Since the ROS is required in this project, so the Kinetic Kame version had been installed and ROS environment had been configured. In order to save time, a quick installation was used rather than a general installation. However, the quick installation is only available for 16.04.x or Linux Mint 18.x. The quick installation include the Network Time Protocol configuration, adding source list, setting key, updating package index, installing ROS Kinetic Kame, initializing rosdep, installing rosintall, load the environment file, creating and initializing a workspace folder. The command lines of quick installation shown as below.

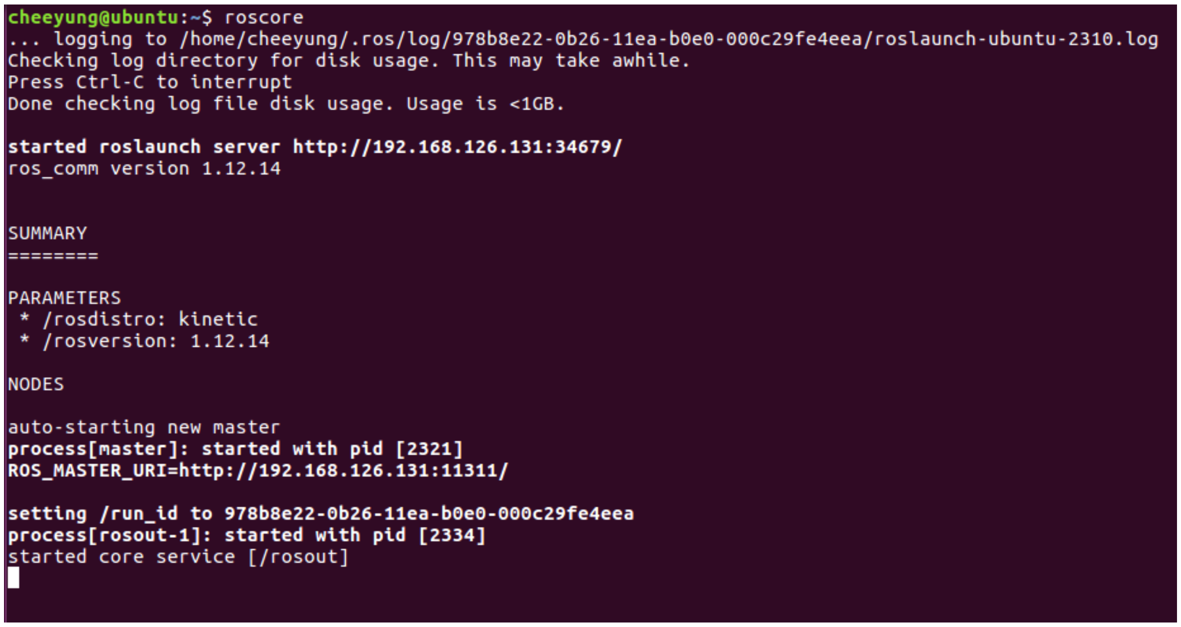


```
$ wget https://raw.githubusercontent.com/ROBOTIS-GIT/robotis_tools/master/install_ros_kinetic.sh
$ chmod 755 ./install_ros_kinetic.sh
$ bash ./install_ros_kinetic.sh
```

Figure 5.2 shows the command lines of quick installation (Pyo, Cho, Jung and Lim, 2017)

5.2 ROS operation test

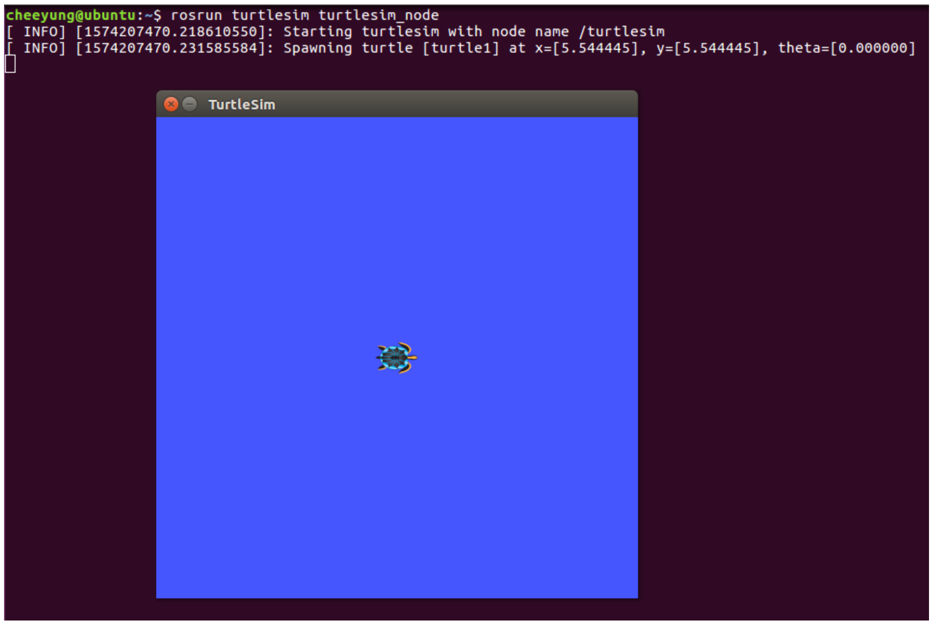
After the installation of ROS, we need to check if it works properly. Thus, a Turtlesim package was used for testing which is provided by ROS. The test is very simple. First of all, a command 'roscore' was entered and run it in a terminal window. This command is used for control the entire ROS system.



```
cheeyung@ubuntu:~$ roscore
... logging to /home/cheeyung/.ros/log/978b8e22-0b26-11ea-b0e0-000c29fe4eea/roslaunch-ubuntu-2310.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.126.131:34679/
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [2321]
ROS_MASTER_URI=http://192.168.126.131:11311/

setting /run_id to 978b8e22-0b26-11ea-b0e0-000c29fe4eea
process[rosout-1]: started with pid [2334]
started core service [/rosout]
```

Figure 5.3 shows the 'roscore' command run successfully

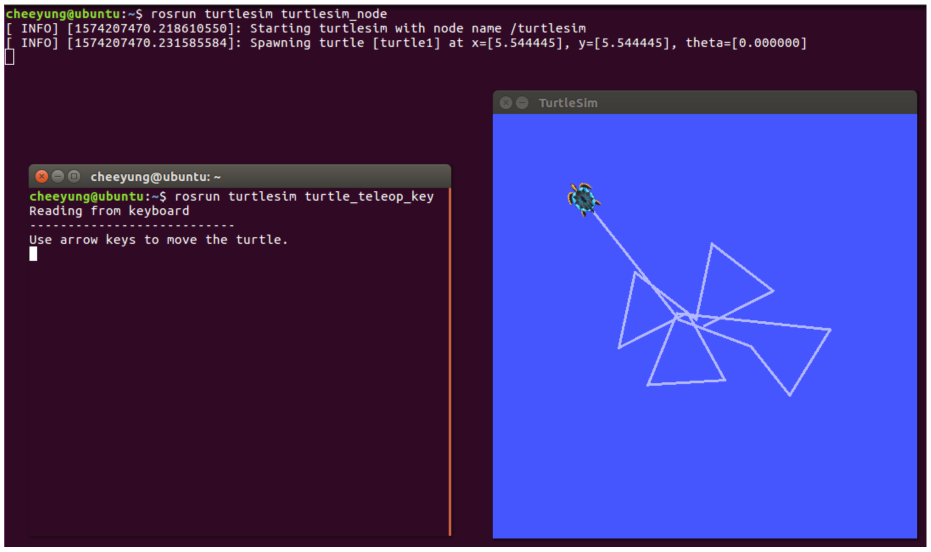
After that, a turtlesim_node need to be created in order to display a turtle on the screen. A new terminal window was opened and enter the command "rosrun turtlesim turtlesim_node". If the node was created successfully, then a turtle will be displayed on a screen.



```
cheeyung@ubuntu:~$ rosrun turtlesim turtlesim_node
[ INFO] [1574207470.218610550]: Starting turtlesim with node name /turtlesim
[ INFO] [1574207470.231585584]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

Figure 5.4 shows the ‘turtlesim_node’ was created successfully

Lastly, a node that controls the turtle also need to be created. The command is “rosrun turtlesim turtle_teleop_key”. As before, a new terminal window was opened and enter this command. If the node was created, the turtle can be controlled by pressing the arrow keys on keyboard in this terminal window.

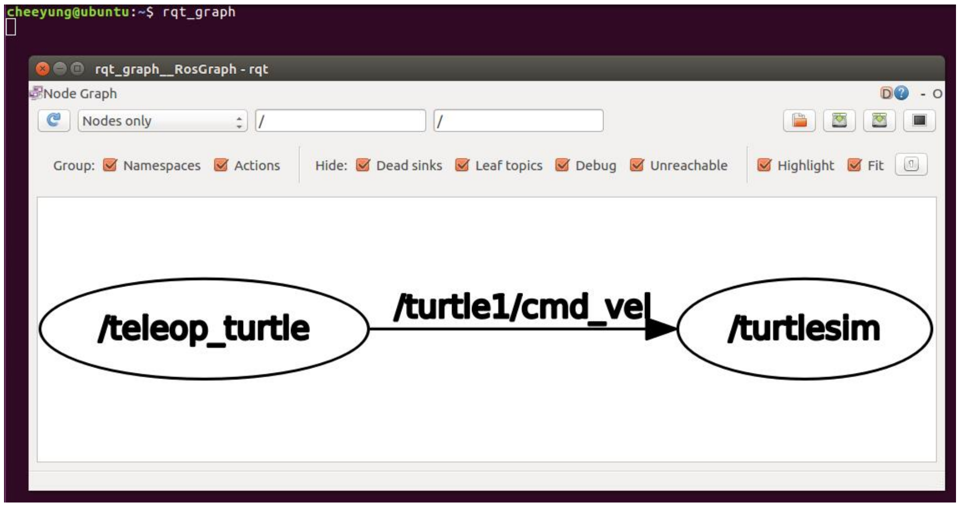


```
cheeyung@ubuntu:~$ rosrun turtlesim turtlesim_node
[ INFO] [1574207470.218610550]: Starting turtlesim with node name /turtlesim
[ INFO] [1574207470.231585584]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

```
cheeyung@ubuntu:~$ rosrun turtlesim turtle_teleop_key
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

Figure 5.5 shows the ‘turtle_teleop_key’ was created successfully

Also, a command ‘rqt_graph’ was used to show a diagram of the information of currently running nodes in a GUI form.



```
cheeyung@ubuntu:~$ rqt_graph
```

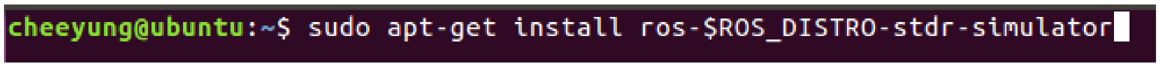
rqt_graph__RosGraph - rqt
Node Graph
Nodes only
/
/
Group: Namespaces Actions
Hide: Dead sinks Leaf topics Debug Unreachable
Highlight Fit
/teleop_turtle
/turtle1/cmd_vel
/turtlesim

Figure 5.6 shows the rqt graph

The circle represents a node. The arrow is point from the ‘/teleop_turtle’ to ‘/turtlesim’ through a topic ‘/turtle1/cmd_vel’. The topic ‘/turtle1/cmd_vel’ which is a sub-topic of the turtle1 topic. When ‘/teleop_turtle’ get a speed input, it sent the input as a message in the topic to the ‘/turtlesim’. Therefore, ‘/turtlesim’ can use the message in the topic as input to visualize the speed command.

5.3 Install STDR simulator

In order to simulate the path planning and movement of the robot, the STDR simulator will be used.



```
cheeyung@ubuntu:~$ sudo apt-get install ros-$ROS_DISTRO-stdr-simulator
```

Figure 5.7 shows the command line for installing the STDR simulator

If running on Ubuntu Linux or Linux Mint, this command line can be used to install the STDR simulator.

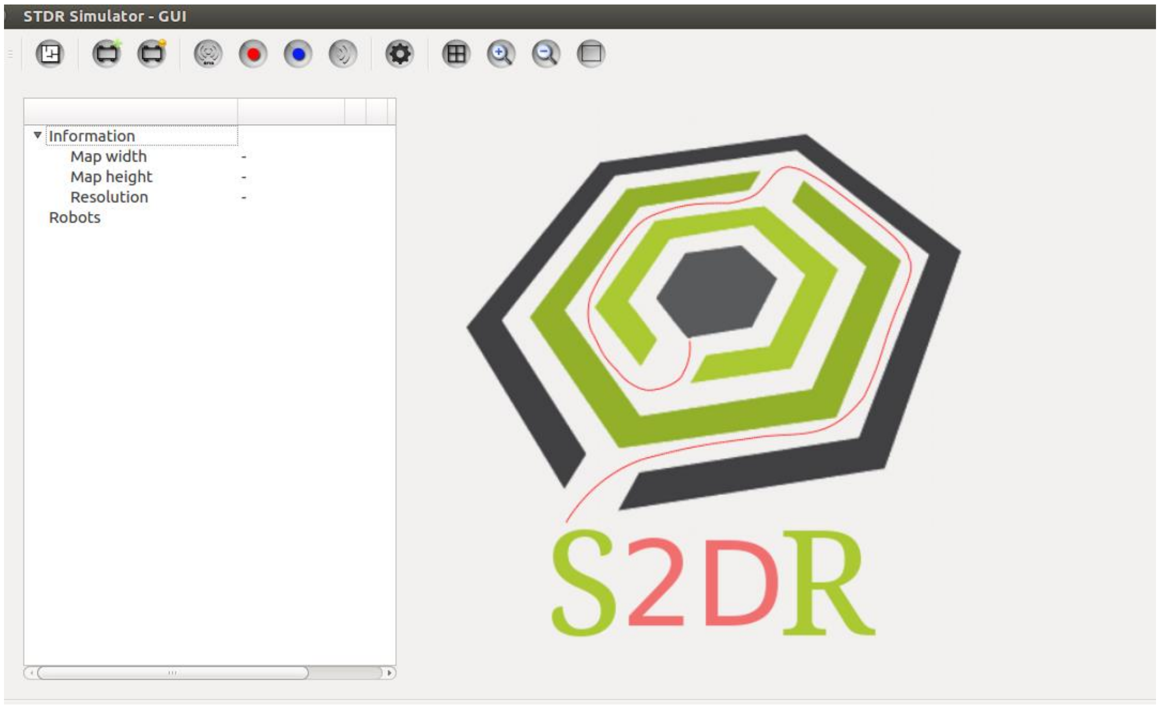


Figure 5.8 shows the STDR simulator

5.4 Create a launch file to load map and robot into STDR simulator

For convenience, a launch file "server_with_map_and_gui_plus_robot.launch" was written in order to load the map and robot at the same time instead of loading the map and the robot one by one.

```
<launch>

        <include file="$(find stdr_robot)/launch/robot_manager.launch" />

        <node type="stdr_server_node" pkg="stdr_server" name="stdr_server"
output="screen" args="$(find stdr_resources)/maps/simple_rooms_no_walls.yaml"/>

        <include file="$(find stdr_gui)/launch/stdr_gui.launch"/>

        <node pkg="stdr_robot" type="robot_handler" name="$(anon robot_spawn)"
args="add $(find stdr_resources)/resources/robots/simple_robot.yaml 2 2 0" />


</launch>
```

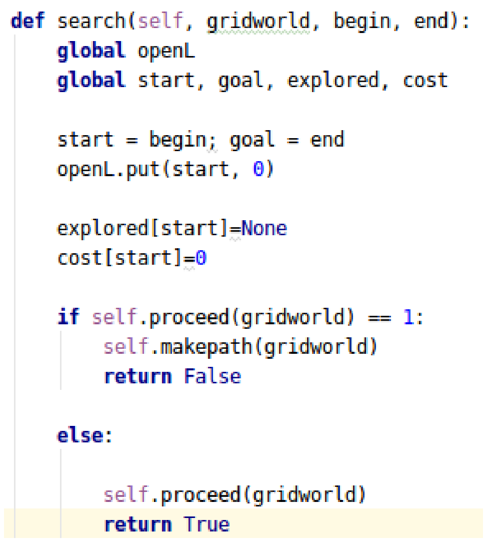
Figure 5.9 shows the launch file that loads map and robot into the STDR simulator

"robot_manager" launch file is included in order to listen to the ROS services. The node "stdr_server_node" in "stdr_server" package is called and the argument is the path of the map's YAML file. After that, stdr_gui launch file is included in order to open the STDR simulator. After opening the simulator, the node “robot_handler” in “stdr_robot” package is called with the argument “add $(find stdr_resources) /resources /robots /simple_robot.yaml 2 2 0”. This means that a robot described in simple_robot.yaml will be added, where x = 2, y = 2, and theta = 0.

5.5 Write path planning algorithms and code to control robot motion

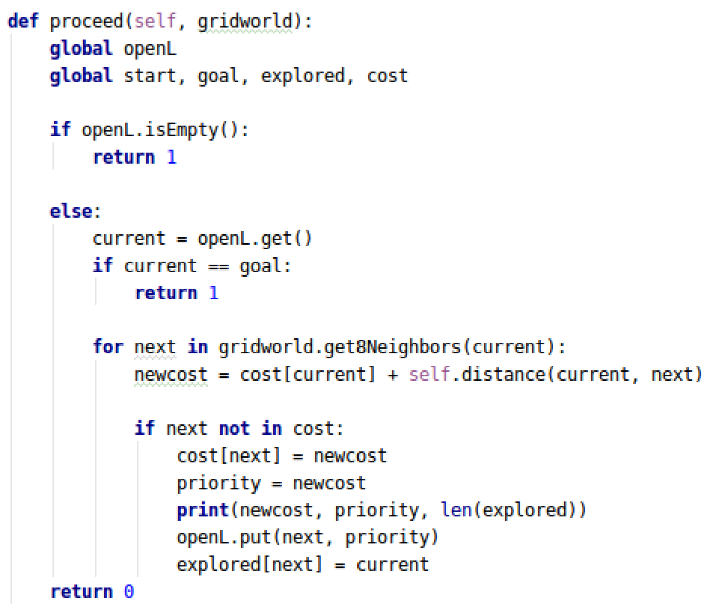
The following sections will show the main function codes of those algorithms and “Safegoto.py”.

5.5.1 Dijkstra’s algorithm



```
def search(self, gridworld, begin, end):
    global openL
    global start, goal, explored, cost

    start = begin; goal = end
    openL.put(start, 0)

    explored[start]=None
    cost[start]=0

    if self.proceed(gridworld) == 1:
        self.makepath(gridworld)
        return False

    else:

        self.proceed(gridworld)
        return True
```

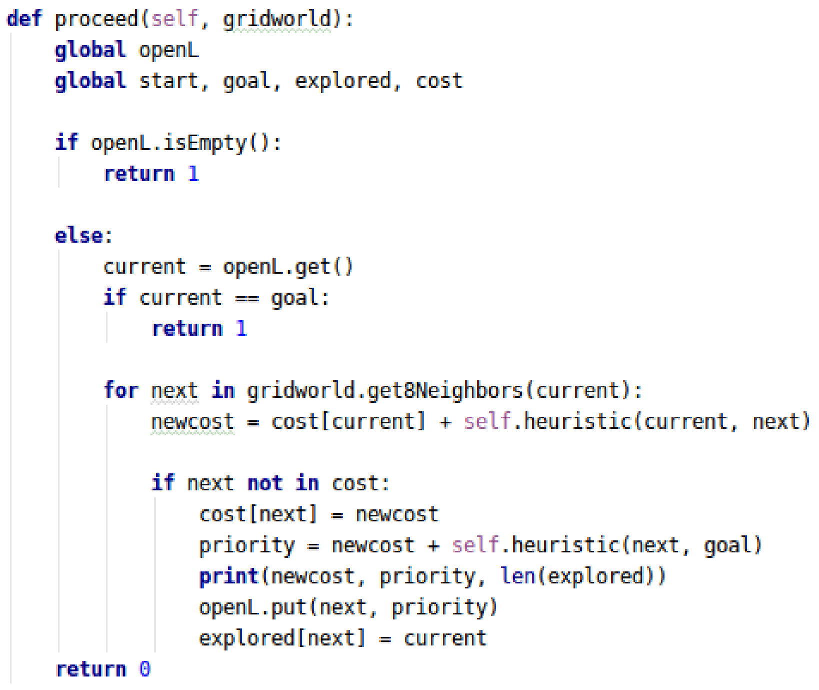
Figure 5.10 shows the search function of Dijkstra



```
def proceed(self, gridworld):
    global openL
    global start, goal, explored, cost

    if openL.isEmpty():
        return 1

    else:
        current = openL.get()
        if current == goal:
            return 1

        for next in gridworld.get8Neighbors(current):
            newcost = cost[current] + self.distance(current, next)

            if next not in cost:
                cost[next] = newcost
                priority = newcost
                print(newcost, priority, len(explored))
                openL.put(next, priority)
                explored[next] = current
    return 0
```

Figure 5.11 shows the proceed function of Dijkstra

5.5.1 A* algorithm

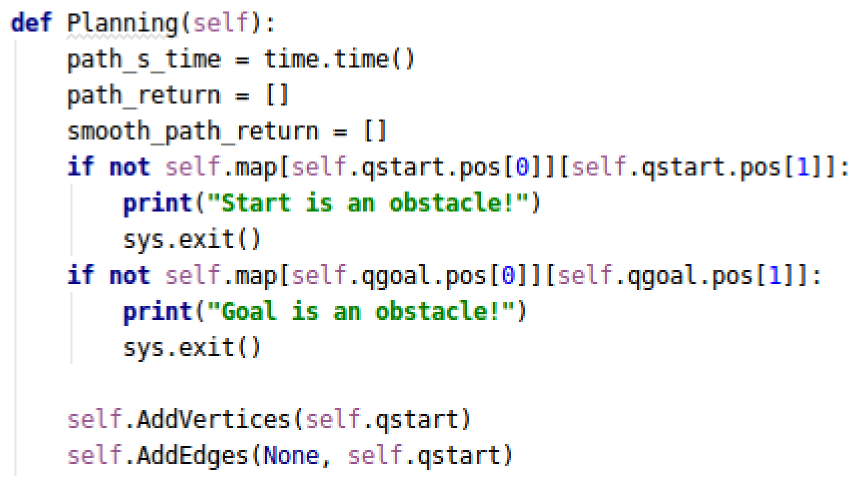


```
def proceed(self, gridworld):
    global openL
    global start, goal, explored, cost

    if openL.isEmpty():
        return 1

    else:
        current = openL.get()
        if current == goal:
            return 1

        for next in gridworld.get8Neighbors(current):
            newcost = cost[current] + self.heuristic(current, next)

            if next not in cost:
                cost[next] = newcost
                priority = newcost + self.heuristic(next, goal)
                print(newcost, priority, len(explored))
                openL.put(next, priority)
                explored[next] = current
    return 0
```

*Figure 5.12 shows the proceed function of A**

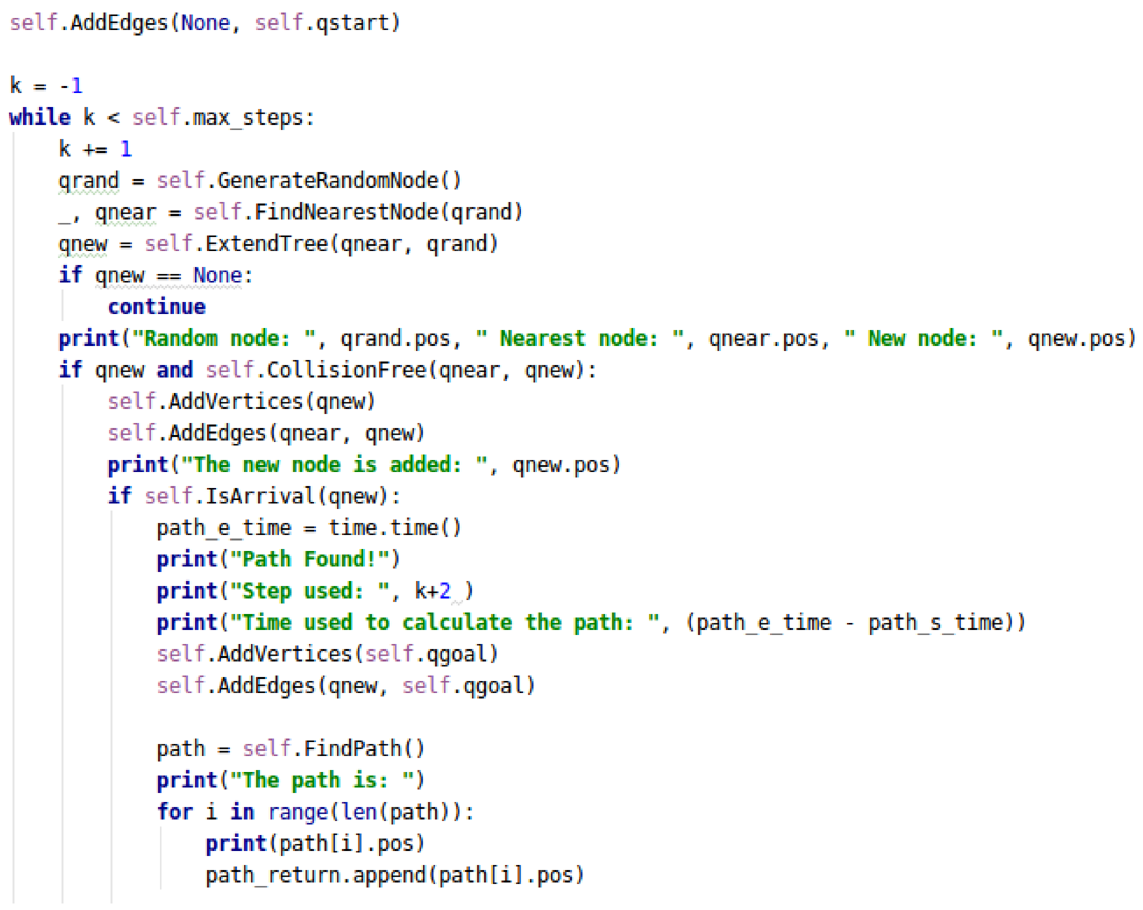
The priority of A* adds a heuristic function.

5.5.3 RRT algorithm



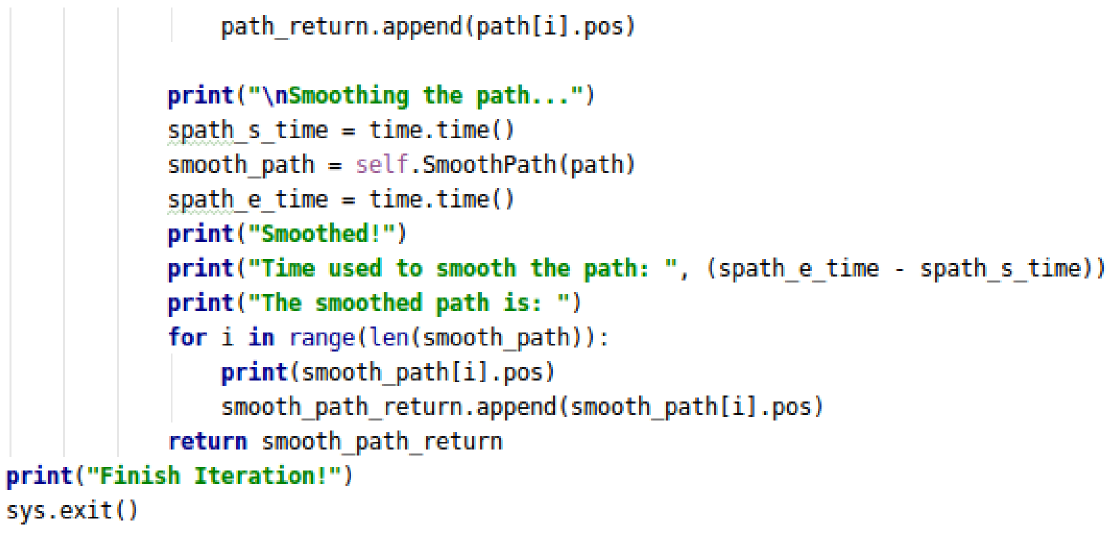
```
def Planning(self):
    path_s_time = time.time()
    path_return = []
    smooth_path_return = []
    if not self.map[self.qstart.pos[0]][self.qstart.pos[1]]:
        print("Start is an obstacle!")
        sys.exit()
    if not self.map[self.qgoal.pos[0]][self.qgoal.pos[1]]:
        print("Goal is an obstacle!")
        sys.exit()

    self.AddVertices(self.qstart)
    self.AddEdges(None, self.qstart)
```

Figure 5.13 shows the Planning function of RRT (1)



```
self.AddEdges(None, self.qstart)

k = -1
while k < self.max_steps:
    k += 1
    qrand = self.GenerateRandomNode()
    _, qnear = self.FindNearestNode(qrand)
    qnew = self.ExtendTree(qnear, qrand)
    if qnew == None:
        continue
    print("Random node: ", qrand.pos, " Nearest node: ", qnear.pos, " New node: ", qnew.pos)
    if qnew and self.CollisionFree(qnear, qnew):
        self.AddVertices(qnew)
        self.AddEdges(qnear, qnew)
        print("The new node is added: ", qnew.pos)
        if self.IsArrival(qnew):
            path_e_time = time.time()
            print("Path Found!")
            print("Step used: ", k+2 )
            print("Time used to calculate the path: ", (path_e_time - path_s_time))
            self.AddVertices(self.qgoal)
            self.AddEdges(qnew, self.qgoal)

            path = self.FindPath()
            print("The path is: ")
            for i in range(len(path)):
                print(path[i].pos)
                path_return.append(path[i].pos)
```

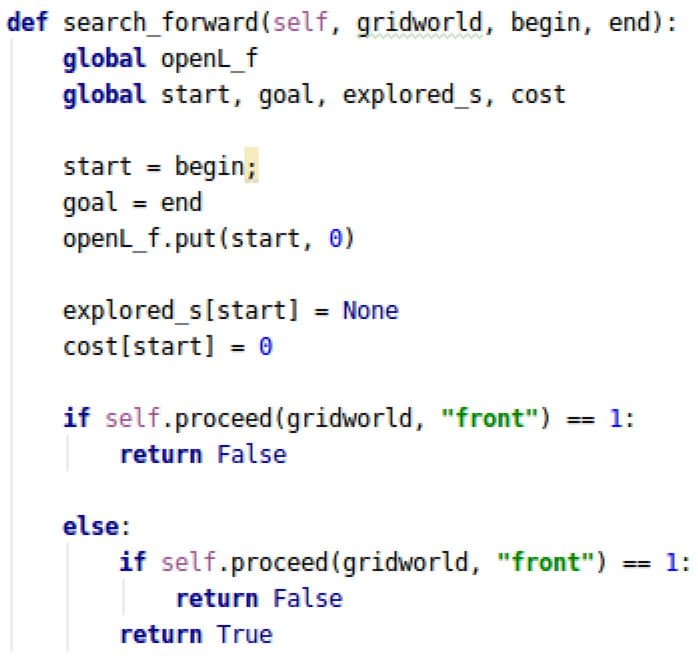
Figure 5.14 shows the Planning function of RRT (2)



```
                path_return.append(path[i].pos)

            print("\nSmoothing the path...")
            spath_s_time = time.time()
            smooth_path = self.SmoothPath(path)
            spath_e_time = time.time()
            print("Smoothed!")
            print("Time used to smooth the path: ", (spath_e_time - spath_s_time))
            print("The smoothed path is: ")
            for i in range(len(smooth_path)):
                print(smooth_path[i].pos)
                smooth_path_return.append(smooth_path[i].pos)
            return smooth_path_return
print("Finish Iteration!")
sys.exit()
```

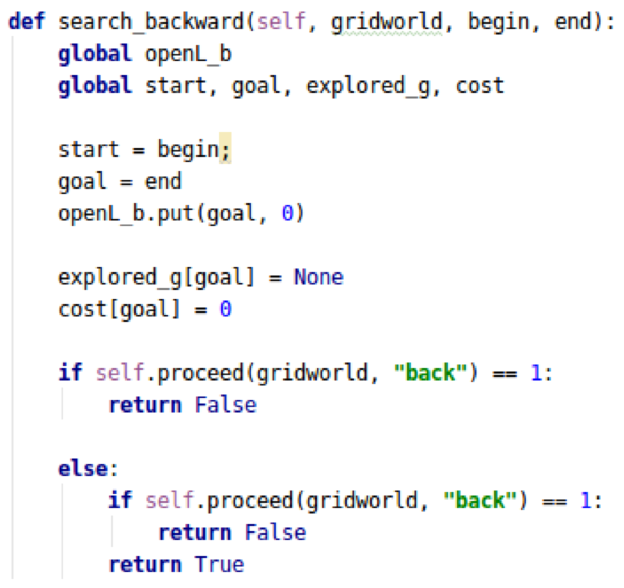
Figure 5.15 shows the Planning function of RRT (3)

5.5.3 Modified A* algorithm



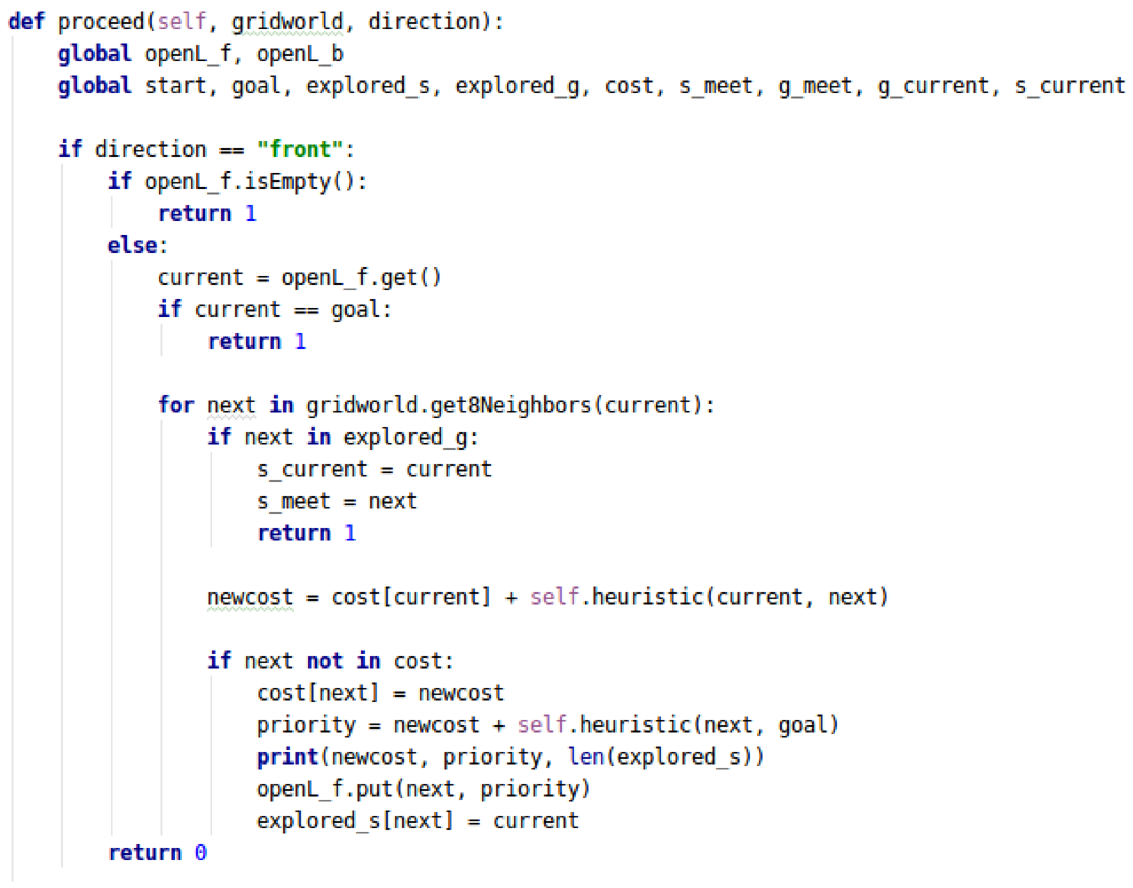
```
def search_forward(self, gridworld, begin, end):
    global openL_f
    global start, goal, explored_s, cost

    start = begin;
    goal = end
    openL_f.put(start, 0)

    explored_s[start] = None
    cost[start] = 0

    if self.proceed(gridworld, "front") == 1:
        return False

    else:
        if self.proceed(gridworld, "front") == 1:
            return False
        return True
```

*Figure 5.16 shows the search_forward of modified A**



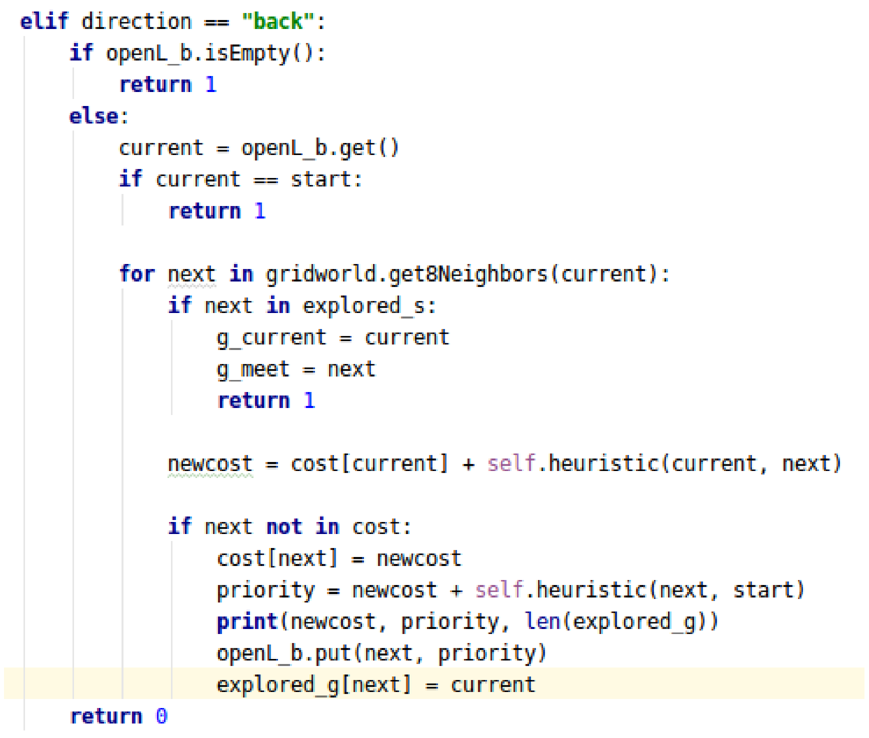
```
def search_backward(self, gridworld, begin, end):
    global openL_b
    global start, goal, explored_g, cost

    start = begin;
    goal = end
    openL_b.put(goal, 0)

    explored_g[goal] = None
    cost[goal] = 0

    if self.proceed(gridworld, "back") == 1:
        return False

    else:
        if self.proceed(gridworld, "back") == 1:
            return False
        return True
```

*Figure 5.17 shows the search_backward function of modified A**



```
def proceed(self, gridworld, direction):
    global openL_f, openL_b
    global start, goal, explored_s, explored_g, cost, s_meet, g_meet, g_current, s_current

    if direction == "front":
        if openL_f.isEmpty():
            return 1
        else:
            current = openL_f.get()
            if current == goal:
                return 1

            for next in gridworld.get8Neighbors(current):
                if next in explored_g:
                    s_current = current
                    s_meet = next
                    return 1

                newcost = cost[current] + self.heuristic(current, next)

                if next not in cost:
                    cost[next] = newcost
                    priority = newcost + self.heuristic(next, goal)
                    print(newcost, priority, len(explored_s))
                    openL_f.put(next, priority)
                    explored_s[next] = current
        return 0
```

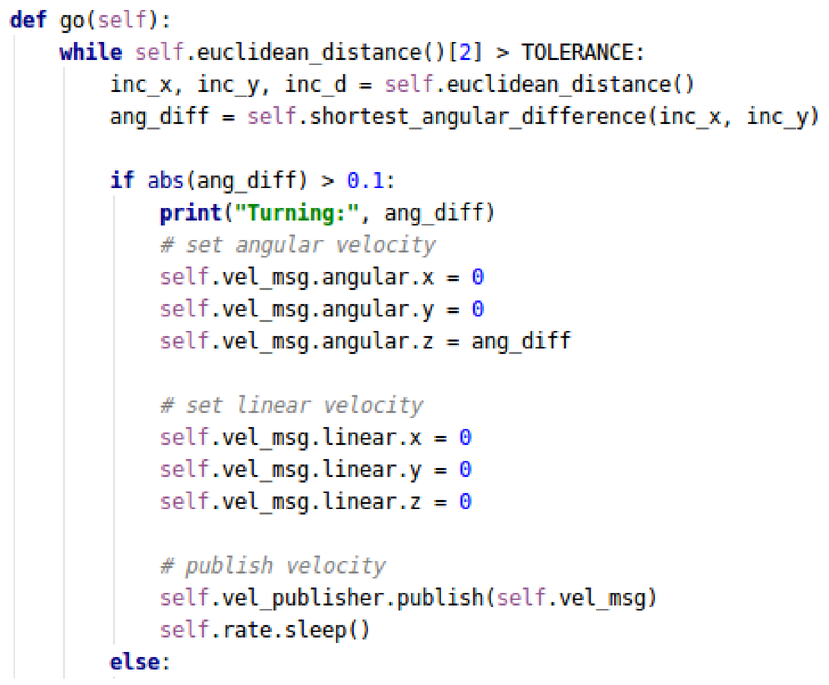
Figure 5.18 shows the proceed function of modified A (1)*



```
elif direction == "back":
    if openL_b.isEmpty():
        return 1
    else:
        current = openL_b.get()
        if current == start:
            return 1

        for next in gridworld.get8Neighbors(current):
            if next in explored_s:
                g_current = current
                g_meet = next
                return 1

            newcost = cost[current] + self.heuristic(current, next)

            if next not in cost:
                cost[next] = newcost
                priority = newcost + self.heuristic(next, start)
                print(newcost, priority, len(explored_g))
                openL_b.put(next, priority)
                explored_g[next] = current
    return 0
```

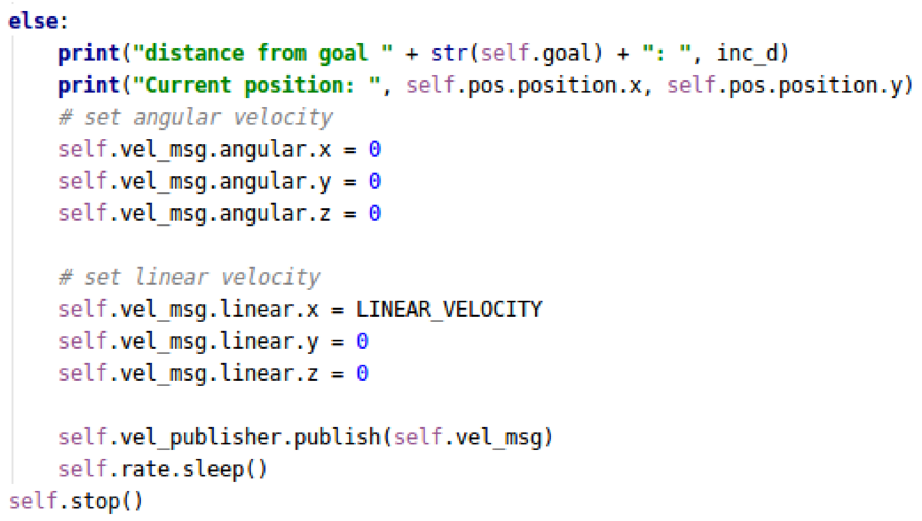
Figure 5.19 shows the proceed function of modified A (2)*

5.5.4 Write a code to control robot motion (Safegoto.py)



```
def go(self):
    while self.euclidean_distance()[2] > TOLERANCE:
        inc_x, inc_y, inc_d = self.euclidean_distance()
        ang_diff = self.shortest_angular_difference(inc_x, inc_y)

        if abs(ang_diff) > 0.1:
            print("Turning:", ang_diff)
            # set angular velocity
            self.vel_msg.angular.x = 0
            self.vel_msg.angular.y = 0
            self.vel_msg.angular.z = ang_diff

            # set linear velocity
            self.vel_msg.linear.x = 0
            self.vel_msg.linear.y = 0
            self.vel_msg.linear.z = 0

            # publish velocity
            self.vel_publisher.publish(self.vel_msg)
            self.rate.sleep()
        else:
```

Figure 5.20 shows the go function for control robot motion (1)

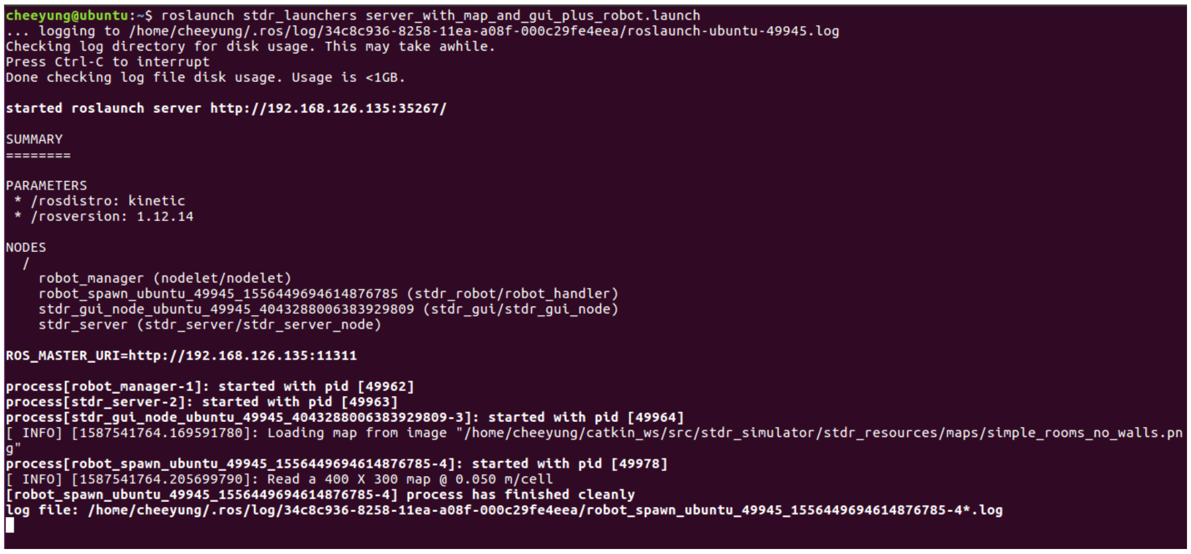


```
else:
    print("distance from goal " + str(self.goal) + ": ", inc_d)
    print("Current position: ", self.pos.position.x, self.pos.position.y)
    # set angular velocity
    self.vel_msg.angular.x = 0
    self.vel_msg.angular.y = 0
    self.vel_msg.angular.z = 0

    # set linear velocity
    self.vel_msg.linear.x = LINEAR_VELOCITY
    self.vel_msg.linear.y = 0
    self.vel_msg.linear.z = 0

    self.vel_publisher.publish(self.vel_msg)
    self.rate.sleep()
self.stop()
```

Figure 5.21 shows the go function for control robot motion (2)

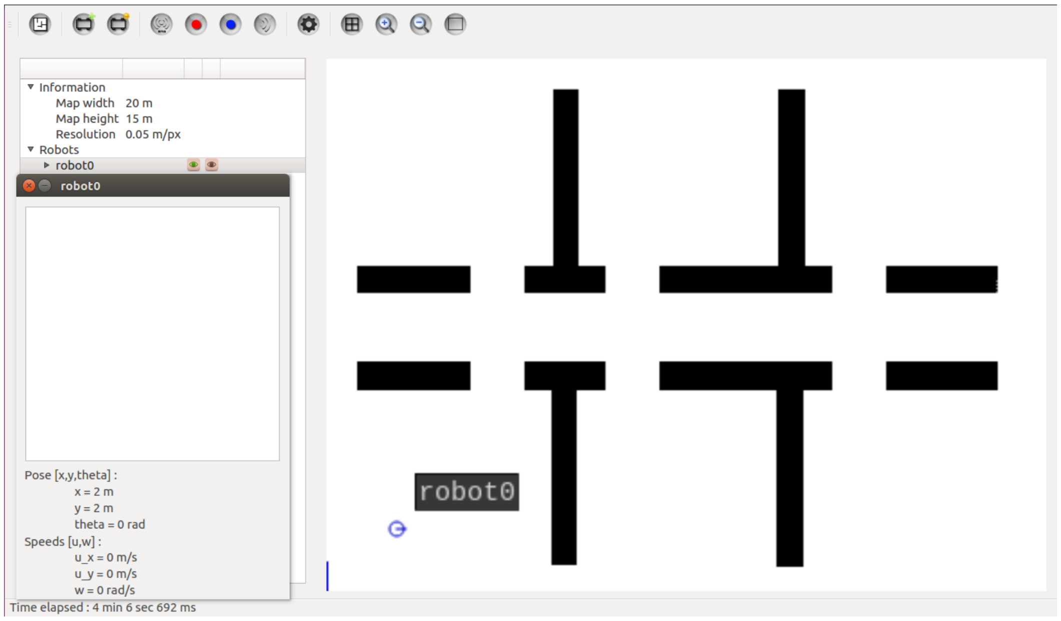
5.6 Testing

5.6.1 Load the map and robot into the STDR simulator



```
cheeyung@ubuntu:~$ roslaunch stdr_launchers server_with_map_and_gui_plus_robot.launch
... logging to /home/cheeyung/.ros/log/34c8c936-8258-11ea-a08f-000c29fe4eea/roslaunch-ubuntu-49945.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.126.135:35267/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES
  /
    robot_manager (nodelet/nodelet)
    robot_spawn_ubuntu_49945_1556449694614876785 (stdr_robot/robot_handler)
    stdr_gui_node_ubuntu_49945_4043288006383929809 (stdr_gui/stdr_gui_node)
    stdr_server (stdr_server/stdr_server_node)

ROS_MASTER_URI=http://192.168.126.135:11311

process[robot_manager-1]: started with pid [49962]
process[stdr_server-2]: started with pid [49963]
process[stdr_gui_node_ubuntu_49945_4043288006383929809-3]: started with pid [49964]
[ INFO] [1587541764.169591780]: Loading map from image "/home/cheeyung/catkin_ws/src/stdr_simulator/stdr_resources/maps/simple_rooms_no_walls.pn
g"
process[robot_spawn_ubuntu_49945_1556449694614876785-4]: started with pid [49978]
[ INFO] [1587541764.205699790]: Read a 400 X 300 map @ 0.050 m/cell
[robot_spawn_ubuntu_49945_1556449694614876785-4] process has finished cleanly
log file: /home/cheeyung/.ros/log/34c8c936-8258-11ea-a08f-000c29fe4eea/robot_spawn_ubuntu_49945_1556449694614876785-4*.log
```

Figure 5.22 shows a terminal with a command line that launch the “server_with_map_and_gui_plus_robot.launch”


Information
Map width 20 m
Map height 15 m
Resolution 0.05 m/px
Robots
robot0

robot0

Pose [x,y,theta] :
x = 2 m
y = 2 m
theta = 0 rad
Speeds [u,w] :
u_x = 0 m/s
u_y = 0 m/s
w = 0 rad/s

Time elapsed : 4 min 6 sec 692 ms

Figure 5.23 shows the output of the launch file

According to the Figure 5.23, the map and robot are successfully loaded into the STDR simulator, and the robot is placed in the correct position.

5.6.2 Move the robot

Next, we move the robot from the starting point (2, 2) to the target point (2, 10). If there is a collision or the goal cannot be reached, it means that there is a problem with the path planning algorithm or the code of "Safegoto.py".

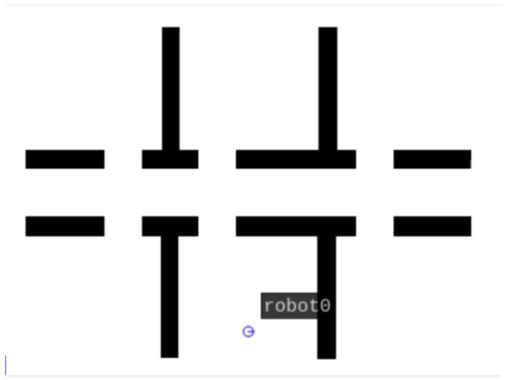
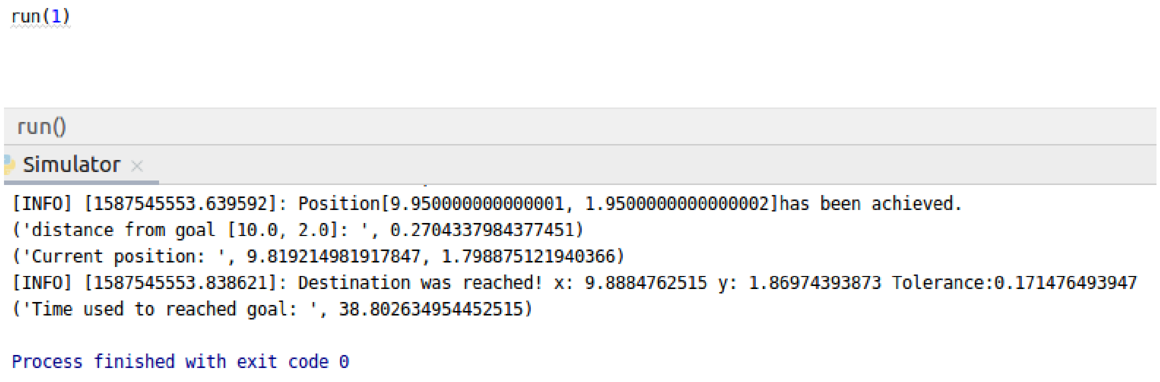


Figure 5.24 shows the robot at the target point (2, 10)

5.6.2.1 Dijkstra's algorithm

The function run(1) will run the Dijkstra's algorithm.



```
run(1)
```

run()

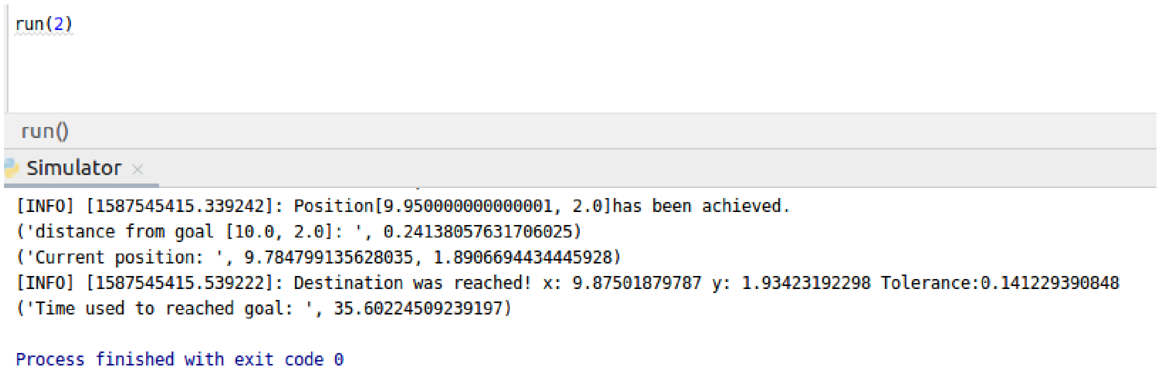
Simulator

```
[INFO] [1587545553.639592]: Position[9.950000000000001, 1.9500000000000002]has been achieved.
('distance from goal [10.0, 2.0]: ', 0.2704337984377451)
('Current position: ', 9.819214981917847, 1.798875121940366)
[INFO] [1587545553.838621]: Destination was reached! x: 9.8884762515 y: 1.86974393873 Tolerance:0.171476493947
('Time used to reached goal: ', 38.802634954452515)

Process finished with exit code 0
```

Figure 5.25 shows the robot successfully reached the goal point by using Dijkstra's

5.6.2.2 A* algorithm

The function run(2) will run the A* algorithm.



```
run(2)
```

run()

Simulator

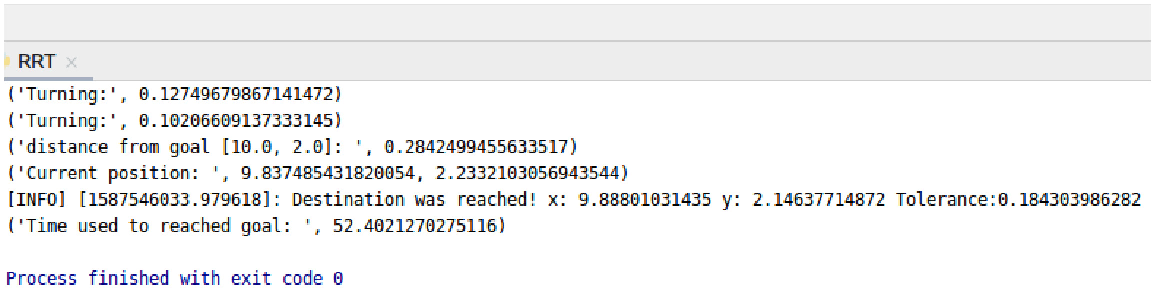
```
[INFO] [1587545415.339242]: Position[9.950000000000001, 2.0]has been achieved.
('distance from goal [10.0, 2.0]: ', 0.24138057631706025)
('Current position: ', 9.784799135628035, 1.8906694434445928)
[INFO] [1587545415.539222]: Destination was reached! x: 9.87501879787 y: 1.93423192298 Tolerance:0.141229390848
('Time used to reached goal: ', 35.60224509239197)

Process finished with exit code 0
```

*Figure 5.26 shows the robot successfully reached the goal point by using A**

5.6.2.3 RRT algorithm

The function rrt.Planning() will run the RRT algorithm.

```
rrt = RRT(map, qstart, qgoal, grid_size, step_size, max_steps, goal_prob)
p = rrt.Planning()
robot = SafeGoto()
robot.travel(p)
```

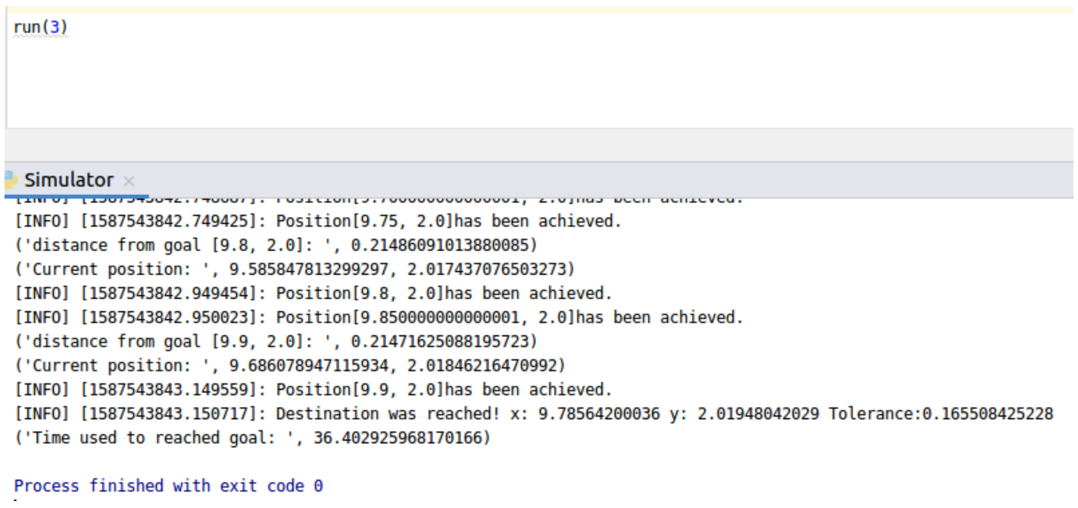


```
RRT
('Turning:', 0.12749679867141472)
('Turning:', 0.10206609137333145)
('distance from goal [10.0, 2.0]: ', 0.2842499455633517)
('Current position: ', 9.837485431820054, 2.2332103056943544)
[INFO] [1587546033.979618]: Destination was reached! x: 9.88801031435 y: 2.14637714872 Tolerance:0.184303986282
('Time used to reached goal: ', 52.4021270275116)

Process finished with exit code 0
```

Figure 5.27 shows the robot successfully reached the goal point by using RRT

5.6.2.4 Modified A* algorithm

The function run(3) will run the modified A* algorithm.



```
run(3)
```

```
Simulator
[INFO] [1587543842.749425]: Position[9.75, 2.0]has been achieved.
('distance from goal [9.8, 2.0]: ', 0.21486091013880085)
('Current position: ', 9.585847813299297, 2.017437076503273)
[INFO] [1587543842.949454]: Position[9.8, 2.0]has been achieved.
[INFO] [1587543842.950023]: Position[9.850000000000001, 2.0]has been achieved.
('distance from goal [9.9, 2.0]: ', 0.21471625088195723)
('Current position: ', 9.686078947115934, 2.01846216470992)
[INFO] [1587543843.149559]: Position[9.9, 2.0]has been achieved.
[INFO] [1587543843.150717]: Destination was reached! x: 9.78564200036 y: 2.01948042029 Tolerance:0.165508425228
('Time used to reached goal: ', 36.402925968170166)

Process finished with exit code 0
```

*Figure 5.28 shows the robot successfully reached the goal point by using Modified A**

Chapter 6: Experiment Result and Comparison

6.1 1st Experiment

The first experiment is to use the "simple_rooms_no_walls" map, and the robot must move from the starting point (2, 2) to the target point (16, 12). The position of robot 0 in Figure 6.1 is the starting point (2, 2), and the position of robot 1 in Figure 6.1 is the target point (16, 12).

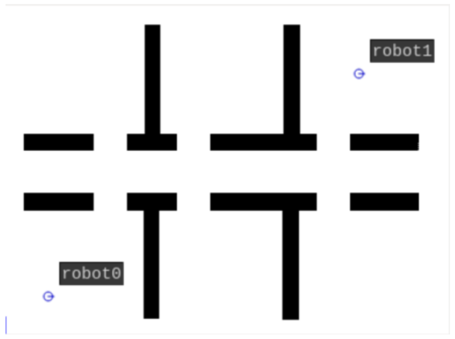
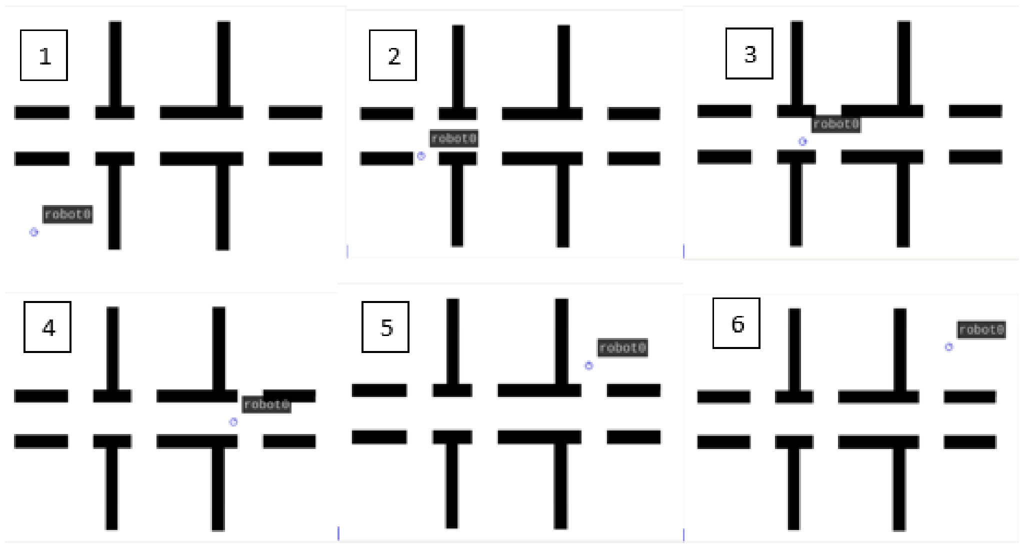


Figure 6.1 shows the two robots in "simple_rooms_no_walls"

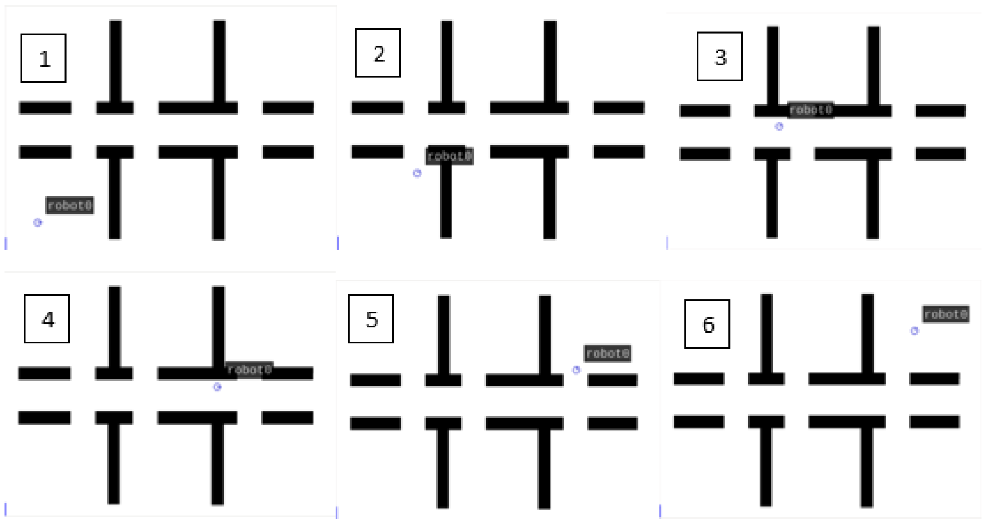
6.1.1 Dijkstra's algorithm



```
Path found!
('Time used to calculate the path: ', 11.418997049331665)
('The distance is:', 414.97770542341397)
('Explored: ', 73940)
```

Figure 6.2 shows the result of the Dijkstra

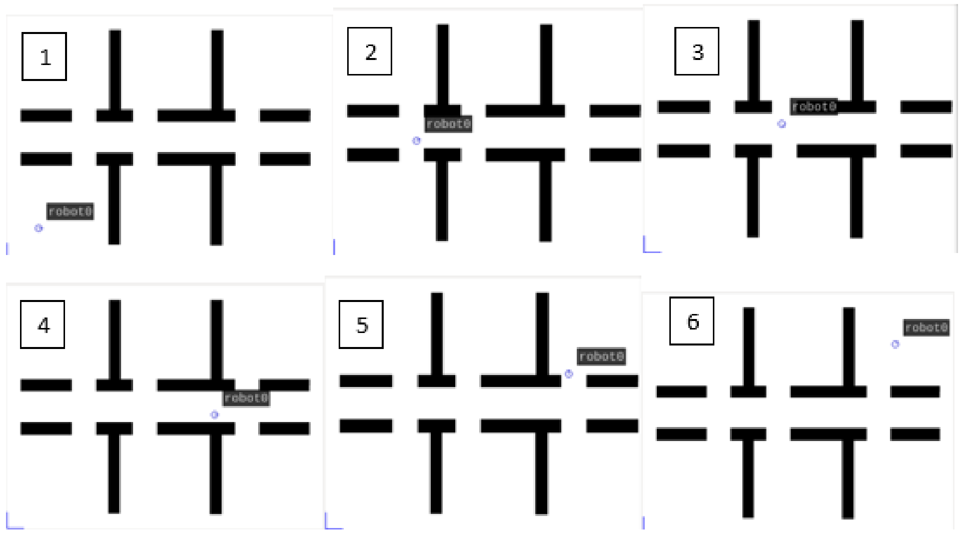
6.1.2 A* algorithm



```
Path found!
('Time used to calculate the path: ', 4.8768370151519775)
('The distance is:', 423.26197667087604)
('Explored: ', 22860)
```

*Figure 6.3 shows the result of the A**

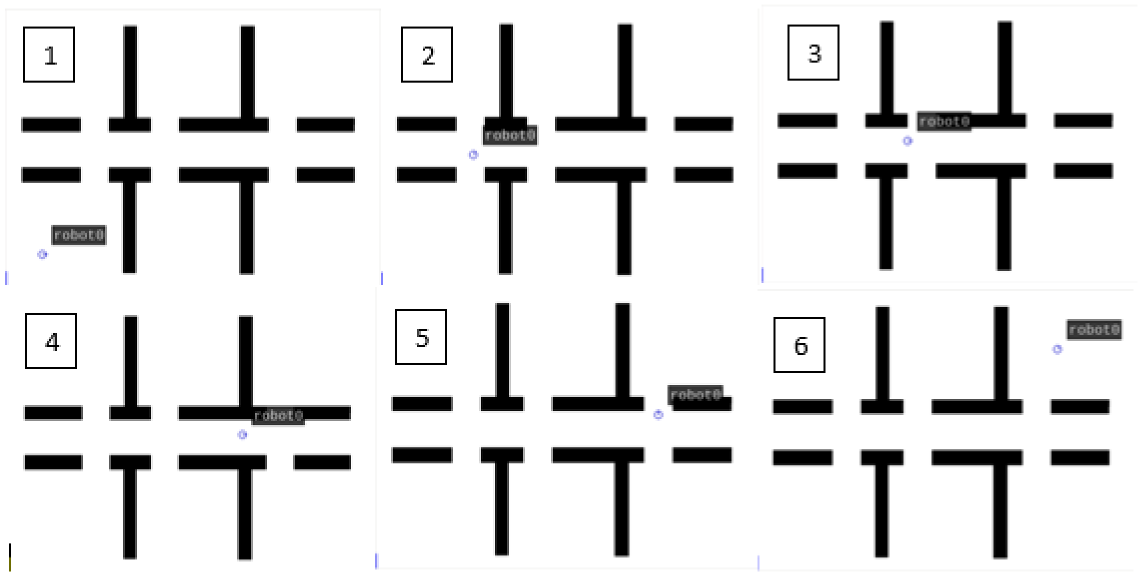
6.1.3 Modified A* algorithm

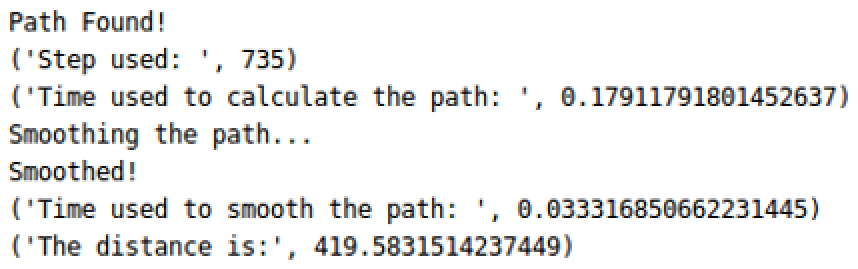


```
Path found!
('Time used to calculate the path: ', 2.751091957092285)
('The distance is:', 440.6589462905465)
('Explored: ', 18315)
```

*Figure 6.4 shows the result of the Modified A**

6.1.4 RRT algorithm





```
Path Found!
('Step used: ', 735)
('Time used to calculate the path: ', 0.17911791801452637)
Smoothing the path...
Smoothed!
('Time used to smooth the path: ', 0.033316850662231445)
('The distance is:', 419.5831514237449)
```

Figure 6.5 shows the result of the RRT

6.1.5 Comparison of Experiment I

	Explored cell	**Iteration**	**Distance (pixel)**	**Time**
Dijkstra	73940	-	414.9777	11.4290
A*	22860	-	423.2620	4.8768
Modified A*	18315	-	440.5890	2.7511
RRT	-	735	419.5832	0.1791

Table 6.1 shows the comparison of experiment I

According to the Table 6.1, Dijkstra has the smallest distance among these algorithms. However, Dijkstra spent the most time to find the path. In contrast, RRT took the least amount of time to find a path with a distance of 419.5832. In addition, the modified A* is faster than A*, but the modified A* has a greater distance than A*.

6.2 2nd Experiment

The second experiment is to use the "sparse_obstacles" map, and the robot must move from the starting point (2, 2) to the target point (14, 12). The position of robot 0 in Figure 6.6 is the starting point (2, 2), and the position of robot 1 in Figure 6.6 is the target point (14, 12).

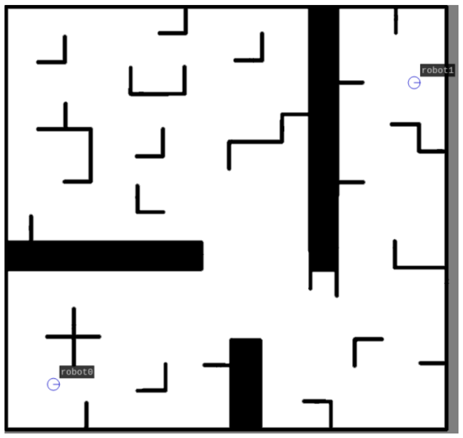
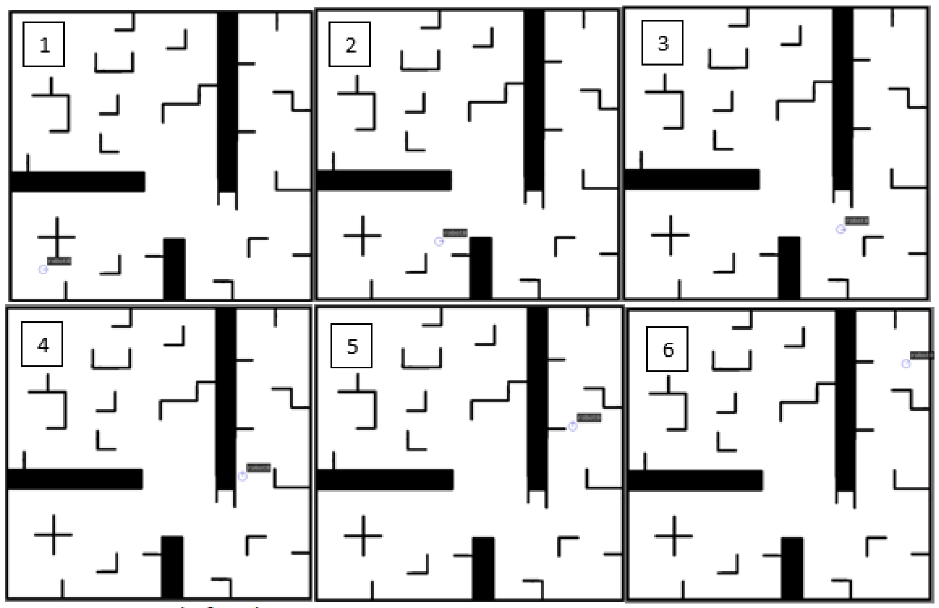


*Figure 6.6 shows the two robots in "*sparse_obstacles*"*

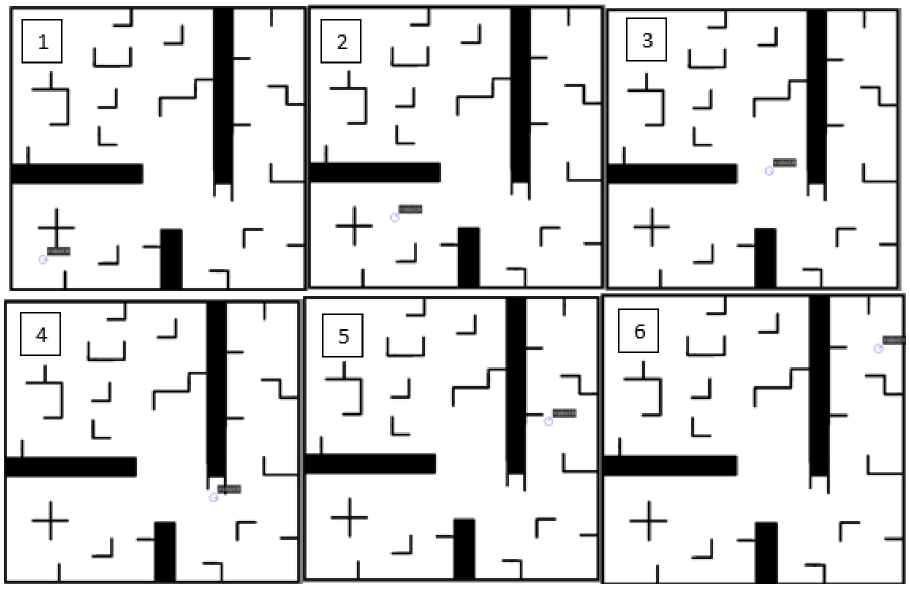
6.2.1 Dijkstra's algorithm



```
Path found!
('Time used to calculate the path: ', 73.85504007339478)
('The distance is:', 961.7544007200521)
('Explored: ', 276362)
```

Figure 6.7 shows the result of the Dijkstra

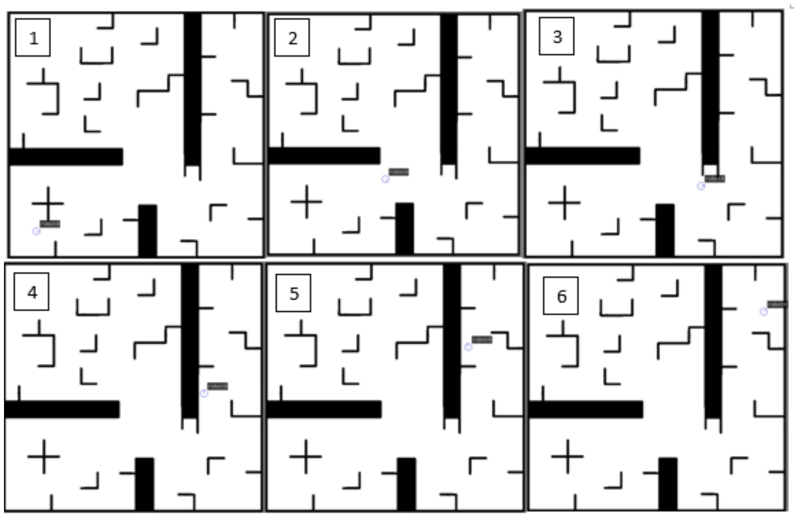
6.2.2 A* algorithm



```
Path found!
('Time used to calculate the path: ', 71.78962397575378)
('The distance is:', 1057.0235200658667)
('Explored: ', 158097)
```

*Figure 6.8 shows the result of the A**

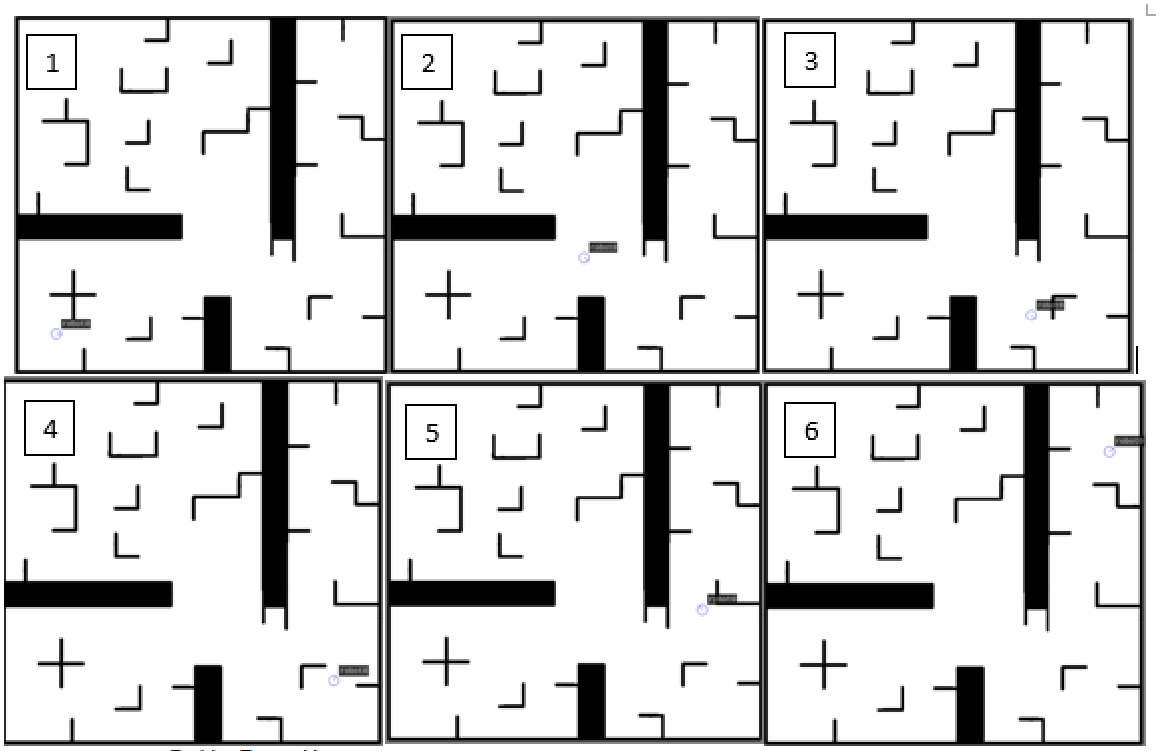
6.2.3 Modified A* algorithm



```
Path found!
('Time used to calculate the path: ', 42.35676717758179)
('The distance is:', 1031.8275605729727)
('Explored: ', 105922)
```

*Figure 6.9 shows the result of the Modified A**

6.2.3 RRT algorithm



```
Path Found!
('Step used: ', 12786)
('Time used to calculate the path: ', 30.70873188972473)
Smoothing the path...
Smoothed!
('Time used to smooth the path: ', 0.22470998764038086)
('The distance is:', 1237.4604640595624)
```

Figure 6.10 shows the result of the RRT

6.2.5 Comparison of Experiment II

	Explored cell	**Iteration**	**Distance (pixel)**	**Time**
Dijkstra	276362	-	961.7544	73.8550
A*	158097	-	1057.0235	71.7896
Modified A*	105922	-	1031.8276	42.3568
RRT	-	12786	1237.4605	30.7087

Table 6.2 shows the comparison of experiment II

According to Table 6.2, Dijkstra has the smallest distance among these algorithms. However, Dijkstra spends most of its time searching for route. In contrast, RRT takes the least time but has the largest distance. In addition, modified A* is faster than A* and the distance of modified A* is less than A*.

6.3 3rd Experiment

The third experiment is to use the "utar" map, and the robot must move from the starting point (4, 19) to the target point (28, 20). The position of robot 0 in Figure 6.11 is the starting point (4, 19), and the position of robot 1 in Figure 6.11 is the target point (28, 20).

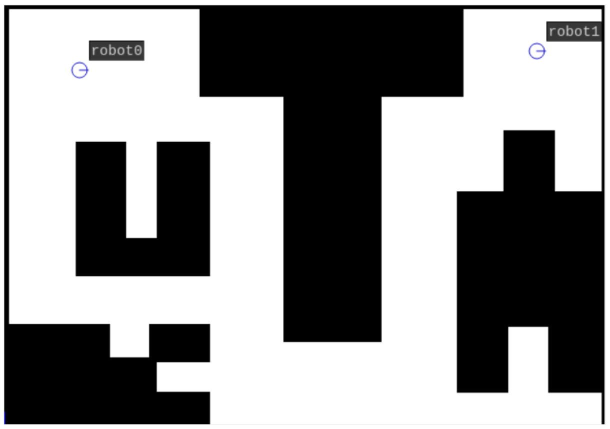
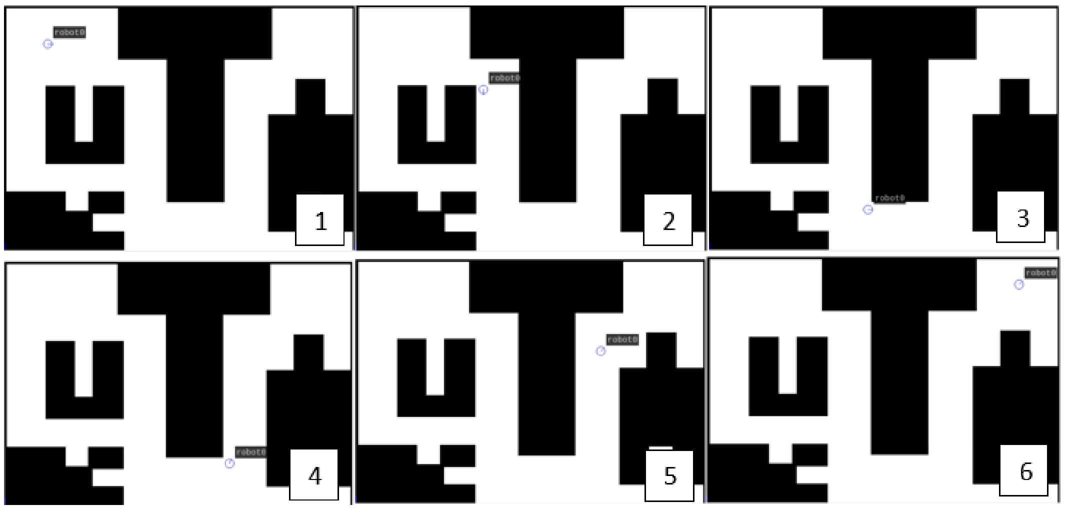
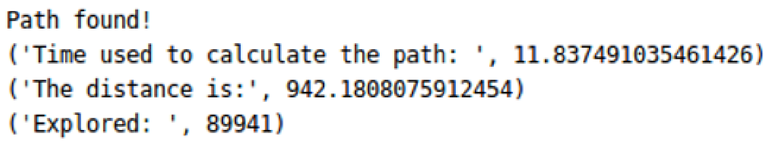


*Figure 6.11 shows the two robots in “*utar*”*

6.3.1 Dijkstra's algorithm

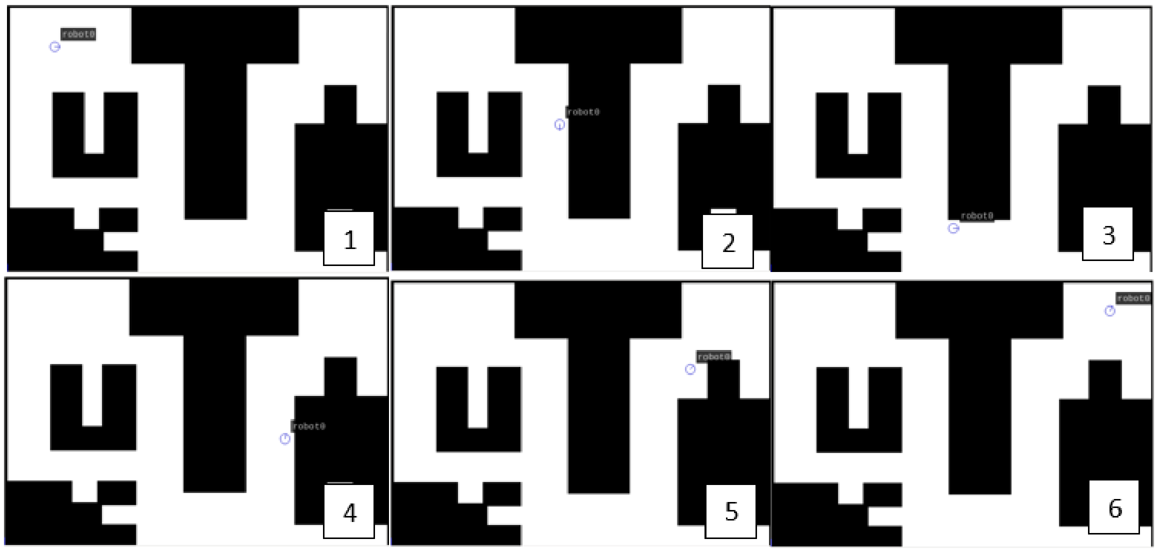


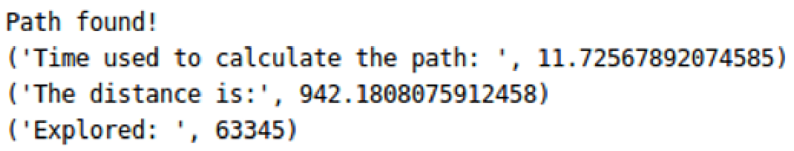


```
Path found!
('Time used to calculate the path: ', 11.837491035461426)
('The distance is:', 942.1808075912454)
('Explored: ', 89941)
```

Figure 6.12 shows the result of the Dijkstra

6.3.2 A* algorithm

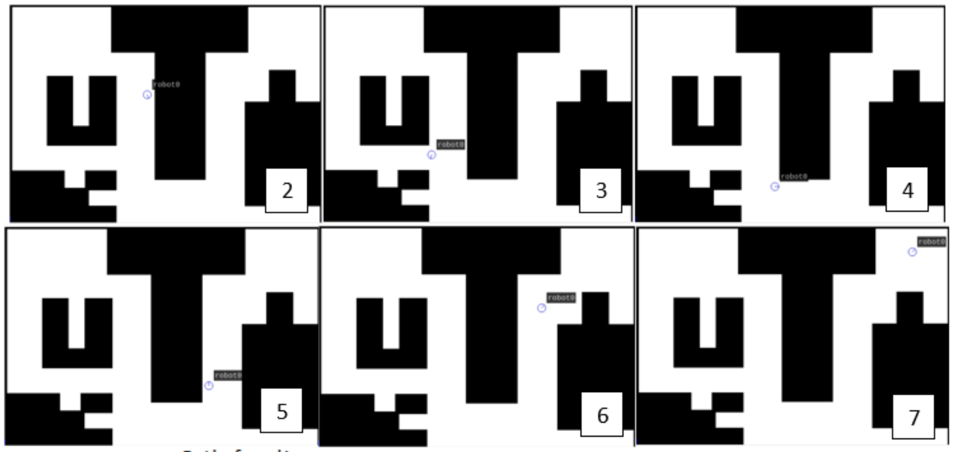




```
Path found!
('Time used to calculate the path: ', 11.72567892074585)
('The distance is:', 942.1808075912458)
('Explored: ', 63345)
```

*Figure 6.13 shows the result of the A**

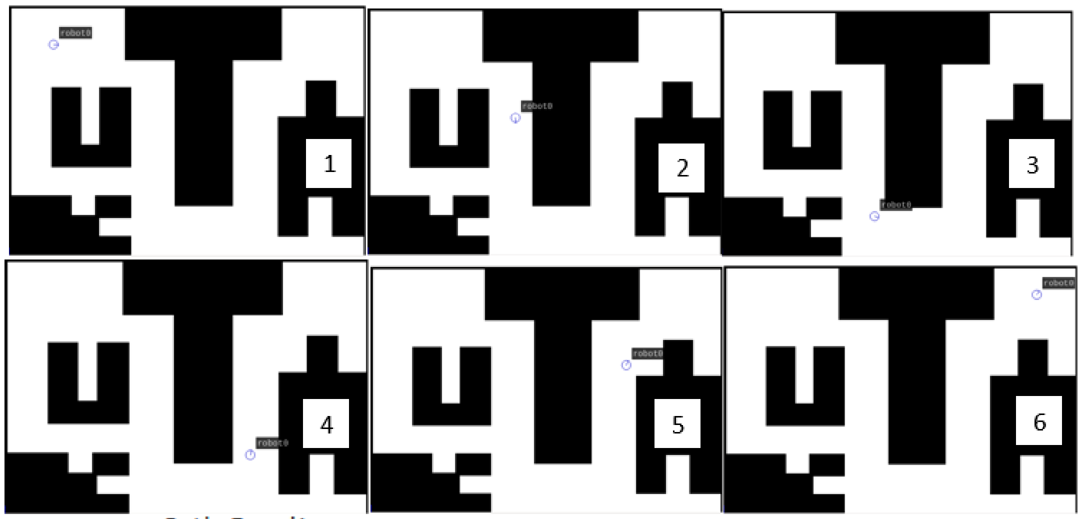
6.3.3 Modified A* algorithm



```
Path found!
('Time used to calculate the path: ', 12.664036989212036)
('The distance is:', 988.6437943888992)
('Explored: ', 74747)
```

*Figure 6.14 shows the result of the Modified A**

6.3.3 RRT algorithm



```
Path Found!
('Step used: ', 2488)
('Time used to calculate the path: ', 1.1798150539398193)
Smoothing the path...
Smoothed!
('Time used to smooth the path: ', 0.11949896812438965)
('The distance is:', 928.073771852548)
```

Figure 6.15 shows the result of the RRT

6.3.5 Comparison of Experiment III

	Explored cell	Iteration	Distance (pixel)	Time
Dijkstra	89941	-	942.1808	11.8375
A*	63345	-	942.1808	11.7257
Modified A*	74747	-	988.6438	12.6640
RRT	-	2488	928.0737	1.1798

Table 6.3 shows the comparison of experiment III

According to Table 6.3, the RRT has a minimum distance and time for calculating the route. The distance between Dijkstra and A* is the same. However, A* uses less time than Dijkstra. In this experiment, the modified A* is slower than A* and the distance is greater than A*.

6.4 4th Experiment

The fourth experiment is to use the "utar2" map, which is more obstacles added in the “utar” map. The robot must move from the starting point (4, 19) to the target point (28, 20). The position of robot 0 in Figure 6.16 is the starting point (4, 19), and the position of robot 1 in Figure 6.16 is the target point (28, 20).

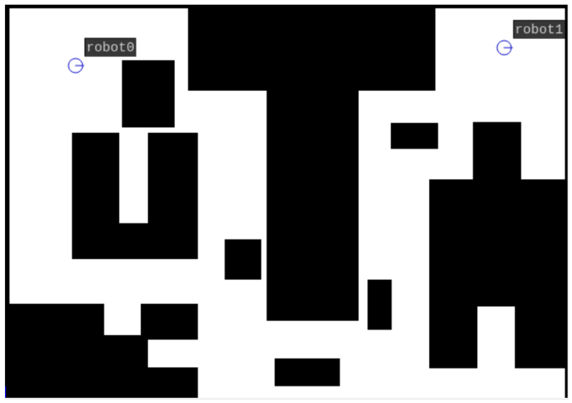
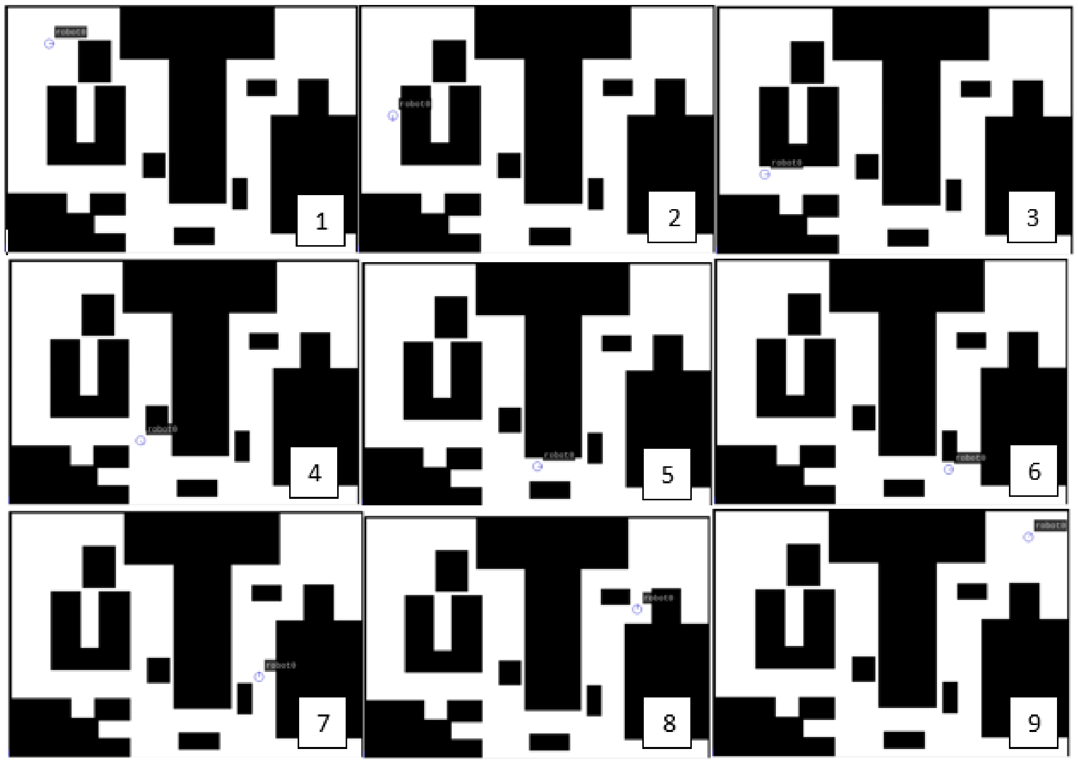


*Figure 6.16 shows the two robots in “*utar2*”*

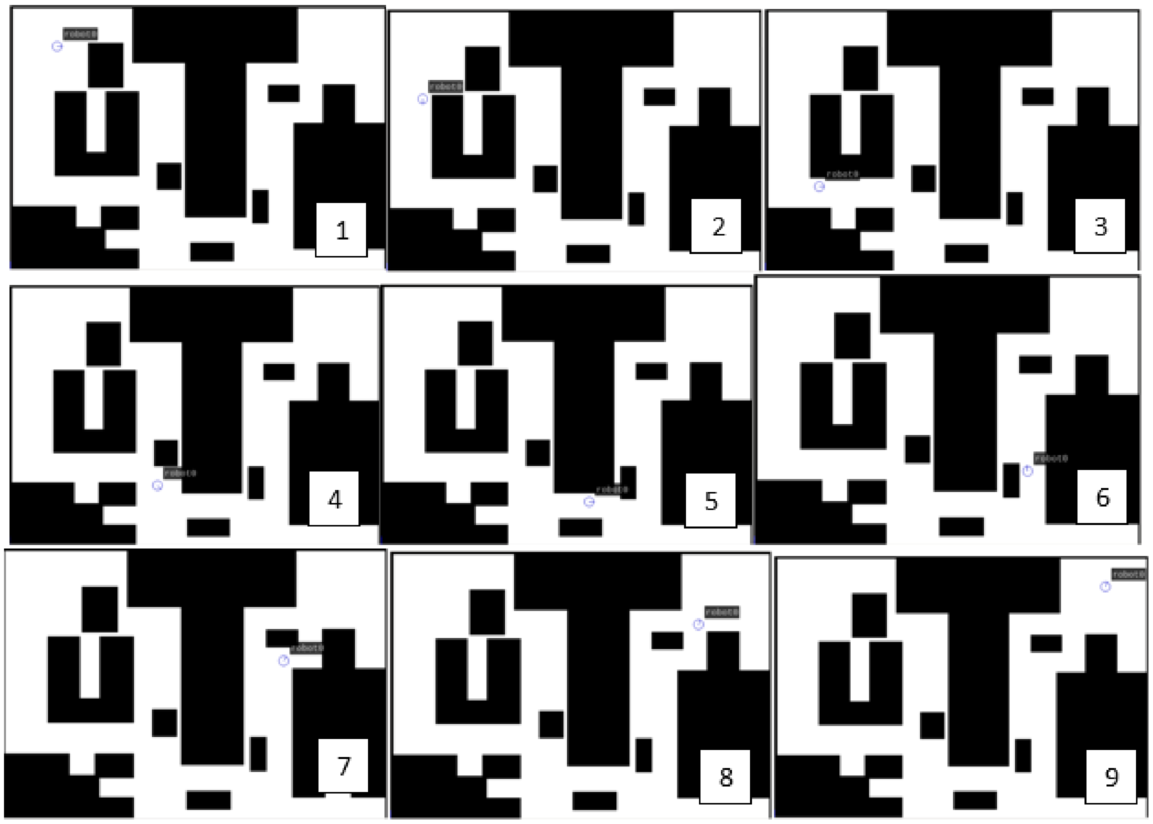
6.4.1 Dijkstra's algorithm



```
Path found!
('Time used to calculate the path: ', 8.228917121887207)
('The distance is:', 1047.0437226013973)
('Explored: ', 67378)
```

Figure 6.17 shows the result of the Dijkstra

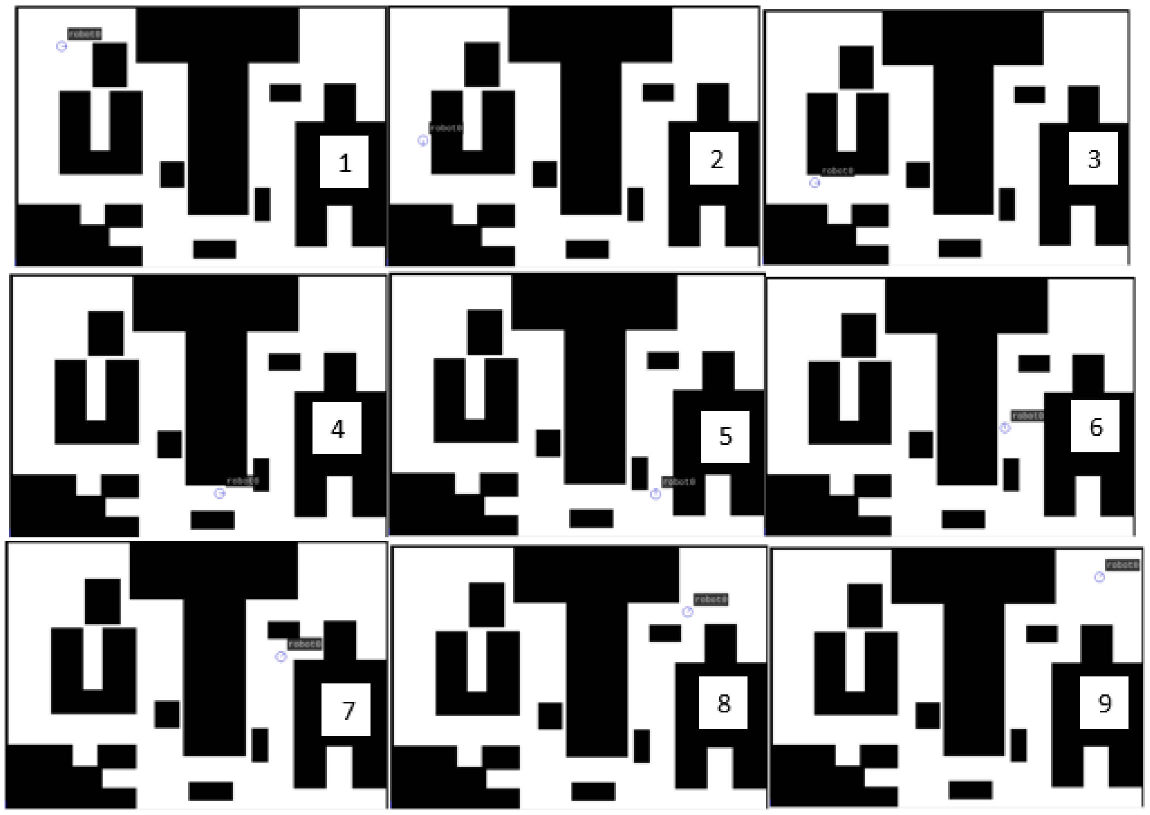
6.4.2 A* algorithm



```
Path found!
('Time used to calculate the path: ', 8.113867998123169)
('The distance is:', 1065.2691193458136)
('Explored: ', 53184)
```

*Figure 6.18 shows the result of the A**

6.4.3 Modified A* algorithm



```
Path found!
('Time used to calculate the path: ', 8.034536838531494)
('The distance is:', 1097.2346314604083)
('Explored: ', 55708)
```

*Figure 6.19 shows the result of the Modified A**

6.4.3 RRT algorithm

```
Finish Iteration! Cannot find the path!
('Time used: ', 93.85454297065735)
```

Figure 6.20 shows the result of the RRT

6.4.5 Comparison of Experiment IV

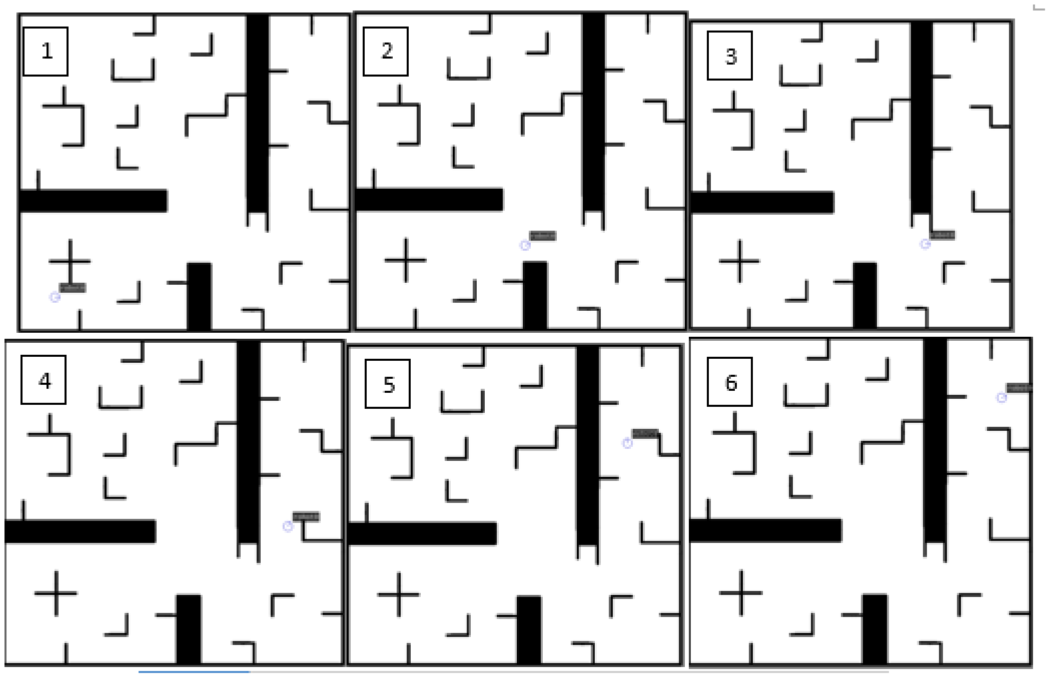
	Explored cell	**Iteration**	**Distance (pixel)**	**Time**
Dijkstra	67378	-	1047.0437	8.2289
A*	53184	-	1065.2691	8.1139
Modified A*	55708	-	1097.2346	8.0345
RRT	-	-	-	-

Table 6.4 shows the comparison of experiment IV

According to Table 6.4, Dijkstra has the smallest distance among these algorithms. However, Dijkstra spends most of its time searching for route. In addition, modified A* is faster than A* but the distance of modified A* is longer than A*. Apart from that, the RRT still cannot find the path after 50,000 iterations.

6.5 Analysis the experiments

Through these experiments, we know that Dijkstra's always result the shortest distance path, but in most cases, it takes the longest time to calculate the path. Apart from that, the modified A* is faster than A*, but sometimes the modified A* is slower or return a distance greater than distance of A*. This may be because the modified A * of the forward search and the backward search has a larger bifurcation, so they meet for a longer time, which also results in a longer distance. The RRT algorithm takes the least amount of time to find the route, but the route is not optimal. In addition, when the map contains many obstacles, RRT needs more iteration time to find the path. In Experiment IV, we can see that RRT cannot find a route in a map with too many obstacles even though it has already spent 50,000 iterations. Moreover, the route found by RRT every time may be different from the last time. To prove this, experiment II was conducted again using the RRT algorithm.



```
('Step used: ', 10242)
('Time used to calculate the path: ', 20.51811408996582)
Smoothing the path...
Smoothed!
('Time used to smooth the path: ', 0.16437292098999023)
('The distance is:', 1230.335569493429)
```

Figure 6.21 shows the result of the RRT

	Iteration	**Distance (pixel)**	**Time**
1st RRT	12786	1237.4605	30.7087
2nd RRT	5361	1230.3356	20.5181

Table 6.5 shows the comparison of 2 RRTs

According to Table 6.5, the results prove that each time route calculated by RRT is different.

Chapter 7: Conclusion

Path planning is a process of finding an optimal path for robot move from source to destination. Besides that, different path planning algorithms has its own applicable fields, performance in various situations. Therefore, it is significant to understand the properties of various path planning algorithms. In this paper, Robot Operating System (ROS), and STDR simulator are used to simulate a robot that is implemented by various path planning algorithms and environments.

In this paper, we can conclude that if you need to find the shortest distance path, Dijkstra is the best choice. If you want to speed up the search time and there are fewer obstacles on the map, it is recommended to use A*, modified A* or RRT. However, if there are many obstacles on the map, RRT will not be suitable for searching. They will use shorter time to search for the path, but they may not lead to the shortest path. In addition, the route discovered by RRT may be different from the route discovered last time.

By reading this paper, readers will be able to understand the properties of various path planning algorithms without having to spend extra time reading irrelevant information. Also, the readers can observe the difference between path planning algorithms more clearly on the simulator. All in all, if we want to design a robot, we can better understand which algorithm is best suited for which situation to use in order to get the maximum benefit.

In future work, more path planning algorithms will be reviewed and compared. In addition, existing path planning algorithms will be modified or improved to increase accuracy or reduce the time used to calculate paths. For example, the modified A* search speed can be improved by using parallelism.

BIBLIOGRAPHY

BrainKart. n.d.. *Brainkart*. [online] Available at: <https://www.brainkart.com/article/A--Search--Concept,-Algorithm,-Implementation,-Advantages,-Disadvantages_8883/ > [Accessed 13 August 2019].

Cs.cmu.edu. 2002. [online] Available at: <https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf> [Accessed 17 August 2019].

Cs.cmu.edu. 2007. [online] Available at: <http://www.cs.cmu.edu/~motionplanning/lecture/lec21.pdf> [Accessed 10 November 2019].

Design News. 2019. *ROS 101: An Intro To The Robot Operating System*. [online] Available at: <https://www.designnews.com/gadget-freak/ros-101-intro-robot-operating-system/107053141061075> [Accessed 15 November 2019].

Hindex.org. 2014. [online] Available at: <http://www.hindex.org/2014/p520.pdf> [Accessed 17 August 2019].

IoT Agenda. 2019. *What Is A Drone? - Definition From Whatis.Com*. [online] Available at: <https://internetofthingsagenda.techtarget.com/definition/drone> [Accessed 16 November 2019].

Kangutkar, R., 2017. *Obstacle Avoidance And Path Planning For Smart Indoor Agents*. [online] RIT Scholar Works. Available at: <https://scholarworks.rit.edu/theses/9521/> [Accessed 12 November 2019].

Merriam-webster.com. 2019. *Definition Of ROBOT*. [online] Available at: <https://www.merriam-webster.com/dictionary/robot> [Accessed 14 November 2019].

Msl.cs.uiuc.edu. n.d.. [online] Available at: <http://msl.cs.uiuc.edu/~lavalle/papers/LavKuf01.pdf> [Accessed 15 August 2019].

Pyo, Y., Cho, H., Jung, R. and Lim, T., 2017. *ROS Robot Programming*. ROBOTIS Co.,Ltd.

RobotWorx. 2019. *Benefits Of Using Robotics*. [online] Available at: <https://www.robots.com/articles/benefits-of-using-robotics> [Accessed 15 November 2019].

Sciencedirect.com. 2017. *Path Planning - An Overview | Sciencedirect Topics*. [online] Available at: <https://www.sciencedirect.com/topics/engineering/path-planning> [Accessed 16 August 2019].

Soffar, H., 2019. *Aquatic Robots (Swimming Robots Or Robot Fish) Types, Uses, Cons & Pros | Science Online*. [online] Science online. Available at: <https://www.online-sciences.com/robotics/aquatic-robots-swimming-robots-or-robot-fish-types-uses-cons-pros/> [Accessed 18 November 2019].

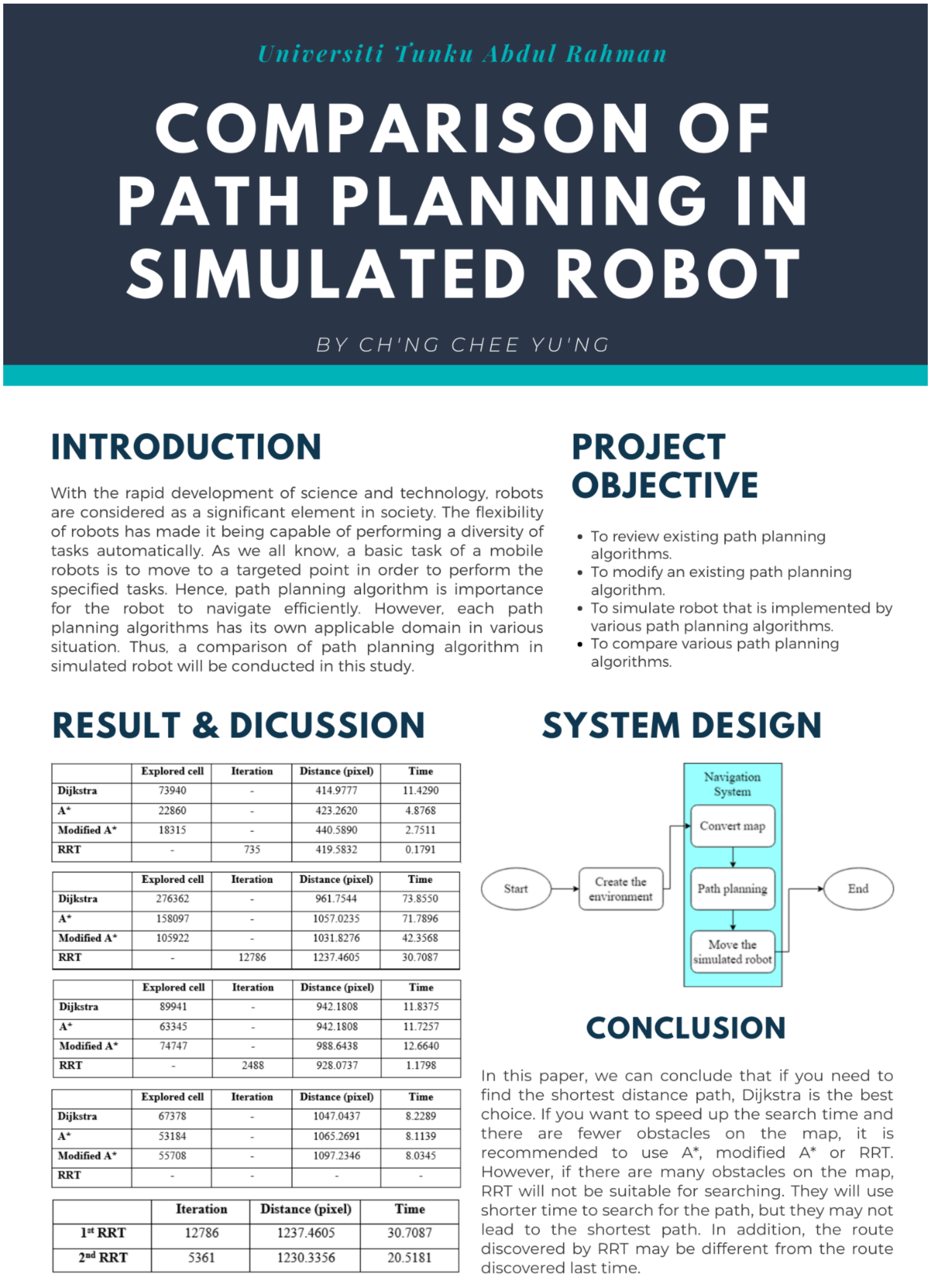
Techunited.nl. 2010. *Path Planning: RRT - Explanation RRT - Tech United Eindhoven*. [online] Available at: <http://www.techunited.nl/wiki/index.php?title=Path_planning:_RRT_-_Explanation_RRT> [Accessed 15 August 2019].

Users.monash.edu. n.d.. [online] Available at: <http://users.monash.edu/~cema/courses/FIT3094/lecturePDFs/lecture6a_Astar.pdf> [Accessed 16 August 2019].

Wiki.ros.org. 2018. *ROS/Introduction - ROS Wiki*. [online] Available at: <http://wiki.ros.org/ROS/Introduction> [Accessed 18 November 2019].

Wiki.ros.org. 2019. *Turtlesim/Tutorials/Moving In A Straight Line - ROS Wiki*. [online] Available at: <http://wiki.ros.org/turtlesim/Tutorials/Moving%20in%20a%20Straight%20Line> [Accessed 19 November 2019].

POSTER



Universiti Tunku Abdul Rahman

COMPARISON OF PATH PLANNING IN SIMULATED ROBOT

BY CH'NG CHEE YU'NG

INTRODUCTION

With the rapid development of science and technology, robots are considered as a significant element in society. The flexibility of robots has made it being capable of performing a diversity of tasks automatically. As we all know, a basic task of a mobile robots is to move to a targeted point in order to perform the specified tasks. Hence, path planning algorithm is importance for the robot to navigate efficiently. However, each path planning algorithms has its own applicable domain in various situation. Thus, a comparison of path planning algorithm in simulated robot will be conducted in this study.

PROJECT OBJECTIVE

- To review existing path planning algorithms.
- To modify an existing path planning algorithm.
- To simulate robot that is implemented by various path planning algorithms.
- To compare various path planning algorithms.

RESULT & DICUSSION

	Explored cell	Iteration	Distance (pixel)	Time
Dijkstra	73940	-	414.9777	11.4290
A*	22860	-	423.2620	4.8768
Modified A*	18315	-	440.5890	2.7511
RRT	-	735	419.5832	0.1791

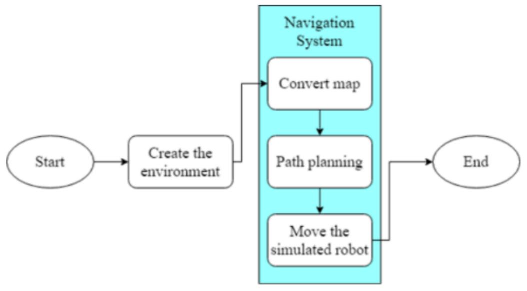
	Explored cell	Iteration	Distance (pixel)	Time
Dijkstra	276362	-	961.7544	73.8550
A*	158097	-	1057.0235	71.7896
Modified A*	105922	-	1031.8276	42.3568
RRT	-	12786	1237.4605	30.7087

	Explored cell	Iteration	Distance (pixel)	Time
Dijkstra	89941	-	942.1808	11.8375
A*	63345	-	942.1808	11.7257
Modified A*	74747	-	988.6438	12.6640
RRT	-	2488	928.0737	1.1798

	Explored cell	Iteration	Distance (pixel)	Time
Dijkstra	67378	-	1047.0437	8.2289
A*	53184	-	1065.2691	8.1139
Modified A*	55708	-	1097.2346	8.0345
RRT	-	-	-	-

	Iteration	Distance (pixel)	Time
1st RRT	12786	1237.4605	30.7087
2nd RRT	5361	1230.3356	20.5181

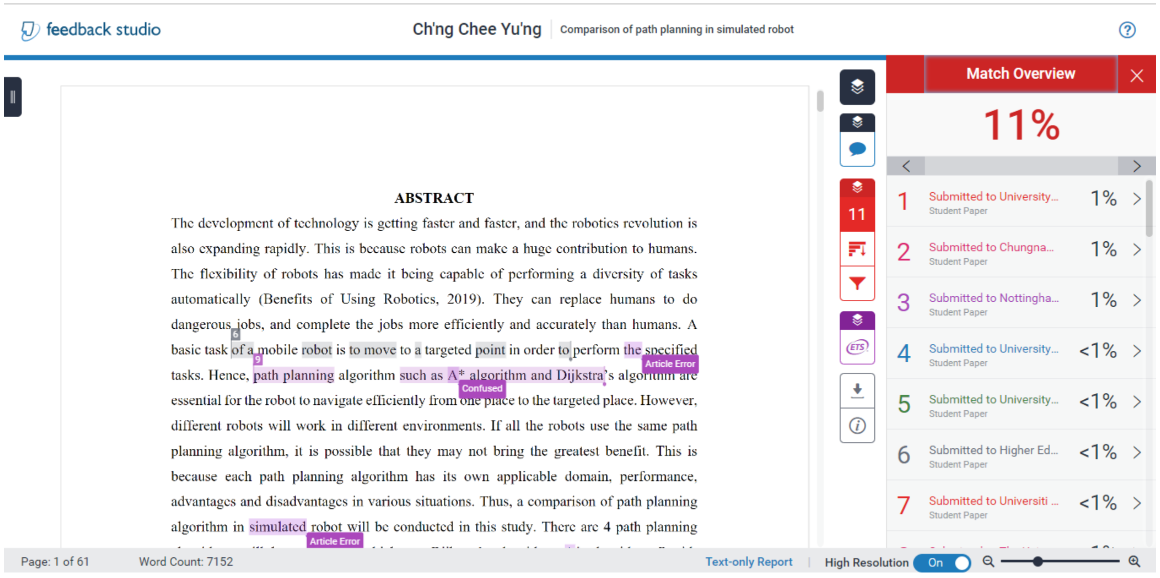
SYSTEM DESIGN



CONCLUSION

In this paper, we can conclude that if you need to find the shortest distance path, Dijkstra is the best choice. If you want to speed up the search time and there are fewer obstacles on the map, it is recommended to use A*, modified A* or RRT. However, if there are many obstacles on the map, RRT will not be suitable for searching. They will use shorter time to search for the path, but they may not lead to the shortest path. In addition, the route discovered by RRT may be different from the route discovered last time.

PLAGIARISM CHECK RESULT



feedback studio

Ch'ng Chee Yu'ng | Comparison of path planning in simulated robot

ABSTRACT

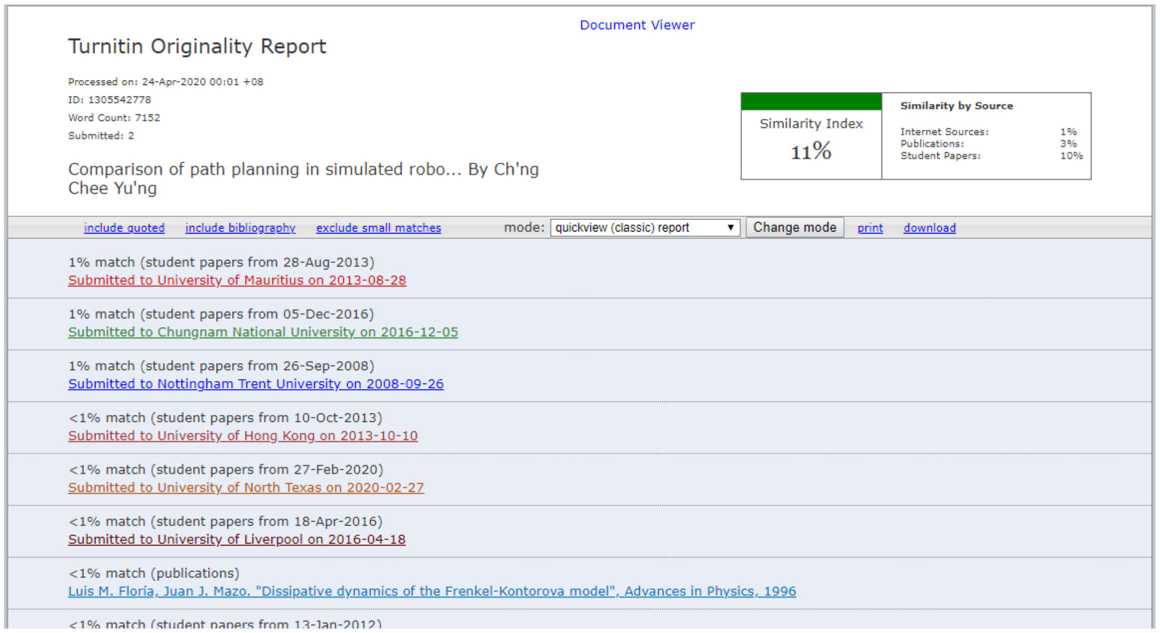
The development of technology is getting faster and faster, and the robotics revolution is also expanding rapidly. This is because robots can make a huge contribution to humans. The flexibility of robots has made it being capable of performing a diversity of tasks automatically (Benefits of Using Robotics, 2019). They can replace humans to do dangerous jobs, and complete the jobs more efficiently and accurately than humans. A basic task of a mobile robot is to move to a targeted point in order to perform the specified tasks. Hence, path planning algorithm such as A* algorithm and Dijkstra's algorithm are essential for the robot to navigate efficiently from one place to the targeted place. However, different robots will work in different environments. If all the robots use the same path planning algorithm, it is possible that they may not bring the greatest benefit. This is because each path planning algorithm has its own applicable domain, performance, advantages and disadvantages in various situations. Thus, a comparison of path planning algorithm in simulated robot will be conducted in this study. There are 4 path planning

Match Overview

11%

1	Submitted to University... Student Paper	1%
2	Submitted to Chungna... Student Paper	1%
3	Submitted to Nottingha... Student Paper	1%
4	Submitted to University... Student Paper	<1%
5	Submitted to University... Student Paper	<1%
6	Submitted to Higher Ed... Student Paper	<1%
7	Submitted to Universiti ... Student Paper	<1%

Page: 1 of 61 Word Count: 7152 Text-only Report | High Resolution On



Document Viewer

Turnitin Originality Report

Processed on: 24-Apr-2020 00:01 +08
ID: 1305542778
Word Count: 7152
Submitted: 2

Comparison of path planning in simulated robo... By Ch'ng Chee Yu'ng

Similarity Index	Similarity by Source	
11%	Internet Sources:	1%
	Publications:	3%
	Student Papers:	10%

include quoted include bibliography exclude small matches mode: quickview (classic) report Change mode print download

1% match (student papers from 28-Aug-2013)
Submitted to University of Mauritius on 2013-08-28

1% match (student papers from 05-Dec-2016)
Submitted to Chungnam National University on 2016-12-05

1% match (student papers from 26-Sep-2008)
Submitted to Nottingham Trent University on 2008-09-26

<1% match (student papers from 10-Oct-2013)
Submitted to University of Hong Kong on 2013-10-10

<1% match (student papers from 27-Feb-2020)
Submitted to University of North Texas on 2020-02-27

<1% match (student papers from 18-Apr-2016)
Submitted to University of Liverpool on 2016-04-18

<1% match (publications)
Luis M. Floría, Juan J. Mazo. "Dissipative dynamics of the Frenkel-Kontorova model", Advances in Physics, 1996

<1% match (student papers from 13-Jan-2012)

Universiti Tunku Abdul Rahman			
Form Title : Supervisor's Comments on Originality Report Generated by Turnitin for Submission of Final Year Project Report (for Undergraduate Programmes)			
Form Number: FM-IAD-005	Rev No.: 0	Effective Date: 01/10/2013	Page No.: 1of 1

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Full Name(s) of Candidate(s)	Ch'ng Chee Yu'ng
ID Number(s)	16ACB04126
Programme / Course	CS
Title of Final Year Project	Comparion of Path Planning in Simulated Robot

Similarity	**Supervisor's Comments (Compulsory if parameters of originality exceeds the limits approved by UTAR)**
Overall similarity index: 11 **%** **Similarity by source** Internet Sources: 1 % Publications: 3 % Student Papers: 10 %	
Number of individual sources listed of more than 3% similarity: 0	
Parameters of originality required and limits approved by UTAR are as Follows: **(i) Overall similarity index is 20% and below, and** **(ii) Matching of individual sources listed must be less than 3% each, and** **(iii) Matching texts in continuous block must not exceed 8 words** ***Note: Parameters (i) – (ii) shall exclude quotes, bibliography and text matches which are less than 8 words.***	

Note Supervisor/Candidate(s) is/are required to provide softcopy of full set of the originality report to Faculty/Institute

Based on the above results, I hereby declare that I am satisfied with the originality of the Final Year Project Report submitted by my student(s) as named above.

[illegible]

Signature of Supervisor	Signature of Co-Supervisor
Name: CHANG JING JING	Name:
Date: 4/22/2020	Date:

UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY
(KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

Student Id	16ACB04126
Student Name	Ch'ng Chee Yu'ng
Supervisor Name	Dr. Chang Jing Jing

TICK (√)	**DOCUMENT ITEMS** Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
√	Front Cover
√	Signed Report Status Declaration Form
√	Title Page
√	Signed form of the Declaration of Originality
√	Acknowledgement
√	Abstract
√	Table of Contents
√	List of Figures (if applicable)
√	List of Tables (if applicable)
	List of Symbols (if applicable)
√	List of Abbreviations (if applicable)
√	Chapters / Content
√	Bibliography (or References)
√	All references in bibliography are cited in the thesis, especially in the chapter of literature review
	Appendices (if applicable)
√	Poster
√	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)

*Include this form (checklist) in the thesis (Bind together as the last page)

I, the author, have checked and confirmed all the items listed in the table are included in my report. [illegible] ______________________ (Signature of Student) Date: 23 April 2020	Supervisor verification. Report with incorrect format can get 5 mark (1 grade) reduction. [illegible] ______________________ (Signature of Supervisor) Date: 22 April 2020