

# Rescue Robot

Sheikh Muhammad Adib bin Sh Abu  
Bakar  
*Robohood*

Soest, Germany  
sheikh-muhammad-adib.bin-sh-abu-  
bakar@stud.hshl.de

Hadi Imran bin Md Radzi  
*Robohood*

Soest, Germany  
hadi-imran.bin-md-radzi@stud.hshl.de

Zafirul Izzat bin Mohamad Zaidi  
*Robohood*

Soest, Germany  
zafirul-izzat-bin.mohamad-  
zaidi@stud.hshl.de

Giwa Habeeb Rilwan  
*Robohood*

Hamm, Germany  
habeeb-rilwan.giwa@stud.hshl.de

Ammar Haziq bin Mohd Halim  
*Robohood*

Soest, Germany  
ammar-haziq.bin-mohd-  
halim@stud.hshl.de

Adijat Ajoke Sulaimon  
*Robohood*

Hamm, Germany  
Adijat-ajoke.sulaimon@stud.Hshl.de

**Abstract—** The number and scale of disasters that arise out of natural and human causes has been unprecedented. In the aftermath of these tragedies, the invention and development of future search and rescue technology has resulted in significant suffering for many victims. Search and rescue robots are one of the most important areas of application within this scope. Researchers and rescuers draw ever more attention to robots equipped with sensing and maneuvering abilities in the area of disasters. This project aims to create a new generation of search and rescue robot that can work independently and semi-autonomously and can be used more effectively through the use of advanced and economic sensors in the harsh physical conditions of disaster regions. This article describes our effort to build a rescue robot from the ground up. All diagrams will be clearly and methodically discussed. After meeting all of the requirements of the diagrams, this article will illustrate how we designed our robot so that it can travel not only on land but also on water. This post would also demonstrate our approach to coding by starting with an activity diagram. Figures are also added to each section to help the reader gain a basic and more in-depth knowledge. Last but not least, the reader may discover more information on our rescue robot project in the appendix section at the end of the article.

**Keywords—**rescue, robot

## I. MOTIVATION

We will start by defining what is a rescue robot and describing the issues that calls the need for a smart rescue robot. So, what is a rescue robot? A rescue robot is defined as a semi-autonomous or fully-autonomous robot that is capable of search and rescue missions in specific situations. The examples of such situations are in earthquakes, building fires and drowning situation in seas or lakes. The general purpose of the rescue robot is for assisting rescue teams making it easier to complete their mission. There are several qualities or features that makes a robot a rescue robot. In our opinion, the basic general requirements of a rescue robot must suit the following criteria:

- Must be durable enough to withstand rough terrains.
- Having the ability to move through one or more types of terrain.
- Having the ability to communicate or send important information to rescue teams.
- Having the ability to sense robot surrounding in order to avoid obstacles.

Nowadays, further research and developments has been carried to push this new technology to the market as it will solve a lot of the problems regarding the safety of people.

Although we do have safety related people such as firemen and lifeguards, problems can still arise in a rescue situation, which could cause injury to the civilian (victim) or even cause death. One of the issues that could happen in a rescue situation is that humans are prone to make errors. We as humans are prone to have a poor performance in our task when we are stressed, under a lot of pressure or lack of concentration. Since all humans have their own personal problems in their lives, that means that rescue teams also will have their own personal problems, which could affect their performance during a rescue situation. Having a rescue robot to assist the rescue team could help prevent mistakes from happening during a rescue situation, thus saving more lives in the process.

Ignoring human errors that could occur during a rescue situation, rescue teams also need a lot of time to act in an emergency to ensure the safety of the majority of the people. They would need to set up safety gears and equipment in which during that time could be used to start the rescue mission. On top of that, the rescue team are not entirely invincible to the dangers of the rescue situation. They are also human and they still have the risk of injuring themselves or death while they are trying to save other people. The main mission is to save as much lives as possible in a rescue situation, including the rescue team. With the assistance of a rescue robot, these stated problems are no longer an issue, which could save more lives during an emergency situation.

Rescue robots that possess the qualities mentioned above may be able to solve said problems during safety missions. Firstly, rescue robot that are durable can move in dangerous paths, that would normally consume more time for the average rescue teams to move, without risking lives. Other than that, the robot can also send information regarding trapped civilians that are difficult to locate. For example, in a collapsed building, the robot can move in small holes and tight areas and help rescue teams locate victims that are trapped under a rock. This in turn will help rescue teams save more lives than before, which is the ultimate priority in a rescue situation.

We strive as a group to develop a rescue robot that could solve all these stated problems and provide safety to humans generally. As a result of our development, we managed to add features to our rescue robot that could solve the problems that were stated above.

Our rescue robot has the ability to travel across both land and water. This is extremely beneficial as it is compatible for the rescue team to use, regardless of the rescue situation. Burning buildings, earthquake emergencies, drowning situations, and other rescue situation could be assisted using the same robot. This also makes the robot more versatile in handling the rescue situation.

The mobility of our rescue robot is also very reliable. Not only is it versatile to move both on land and water, it could also break small obstacles if one is present in the situation, and detect humans to aid them during the rescue situation. This is what makes our rescue robot smart and reliable. There are also other functionalities to aid the rescue team which we are going to dive in deeper in the next section.

In summary, it is extremely beneficial to have the assistance of a rescue robot. With the help of a rescue robot, we can shorten the time to rescue a victim in danger, thus increasing the efficiency of a rescue mission and save a lot more lives in a critical situation. We can also avoid human errors and may even help save the lives of the rescue team if something unexpected happen.

## II. SYSTEM ENGINEERING

### A. Introduction

The first and important step into the development process of our rescue robot is to precisely produce the analysis of the system context and define our problem as detail as possible. we need to have an overall description of our whole system before we start working on the robot itself. we modelled our system with the aid of SysML and UML diagrams and this is to help define the concrete scenario and to also establish boundaries between robot and the specific environment. therefore, we can determine the details of the features/abilities of the robot.

### B. Diagram

#### 1) Requirement Diagram

The first diagram that we will look into is requirement diagram. A requirement is defined as a condition, or a capability needed by us user to solve a problem or achieve an objective. In order to determine which requirement is necessary to be implemented in our system and to avoid future waste, we have also included the specifics of the environment that interacts with our robot in our requirement diagram. For example, we included a breaker tool for our robot so that it can go through obstacles like windows with thickness of 4.8 mm. Further details of our final requirement diagram are as shown.

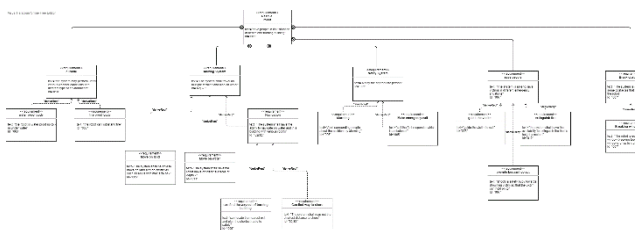


Figure 1: Requirement diagram

An interesting requirement to have a look at is the move system. We have included in our system, the ability for the robot to move on land with the specification that the robot can move height 15 meters and on water with depth of 200 meters. Furthermore, we also equipped our system with a navigation subsystem so that it can find the way out of burning building and find the shortest path to shore.

Another important aspect of our robot that needs to be considered is how the robot will notify the rescue team or a helping bystander when a civilian is in the burning building or drowning in the sea. We have included a notification requirement for our robot so that when an emergency situation is recognized, the robot will sound an alarm alerting its surrounding. Other than that, the robot will also make an emergency call to alert authorities regarding the situation.

In the successive sections, we will provide the reader with scenario related diagrams such as Use case and Sequence diagrams to help visualize how the requirements are planned to be implemented in our rescue robot system. The diagrams mentioned will construct a sequential story for the application of said requirement diagram and how the requirements act with each other and actors that interact with the system.

#### 2) Usecase Diagram

To help us in the development process of the system of the rescue robot in a more effective and efficient way, we first need to define the concrete scenario that the rescue robot is involved in. The important factor in developing our system is determining the boundaries in which our robot interacts with the environment. In the first phase of our project, we have done our research and gathered information on rescue situations around the world. Using the knowledge that we gained about rescue situations, we could create a use case diagram and identify situations that our rescue robot will play a huge role when assisting the rescue team. As a result, we have decided to focus on two main rescue situations, which is burning buildings and drowning victims. The final use case diagram is showcased in figure 2.

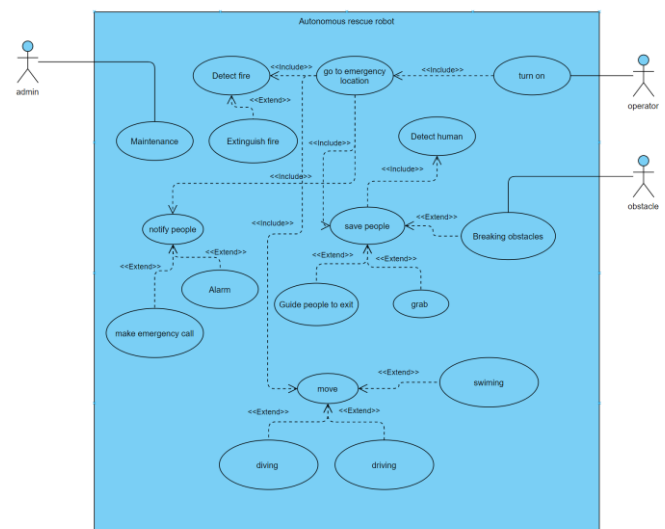


Figure 2: Use case diagram

As you can see from the use case diagram, the operator will start the robot by turning the robot on in order for it to start operating. The robot will then move towards the emergency location in order to assist the victims of the rescue situation. Our robot will notify people nearby the emergency location with the help of its alarm. The robot will also make an emergency call in case more man power is required to deal with the current emergency situation.

In terms of mobility, our robot can move both on water and on land. It is also smart enough to detect humans, fire and obstacles. Using its intelligent sensor, the robot can navigate its way without being stuck if there is an obstacle in the way. In case it got stuck in between multiple obstacles, it can break the obstacle if it is small and fragile enough such as boulders or window glass. It can also find humans in a burning building, and save them by guiding them to the exit. When it detects fire, it has the ability to extinguish the fire and continue its journey. In a drowning situation however, the robot can grab the victim and save the drowning victim.

### 3) Sequence Diagram

In the following section, we continue our development process with the addition of a sequence diagram. With the sequence diagram, we can see the flow of use of the system in our robot given the context situation. More importantly, we again use the analysis of the requirement and all of the preceding diagrams that we have built so that we can design a sequence diagram for the overall system of the robot. We have developed two sequence diagrams according to their rescue situations, which is burning buildings and drowning situation.

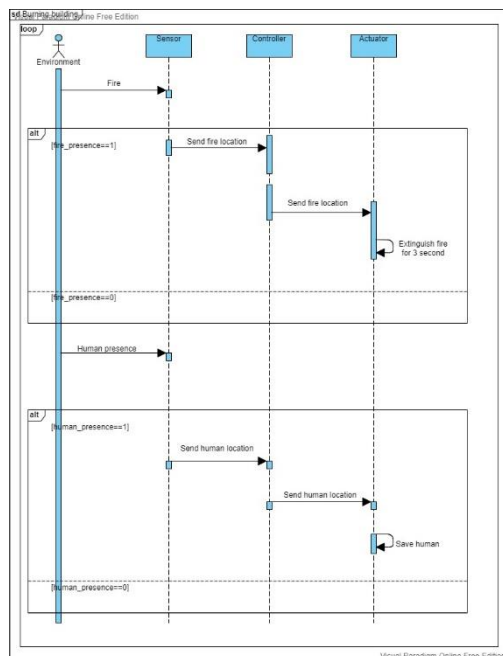


Figure 3: Sequence diagram burning building

Figure 3 shows the sequence diagram of the burning building situation. The environment will send a trigger which is a fire in this case to the sensor of the robot. The robot will

then update the fire\_presence variable depending on whether there is a fire or not nearby the robot. If the fire\_presence is 1, the sensor will send the fire location to the controller of the robot. The controller will then pass the fire location to the actuator, and the actuator will use that information to extinguish the fire for 3 seconds.

Then when the environment sends a signal that a human is nearby, it will send a trigger to the sensor to notify the robot. The sensor will then pass the human location to the controller, and the controller will continue passing the location to the actuator. The actuator then will move and save the human based on the location that it received.

The system will then be looped so that the robot keeps on updating the fire and human presence, thus helping the firemen to extinguish the fire and also helping humans if there are still humans stuck inside a burning building and guiding them to safety.

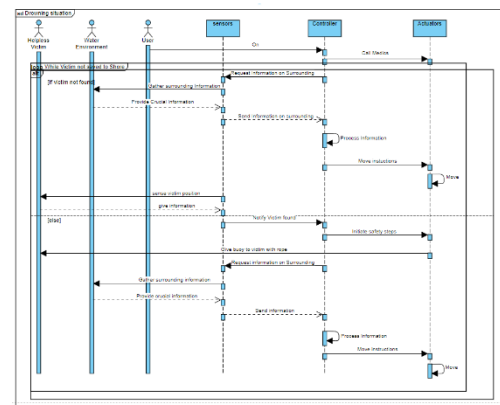


Figure 4: Sequence diagram drowning situation

Figure 4 shows the sequence diagram of a drowning situation. The user or operator will turn the robot on for it to start its operation. It will then start its operation by calling the medics to prepare in case of an emergency after the victim has been saved. It will then enter a loop where it will use its sensors to request information of the surrounding. To be more specific, it will use the information gained from the surrounding to move towards the victim. It will then sense the victim position. If the victim isn't found, it will loop again to move until it finds the victim.

Once the victim is found, it will start initiating safety steps, and shoots a buoy towards the victims that is attached to a rope with the robot. After the robot successfully saved the victim, it will use the same navigating mechanism to find its way back to the shore, and bringing the victim to safety.

### 4) Architecture Diagram

The final diagram that we developed as a result of our previous diagram is our system structure diagram. The purpose of a structure diagram is to describe our system as a whole and give a clear bird's eye view of our system. The type of system structure that we have chosen is layered application. In layered application, the components of the

system are divided into horizontal layers. Each layer of the layered architecture pattern has a specific role and responsibility within the application. We will go in detail for each of the layers in the upcoming paragraph.

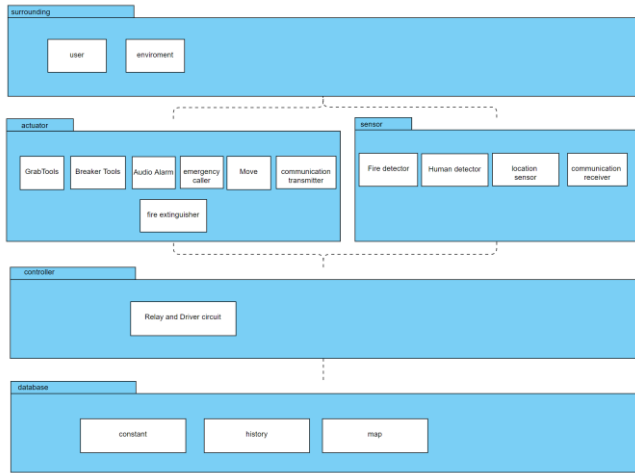


Figure 5: Architecture diagram

The system structure diagram is showcased in figure 5. Our system consists of 4 main layers, which are the surrounding layer, the interface layer, the controller layer and the database layer. In the surrounding layer, we have included 2 components that will interact with our robot, which are user and environment. Both the components will act as an actor that will initiate our robot to do to operate different tasks. For example, the user will turn on the robot for it to start operating, and the environment could be a numerous number of triggers, such as fires, humans, water and obstacles, that could stimulate a response from the robot.

In the next layer, it consists of components from two part of the robot that directly interacts with the environment. There are various components in this layer, so we divided the components into two groups, which are the actuator and the sensor. In the sensor group, the components are responsible for getting information from the environment and passing it to the controller of the system. The components that are responsible for this are fire detector, human detector, location sensor, communication receiver. In the actuator group, the components are responsible for the response of the robot after it receives sets of instructions from the controller. The components that realise this purpose are grab tools, breaker tools, audio alarm, emergency caller, move, communication transmitter and fire extinguisher.

The third and arguably the most important layer of the system is the controller. It serves as the so-called brain of the system, because it is responsible for receiving the information from the sensor, managing or processing said information, and delivering sets of instructions to the corresponding components of the actuators. The controller can be realised by multiple relay and driver circuit.

The fourth and final layer is the database layer. The function of this layer is to store vital data that relates to the robot's activity. We can realise this by a local storage drive in the robot. The information stored can be either the history of the

robot's action or the value of constant variables that has been pre-programmed such as the weight of the robot and the gravitational acceleration.

### III. MODEL PROTOTYPING

#### A. Introduction

Design stage is the most important part of the project. The dimensions of the robot are so important in this stage. The accepted maximum dimension is determined as 250x250mm. But the length of the robot could be longer than the determined 250mm so that to increase its climbing capability. The components that directed the design is divided into two as electronics and mechanics. Because of the perception and charting properties the used sensors, circuits, processors and the covered area and weight of these components are the most important parameters that influence the design. For this reason all required electronic components are determined and table of dimensions and weight is prepared.

Another important subject is the properties and charge duration of the power supply. According to the researches the rechargeable batteries, NiMh or NiCd is preferred. Voltage and the current used in the system is decided by other components' voltage and current used (**Table1**).

According to the usage area and perception property of employed every electronic component and sensor, the layout on the robot is adjudged so the mechanic design begins according to the total weight.

#### B. Mechanical design

According to the requirements of the robot, the robot must walk both on land and water. This is therefore an important consideration made during the design of the robot. We have to assume scenarios, challenges, obstacles, power consumption, constraints and other factors which can affect the movement and overall performance of the robot. These factors and some others are what affects or determine our decision on the design of the robot. It is absolutely important that the system is:

##### 1) Durable

This means that it must perform under all weather conditions and environments. It must be able to withstand water and fire since we want it to be able to save people from burning buildings and people drowning.

##### 2) Moving system

As stated earlier, the system shall travel on different terrains and also on water. Speed and easy navigation are considered to be important.

The first step is to make a sketch. We made a sketch, defining every part according to the assumed scenario and constraints. The first sketches are displayed below:

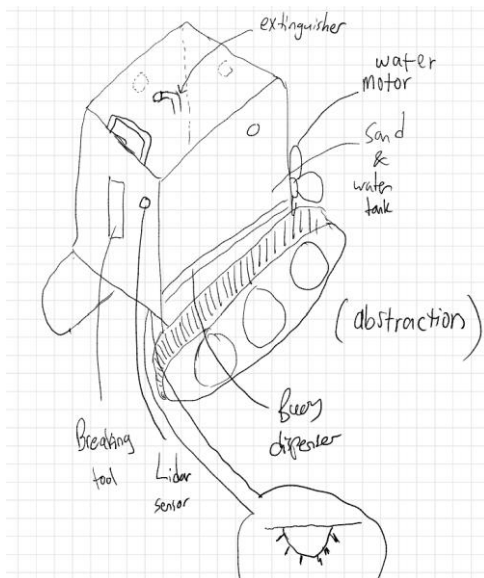


Figure 6: First whole body sketch

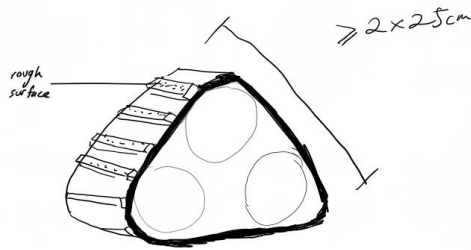


Figure 7: Wheel sketch

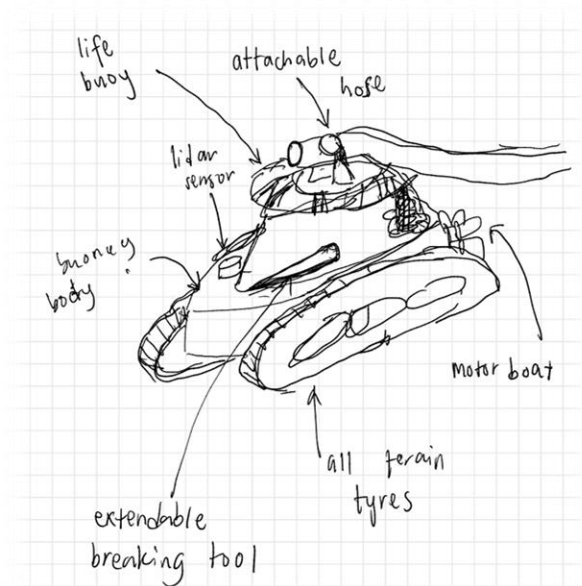


Figure 8: Second and final whole body sketch

## C. Parts

### 1) Moving system

As mentioned earlier, the major requirement of the robot is to move on land and water, therefore, the movement mechanism/system is an important part. We considered and compared different types of movement mechanism. We calculated the biggest obstacle and height we want our robot to climb. We also looked at how fast it can move. Finally, we decided on the rotating continuous track.

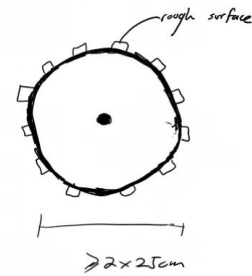


Figure 9: Gear sketch

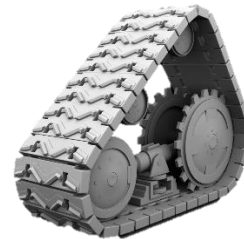


Figure 10: Track 3D model

We chose this mechanism over others due to the following reasons:

- Traction – Even on slippery surfaces like snow or wet ground, it has a high traction.
- Moving on rough terrain – It can move on stones, ditches, stairs and surmount obstacles.
- Power efficiency – It has a high performance and optimized traction system.

There are other reasons why rotating continuous track is better than others, but these are the main reasons we chose it over any other type of moving system.

### 2) Breaking tool

Another requirement is that the robot break some obstacles in its way. The breaking tool should be able to break glass material like windows in a burning building to create an alternative passage. We considered different breaking tools and made different sketches and considered how they will work in some scenarios. Below are some of the initial sketches.



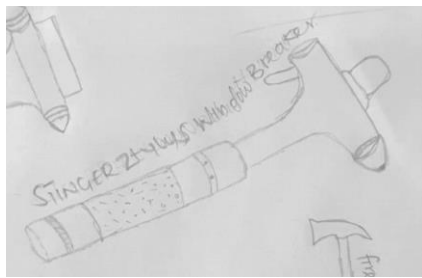


Figure 11: Breaking tool sketch 1



Figure 12: Breaking tool sketch 2



Figure 13: Breaking tool sketch 3

### 3) Life buoy

The life buoy will be shot out by the robot to help people drowning. It will help save people in water. They will hold on to the buoy and will be dragged out of the water. Below is the initial sketch of the life buoy we made.

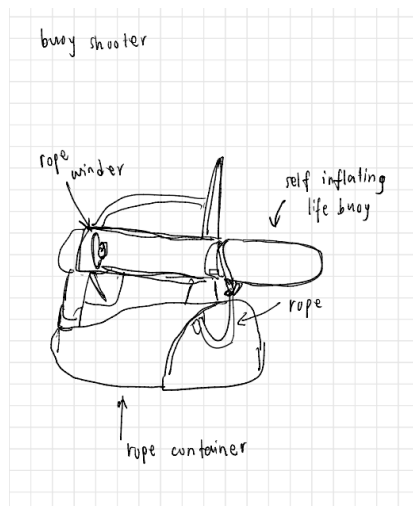


Figure 14: Life buoy sketch

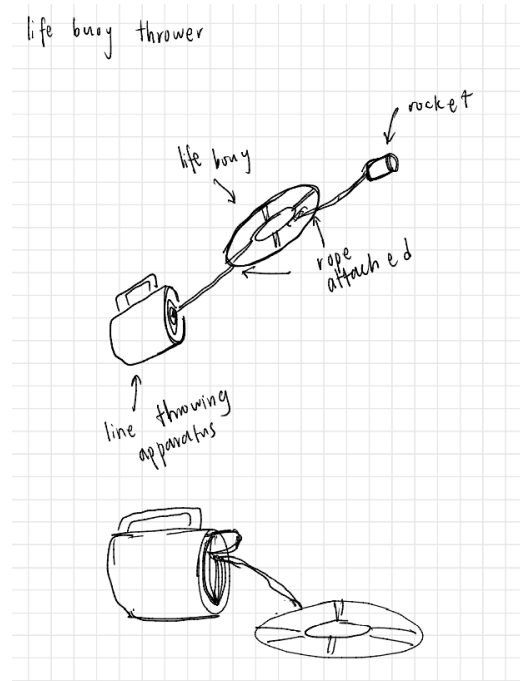


Figure 15: Life buoy thrower sketch

After initial sketches and making decisions on sizes, we did a final sketch with dimensions. We sketched the front and side view of the robot with proper dimensioning. Then we moved on to drawing with solid works in 3D. We drew each part individually and finally connected them together.

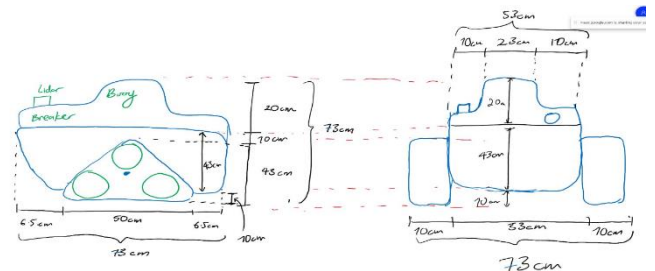


Figure 16: Robot full body sketch with dimensioning

Below is the solid works® 2020.drawing of the parts.

### 4) Propeller

The essence of the propeller is to provide a method of movement. The propeller will provide a thrust force. The propeller itself consists of two or more blades connected together by a central hub that attaches the blades to the shaft.

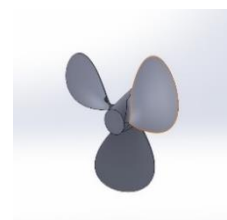


Figure 17: Propeller

### 5) Center gear

The gear is used to transfer or receive motion. In this particular case, it is used to transfer rotational forces between axles. It is sized evenly with spaced and sized teeth.

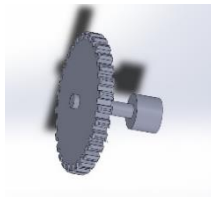


Figure 18: Gear

### 6) Upper body

This serves as a cover for the main body of the robot. It houses the life buoy.



Figure 19: Upper body

### 7) Lower body

This is the main body of the robot. This part will house every other part and component that make up the robot. The wheel or movement system of the robot will also be attached to this part.



Figure 20: Lower body

### 8) Gear housing back

As the name implies, the gear will be attached to this component. This is like shelter for the gear.



Figure 21: Gear housing back

### 9) Rear tyre with shaft

Rear drive gives better traction in uneven paths by increasing the motion. This is part of the movement system of the robot.



Figure 22: Rear tyre with shaft

### 10) Belt

This has lots of functions. It allows the gears to run smoothly with as little noise as possible. A belt transfers rotary motion between two shafts. This is a simple and economical method of motion and power transmission.

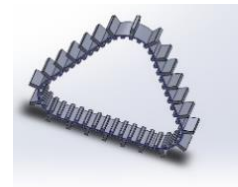


Figure 23: Belt

The next and final phase of the mechanical design is to assemble the parts together. The final look of the robot is in the diagram below.



Figure 24: Final robot model

## IV. IMPLEMENTATION

### A. Introduction

We are now at the next process stage, and we are almost near to build our first rescue robot. In this stage, after construct variety of diagrams in order to fulfil what our rescue robot is required for and also has a solid design and model that we made using solid work. We now move into system implementation stage. The system that we are focus on is moving system to allow the robot to move on a given map to rescue a victim. On this stage, we have 4 different tasks that we need to solve as approach to a reliable moving system. Because of the limitation of time and equipment, we only simulate the moving robot on computer. Basically, what we are going to do is to make sure that our rescue robot can navigate well in real world. Our main goal on this stage is to make sure our rescue robot would be able to move and reach the target with safe and sound. In order to turn dreams in

reality, our rescue robot must be able avoid itself from hitting the wall and can also destroy obstacles. The interesting part is that before the robot moves to the target or destination, it must ensure that it consumes as little energy as possible. As a result, the robot must choose between avoiding barriers and destroying obstacles. Before moving on to the implementation phase, we thoroughly analyse the requirements of each task to identify the best solution. However, for the first solution that we discovered is not the absolute solution that we use for this entire task since we utilize an agile method to have reliable solution.

## B. Approach

Planning the processes to find concrete solution is important part to make sure the development of the solution is on track and progress can be evaluate. We divide our process to four phase as shown in figure 25. For the first step is to analyse the requirement base on the given task. As a group we produce solution that meet the requirement and model it to make sure that all the requirements are fulfilled and divide the task base on the model for implementation. So further validation can be made in short time, thus we can produce concrete solution within a week. Next, we will explain each process phase for each task.

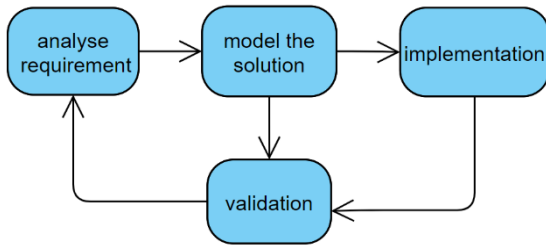


Figure 25: Process development

### 1) Task 1

For the first task we are required to make the robot to able to move to target on a given map without any obstacle. The only thing that the robot needs to do is to be able to move in the direction of north, south, east and west. For more understanding, there are few representations of characters in the codes. Character (O) is to dedicate that the space is empty which is the robot can move freely. Next, wall (#) meaning the robot must be able to avoid it or else the robot will crash and cannot find the target. Other than that, character (T) is the target or in real life the target is known as victim. Last but not least is (R) which is our robot. On this task as mention before, we must utilize our own function to allow the robot to move. Not to mention, in order to avoid our robot failing the assignment, we must ensure that the maximum number of steps, which is 200, is not exceed.

After analyse our requirement, we took a second step ahead by modelling our solution. On this part, we used activity diagram as our references before we jump into implementation part. With the help of activity diagram, we can easily illustrate and visualise clearly what our goals is.

This is also to prevent us from making spaghetti codes. In figure 26 we can see the flow of our activity diagram for the robot to move and reach target. Firstly, with the map given, we find both current location of robot and target. After both locations have been found, the robot will move horizontally. When the distance is more than zero, the robot will move to the left side until it meet target. If target does not find and meet wall, the robot will turn right with distance less than zero. Then, if the target located above or below robot, we will set horizontal distance equal to zero and find vertical distance. The robot will move upward if the distance more than zero and downward if less than zero.

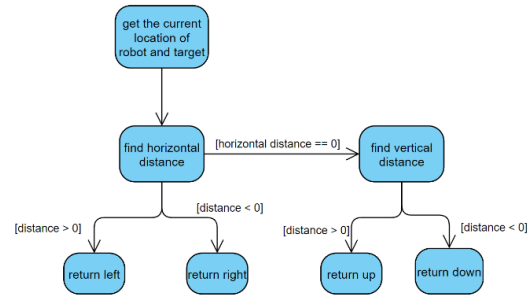


Figure 26

In figure 27 below shows the location of the target and robot. This figure will assist the reader in better understanding specially in the implementation part on how the robot will move and reach the target.

```

char world[200] = {
  '#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#',
  '#','T','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O',
  '#','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O',
  '#','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O',
  '#','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O',
  '#','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O',
  '#','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O',
  '#','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O',
  '#','O','R','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O',
  '#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#',
};
  
```

Figure 27: map task 1

### 2) Task 2

In the second task, we need to implement additional functionality so that the robot may vary its movement strategy based on the surface it is attempting to travel on, whether it is on water or land. At modelling phase in development process, we just expand the current model by adding new function as the model shown in figure 28. The robot will decide whether it need to change the moving method before moving on the next surface.

The implementation for this task is attached in appendix.

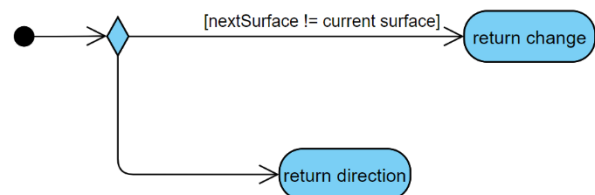


Figure 28



```
char world6[200] = {  
    '#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','\n'  
    '#','O','T','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','\n'  
    '#','~','~','~','~','~','~','~','~','~','~','~','~','~','~','~','~','~','~','\n'  
    '#','~','~','~','~','~','~','~','~','~','~','~','~','~','~','~','~','~','~','\n'  
    '#','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','\n'  
    '#','R','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','O','\n'  
    '#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','#','\n'};
```

### 3) Task 3

The breadth-first search (BFS) technique is used to find the shortest path in a network and normally is used to solve puzzle games. BFS is also an algorithm for searching a tree data structure for a node that meets a certain property. The way of how this algorithm works is why we take BFS as consideration into our implementation. Basic review about BFS is that it starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

First, we add current robot location in queue array. Then we check its neighbour and set the current node as its neighbour parent node and put the neighbour node in queue array. value in n-th element in (parentNode) array store the parent node of node n. We also mark the checked node as visited node, so that no node will be visited more than one time. Wall and obstacle will be skip because the robot cannot move through them. After that we change the current node to the next node in queue. This process will be repeated until the target is found.

arrange the element in (parentNode) to form a path. After that we reverse the arrangement in (reversePath) array to form path array and immediately lock find path activity.

Move function return the direction and it will check whether the robot need to change the moving method by checking the type of current surface and next surface. If the moving method need to be change, this function will return change method instead of the direction. These processes are repeated until the robot reach the target and this activity will be locked immediately.

During the whole process from the initial position of the robot to the target and back to initial position, the energy consumption is recorded in the main function whereas the energy consumption for moving is 10 and for changing the moving method is 30. The figure 30 below depicts a new map that we must use to ensure that the robot reaches the destination.

[illegible]

The implementation for this task is attached in appendix.

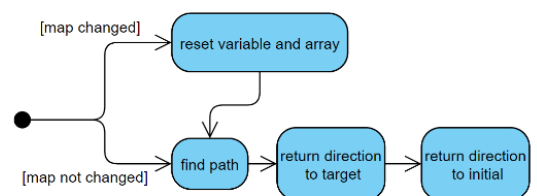


Figure 31

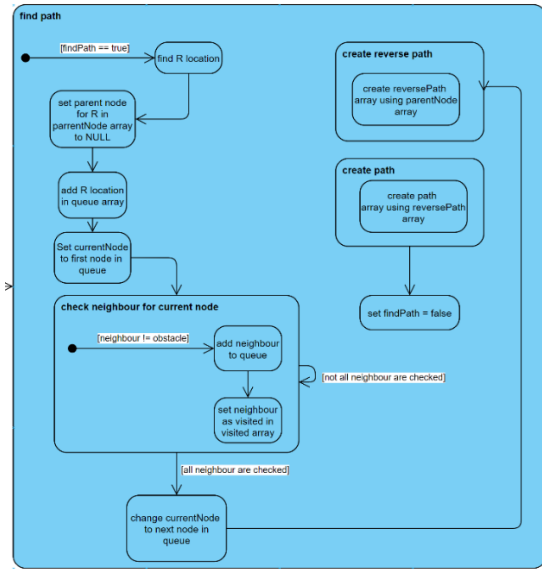


Figure 32

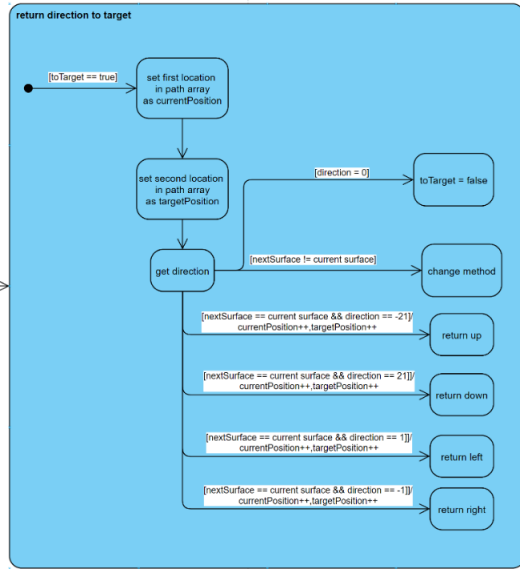


Figure 33

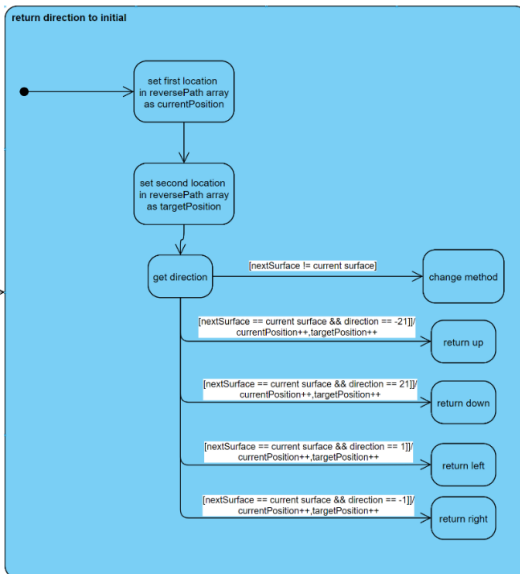


Figure 34

#### 4) Task 4

for the last task we need to add new functionality whereas the robot can destroy obstacles. To fulfil this requirement, we add new option for the robot, and we model the solution as shown in figure 35. So, the robot will check whether there is obstacle on the next surface. If so, this moving function will return destroy instead of direction. Because of the time, we make the robot a bit destructive because we focus on shortest distance to the target rather than less energy consumption. The robot will destroy any obstacle to create the shortest distance to the target. The energy consumption for destroying each obstacle is 70.

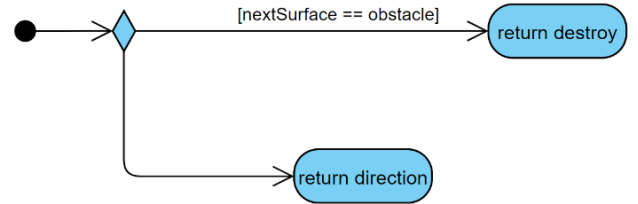


Figure 35

The implementation for this task is attached in appendix.

## V. CONTRIBUTION

### A. Documentation

TABLE I. LINES OF CODE

Member	Contribution	
	Topics	Subtopics
Sheikh Muhammad	Abstract	
	Implementation	Approach
		Task 3
Zafirul Izzat	Abstract	
	Implementation	Task 1
		Task 2
Hadi Imran	Motivation	
	Diagrams	Sequence Diagram
		Usecase Diagram
		System Architecture
Ammar Haziq	Motivation	
	Diagrams	Requirement Diagram
		System Architecture
Giwa Habeeb	Model Prototyping	Introduction
		Mechanical design
		Parts
Ajoke Sulaimon	Model Prototyping	Introduction
		Mechanical design
		Parts

## B. Project development

### 1) System

TABLE II. LINES OF CODE

Member	Contribution
Sheikh Muhammad	System Architecture
Zafirul Izzat	Context Diagram
Hadi Imran	Sequence Diagram
Ammar Haziq	Requirements Diagram
Giwa Habeeb	Block Diagram
Ajoke Sulaimon	Use Case Diagram

### 2) Modeling

Member	Contribution
Sheikh Muhammad	Wheel system
Zafirul Izzat	Upper part body
Hadi Imran	Wheel system
Ammar Haziq	Wheel system
Giwa Habeeb	Breaker tool
Ajoke Sulaimon	Lower part body

### 3) Implementation

TABLE III. LINES OF CODE

Member	Contribution				
	Task 1 move function	Task 2 move function	Task 3 find path algorithm m		Task 4 add destroy obstacle function
Sheikh Muhammad Adib	Participate	Participate	Participate	Participate	Participate
Zafirul Izzat	Participate	Participate	Participate	Participate	
Hadi Imran	Participate	Participate	Participate		
Ammar Haziq	Participate	Participate	Participate		
Giwa Habeeb	Participate	Participate	Participate		
Ajoke Sulaimon	Participate				

## REFERENCES

- [1] Richards, M. (2015). Software architecture patterns. Sebastopol, CA: O'Reilly Media.
- [2] L. Zhu, G. Liu, Y. Zhao, C. Liu, M. Jiang and X. Li, "Design of Motion Control System of Rescue Robot Based on ARM," 2017 International Conference on Computer Technology, Electronics and Communication (ICCTEC), 2017, pp. 1183-1186, doi: 10.1109/ICCTEC.2017.00257.
- [3] A. Denker and M. C. İşeri, "Design and implementation of a semi-autonomous mobile search and rescue robot: SALVOR," 2017 International Artificial Intelligence and Data Processing Symposium (IDAP), 2017, pp. 1-6, doi: 10.1109/IDAP.2017.8090184.

## VI. AKNOWLEDGEMENTACKNOWLEDGEMENT

We are heartily and most grateful to our beloved professors, Prof. Stefan Henkler, Prof. Kristian Rother and Prof. Gido Wahrmann whose inspiration, encouragement, guidance, and support in the beginning towards the final level enabled us to develop our own rescue robot that could help save millions of lives. We would also like to express our respects and regards and benefits to any or all of individuals who supported us whatsoever throughout the completion of the work. For additional information, we would like to express that in this paper, every person in our group has their own responsibility starting from developing the diagrams until writing the documentation and not to forget the implementation code. Every creative decision was discussed and agreed with all team members during our team meeting. We tried our best to make everyone feel involved in the project.

## VII. APPENDIX

### A. Task 1

#### 1) header file

```
#include <stdio.h>
int leftHistory = 0, rightHistory = 0;
int rightLock = 0, leftLock = 0;
int upHistory = 0, downHistory = 0;
int upLock = 0, downLock = 0;
int verticalLock = 0, horizontalLock = 0;

int moveOld(char* world)
{
    int TLocation = 0;
    int RLocation = 0;
    int horizontalTLocation;
    int verticalTLocation;
    int horizontalRLocation;
    int verticalRLocation;

    //get T location
    for (; world[TLocation] != 'T'; TLocation++) {}

    //get R location
    for (; world[RLocation] != 'R'; RLocation++) {}

    //calculate horizontal T position
    horizontalTLocation = TLocation % 21;
    //calculate vertical T distance
    verticalTLocation = (TLocation / 21) + 2;

    //calculate horizontal R position
    horizontalRLocation = RLocation % 21;
    //calculate vertical R distance
    verticalRLocation = (RLocation / 21) + 2;

    //find horizontal direction
    if ((horizontalRLocation - horizontalTLocation) > 0)
    {
        return 4;
    }
}
```

```

if ((horizontalRLocation - horizontalTLocation) < 0)
{
    return 2;
}

//find vertical direction
if ((verticalRLocation - verticalTLocation) < 0)
{
    return 3;
}
if ((verticalRLocation - verticalTLocation) > 0)
{
    return 1;
}

return 0;
}

int move(char* world)//warning : (world[robot_index] = 'O')
need to be changed;
{
    int TLocation = 0;
    int RLocation = 0;
    int horizontalTLocation;
    int verticalTLocation;
    int horizontalRLocation;
    int verticalRLocation;

    //get T location
    for (; world[TLocation] != 'T'; TLocation++) { }

    //get R location
    for (; world[RLocation] != 'R'; RLocation++) { }

    //calculate horizontal T position
    horizontalTLocation = TLocation % 21;
    //calculate vertical T distance
    verticalTLocation = (TLocation / 21) + 2;

    //calculate horizontal R position
    horizontalRLocation = RLocation % 21;
    //calculate vertical R distance
    verticalRLocation = (RLocation / 21) + 2;

    restart:

    //find vertical direction
    if (((verticalRLocation - verticalTLocation) > 0) &&
        verticalLock == 0)//north
    {
        if ((world[RLocation - 21] == '#' || world[RLocation - 21]
            == 'x'))
        {
            //need to move down
            if ((world[RLocation - 1] == '#' || world[RLocation -
                1] == 'x') && (world[RLocation + 1] == '#' ||
                world[RLocation - 1] == 'x'))
            {
                world[RLocation] = 'x';//dummy wall
                //reset
                leftHistory = 0;
                rightHistory = 0;
            }
        }
    }
}

```

```

rightLock = 0;
leftLock = 0;
return 3;//south
}

//stuck at left coner
if (world[RLocation - 1] == '#' && world[RLocation
+ 1] == 'O' && rightHistory == 0)
{
    rightHistory = 1;//countinue moving to right for
next step
    leftHistory = 0;
    world[RLocation] = 'x';//dummy wall
    leftLock = 1; // dont move to the left for the next
step
    rightLock = 0;// only moving to right is
}
//stuck at right coner
if (world[RLocation + 1] == '#' && world[RLocation
- 1] == 'O' && leftHistory == 0)
{
    leftHistory = 1;//countinue moving to left for next
step
    rightHistory = 0;
    world[RLocation] = 'x';//dummy wall
    rightLock = 1; // dont move to the right for the next
step
    leftLock = 0;// only moving to left is
}
//move to left or right
if (leftHistory == 0 && rightLock == 0)
{
    rightHistory = 1;
    return 2;//east
}
if (rightHistory == 0 && leftLock == 0)
{
    leftHistory = 1;
    return 4;
}
}
leftHistory = rightHistory = 0;

}
else if ((verticalRLocation - verticalTLocation) < 0 &&
verticalLock == 0)//south
{
    if ((world[RLocation + 21] == '#' || world[RLocation +
        21] == 'x'))
    {
        //need to move up
        if ((world[RLocation - 1] == '#' || world[RLocation -
            1] == 'x') && (world[RLocation + 1] == '#' ||
            world[RLocation + 1] == 'x'))
        {
            world[RLocation] = 'x';//dummy wall
            //reset
            leftHistory = 0;
            rightHistory = 0;
            rightLock = 0;
            leftLock = 0;
            return 1;//north
        }
    }
}

```

```

    }
    //stuck at left coner
    if (world[RLocation - 1] == '#' && world[RLocation
+ 1] == 'O' && rightHistory == 0)
    {
        rightHistory = 1;//countinue moving to right for
next step
        leftHistory = 0;
        world[RLocation] = 'x';//dummy wall
        leftLock = 1;// dont move to the left for the next
step
        rightLock = 0;// only moving to right is
    }
    //stuck at right coner
    if (world[RLocation + 1] == '#' && world[RLocation
- 1] == 'O' && leftHistory == 0)
    {
        leftHistory = 1;//countinue moving to left for next
step
        rightHistory = 0;
        world[RLocation] = 'x';//dummy wall
        rightLock = 1;// dont move to the right for the next
step
        leftLock = 0;// only moving to left is
    }
    //move to left or right
    if (leftHistory == 0 && rightLock == 0)
    {
        rightHistory = 1;
        return 2;//east
    }
    if (rightHistory == 0 && leftLock == 0)
    {
        leftHistory = 1;
        return 4;
    }
    leftHistory = rightHistory = 0;
    return 3;//south
}
else if (verticalLock&&horizontalLock)
{
    horizontalLock = 0;
}
else
{
    verticalLock = 1;
    printf("lock");
}
//find horizontal direction
if ((horizontalRLocation - horizontalTLocation) <
0&&horizontalLock==0)//east
{
    if (world[RLocation + 1] == '#' || world[RLocation + 1]
== 'x')
    {
        //need to move to west
        if ((world[RLocation - 21] == '#' || world[RLocation -
21] == 'x') && (world[RLocation + 21] == '#' ||
world[RLocation - 21] == 'x'))
        {

```

```

        world[RLocation] = 'x';//dummy wall
        //reset
        upHistory = 0;
        downHistory = 0;
        upLock = 0;
        downLock = 0;
        printf("need to move to west");
        return 2;//east
    }
    //notes are not updated after copy
    //stuck at upper coner
    if (world[RLocation - 21] == '#' && world[RLocation
+ 21] == 'O' && downHistory == 0)
    {
        downHistory = 1;//countinue moving to
upHistory = 0;
        world[RLocation] = 'x';//dummy wall
        upLock = 1;// dont move to the
downLock = 0;// only moving to
        printf("stuck at upper right conner");
    }
    //stuck at lower coner
    if (world[RLocation + 21] == '#' &&
world[RLocation - 21] == 'O' && leftHistory == 0)
    {
        upHistory = 1;//countinue moving to
downHistory = 0;
        world[RLocation] = 'x';//dummy wall
        downLock = 1;// dont move to
upLock = 0;// only moving to
        printf("stuck at down right conner");
    }
    //move to left or right
    if (upHistory == 0 && downLock == 0)
    {
        downHistory = 1;
        printf("move downward ");
        return 3;//south
    }
    if (downHistory == 0 && upLock == 0)
    {
        upHistory = 1;
        printf("move upward ");
        return 1;//north
    }
}
upHistory = downHistory = 0;
printf("move east ");
return 2;//east
}
else if ((horizontalRLocation - horizontalTLocation) >
0&&horizontalLock == 0)//west
{
    if (world[RLocation - 1] == '#' || world[RLocation - 1]
== 'x')
    {
        //need to move to west
        if ((world[RLocation - 21] == '#' || world[RLocation -
21] == 'x') && (world[RLocation + 21] == '#' ||
world[RLocation - 21] == 'x'))
        {
            world[RLocation] = 'x';//dummy wall

```



```

//reset
upHistory = 0;
downHistory = 0;
upLock = 0;
downLock = 0;
return 2;//east
}
//notes are not updated after copy
//stuck at upper coner
if (world[RLocation - 21] == '#' && world[RLocation
+ 21] == 'O' && downHistory == 0)
{
    downHistory = 1;//countinue moving to
    upHistory = 0;
    world[RLocation] = 'x';//dummy wall
    upLock = 1;// dont move to the
    downLock = 0;// only moving to
}
//stuck at lower coner
if (world[RLocation + 21] == '#' &&
world[RLocation - 21] == 'O' && leftHistory == 0)
{
    upHistory = 1;//countinue moving to
    downHistory = 0;
    world[RLocation] = 'x';//dummy wall
    downLock = 1;// dont move to the
    upLock = 0;// only moving to
}
//
if (upHistory == 0 && downLock == 0)
{
    downHistory = 1;
    return 3;//south
}
if (downHistory == 0 && upLock == 0)
{
    upHistory = 1;
    return 11;//north
}
}
upHistory = downHistory = 0;
return 4;//west
}
else
{
    verticalLock = 0;
    horizontalLock = 1;
    goto restart;
}
}

```

## B. Task 2

### 1) Source file

```

#include <stdio.h>
#include "robohood.h"

void Avoid(int avoidLocation)
{

```

```

    avoid[counter] = avoidLocation; //mark location that
    surrounded by obstacle
    counter++;
}

int AvoidLocationChecker(int location)
{
    for (int count = 0; avoid[count] != '\0'; count++)
    {
        if (location == avoid[count])
        {
            return 1;//(cannot go to the location)
        }
    }
    return 0;
}

int ifToChange(char nextSurface)
{
    if (nextSurface == '~' && onLand == 1)//change to move
    on water mood
    {
        onLand = 0;
        onWater = 1;
        return CHANGE;
    }
    if (nextSurface == 'O' && onWater == 1)//change to move
    on land mood
    {
        onLand = 1;
        onWater = 0;
        return CHANGE;
    }
    return 0;//no change
}

int move(char* world)
{
    int TLocation = 0;
    int RLocation = 0;
    int horizontalTLocation;
    int verticalTLocation;
    int horizontalRLocation;
    int verticalRLocation;
    //get T and R location
    for (; world[TLocation] != 'T'; TLocation++) {}
    for (; world[RLocation] != 'R'; RLocation++) {}

    //calculate horizontal and vertical T position
    horizontalTLocation = TLocation % 21;
    verticalTLocation = (TLocation / 21) + 2;

    //calculate horizontal and vertical R position
    horizontalRLocation = RLocation % 21;
    verticalRLocation = (RLocation / 21) + 2;

    //find vertical direction
    //north direction
    start:

```

```

    if (((verticalRLocation - verticalTLocation) > 0) &&
vLock == 0)//north
    {
        printf("north\n");
        if (world[RLocation - 21] == '#' ||
AvoidLocationChecker(RLocation - 21))//if there is obstacle
infront
        {
            //need to move down
            if ((world[RLocation - 1] == '#' ||
AvoidLocationChecker(RLocation - 1)) &&
(world[RLocation + 1] == '#' ||
AvoidLocationChecker(RLocation + 1)))
            {
                Avoid(RLocation);//mark the location as avoid
location
                //reset
                moveLeft = 0;
                moveRight = 0;
                //check for method change
                if (ifToChange(world[RLocation + 21]))
                {
                    return 5;
                }
                printf(" down\n");
                return south;
            }
            //stuck at left coner
            else if ((world[RLocation - 1] == '#' ||
AvoidLocationChecker(RLocation - 1)) &&
(world[RLocation + 1] == 'O' || world[RLocation + 1] == '~'))
            {
                moveRight = 1;//countinue moving to right for next
step
                moveLeft = 0;
                Avoid(RLocation);//mark the location as avoid
location
            }
            //stuck at right coner
            else if ((world[RLocation + 1] == '#' ||
AvoidLocationChecker(RLocation + 1)) &&
(world[RLocation - 1] == 'O' || world[RLocation - 1] == '~'))
            {
                moveLeft = 1;//countinue moving to left for next
step
                moveRight = 0;
                Avoid(RLocation);//mark location as void location
            }
            //move left option
            else if (world[RLocation - 1] != '#' ||
AvoidLocationChecker(RLocation - 1) != 1)
            {
                moveLeft = 1;
            }
            //move right option
            else if (world[RLocation + 1] != '#' ||
AvoidLocationChecker(RLocation + 1) != 1)
            {
                moveRight = 1;
            }

```

```

//action(left or right)
if (moveRight == 1)
{
    //check for method change
    if (ifToChange(world[RLocation + 1]))
    {
        return 5;
    }
    printf(" right\n");
    return east;
}
else if (moveLeft == 1)
{
    //check for method change
    if (ifToChange(world[RLocation - 1]))
    {
        return 5;
    }
    printf(" left\n");
    return west;
}
}
//default when there is no obstacle
hLock = 0;
moveLeft = 0;
moveRight = 0;
//check for method change
if (ifToChange(world[RLocation - 21]))
{
    return 5;
}
printf(" up\n");
return north;
}

//moving south
else if ((verticalRLocation - verticalTLocation) < 0 &&
vLock == 0)//south
{
    printf("south\n");
    if (world[RLocation + 21] == '#' ||
AvoidLocationChecker(RLocation + 21))//if there is obstacle
at bottom
    {
        //need to move up
        if ((world[RLocation + 1] == '#' ||
AvoidLocationChecker(RLocation + 1)) &&
(world[RLocation - 1] == '#' ||
AvoidLocationChecker(RLocation - 1)))
        {
            Avoid(RLocation);//mark location as void location
            //reset
            moveLeft = 0;
            moveRight = 0;
            //check for method change
            if (ifToChange(world[RLocation - 21]))
            {
                return 5;
            }
            return north;

```

```

    }
    //stuck at left coner
    else if ((world[RLocation + 1] == '#' ||
AvoidLocationChecker(RLocation + 1)) &&
(world[RLocation - 1] == 'O' || world[RLocation - 1] == '~'))
    {
        moveRight = 0;//countinue moving to right for next
step
        moveLeft = 1;
        Avoid(RLocation);//mark location as void location
    }
    //stuck at right coner
    else if ((world[RLocation - 1] == '#' ||
AvoidLocationChecker(RLocation - 1)) &&
(world[RLocation + 1] == 'O' || world[RLocation + 1] == '~'))
    {
        moveLeft = 0;//countinue moving to left for next
step
        moveRight = 1;
        Avoid(RLocation);//mark location as void location
    }
    //move right option
    else if (world[RLocation + 1] != '#' ||
AvoidLocationChecker(RLocation + 1) != 1)
    {
        moveRight = 1;
    }
    //move left option
    else if (world[RLocation - 1] != '#' ||
AvoidLocationChecker(RLocation - 1) != 1)
    {
        moveLeft = 1;
    }
    //action
    if (moveRight == 1)
    {
        //check for method change
        if (ifToChange(world[RLocation + 1]))
        {
            return 5;
        }
        return east;
    }
    else if (moveLeft == 1)
    {
        //check for method change
        if (ifToChange(world[RLocation - 1]))
        {
            return 5;
        }
        return west;
    }
}
//default when there is no obstacle
moveLeft = 0;
moveRight = 0;
hLock = 0;
//check for method change
if (ifToChange(world[RLocation + 21]))
{
    return 5;
}

```

```

    }
    return south;
}
//temporary lock for vertical movement algorithm
else
{
    vLock = 1;
}

//find horizontal direction
//move to right
if ((horizontalRLocation - horizontalTLocation) < 0 &&
hLock == 0)//east
{
    printf("east\n");
    if (world[RLocation + 1] == '#' ||
AvoidLocationChecker(RLocation + 1))//if there is obstacle
on right
    {
        //need to move left
        if ((world[RLocation - 21] == '#' ||
AvoidLocationChecker(RLocation - 21)) &&
(world[RLocation + 21] == '#' ||
AvoidLocationChecker(RLocation + 21)))
        {
            Avoid(RLocation);//mark location as void location
            //reset
            moveUp = 0;
            moveDown = 0;
            //check for method change
            if (ifToChange(world[RLocation - 1 ]))
            {
                return 5;
            }
            return west;
        }
        //stuck at upper right conner
        else if ((world[RLocation - 21] == '#' ||
AvoidLocationChecker(RLocation - 21)) &&
(world[RLocation + 21] == 'O' || world[RLocation + 21] ==
 '~'))
        {
            moveDown = 1;//countinue moving down for next
step
            moveUp = 0;
            Avoid(RLocation);//mark location as void location
        }
        //stuck at lower right coner
        else if ((world[RLocation + 21] == '#' ||
AvoidLocationChecker(RLocation + 21)) &&
(world[RLocation - 21] == 'O' || world[RLocation - 21] ==
 '~'))
        {
            moveUp = 1;//countinue moving up for next step
            moveDown = 0;
            Avoid(RLocation);//mark location as void location
        }
        //move up option
        else if (world[RLocation + 21] != '#' ||
AvoidLocationChecker(RLocation + 21) != 1)
        {

```

```

        moveUp = 1;
    }
    //move down option
    else if (world[RLocation - 21] != '#' ||
AvoidLocationChecker(RLocation - 21) != 1)
    {
        moveDown = 1;
    }

    //action
    if (moveDown == 1)
    {
        //check for method change
        if (ifToChange(world[RLocation + 21]))
        {
            return 5;
        }
        return south;
    }
    else if (moveUp == 1)
    {
        //check for method change
        if (ifToChange(world[RLocation - 21]))
        {
            return 5;
        }
        return north;
    }
}
//default when there is no obstacle
moveUp = 0;
moveDown = 0;
vLock = 0;
//check for method change
if (ifToChange(world[RLocation + 1]))
{
    return 5;
}
return east;
}
//move to left
else if ((horizontalRLocation - horizontalTLocation) > 0
&& hLock == 0)//west
{
    printf("west\n");
    if (world[RLocation - 1] == '#' ||
AvoidLocationChecker(RLocation - 1))//if there is obstacle
on left
    {
        //need to move right
        if ((world[RLocation - 21] == '#' ||
AvoidLocationChecker(RLocation - 21)) &&
(world[RLocation + 21] == '#' ||
AvoidLocationChecker(RLocation + 21)))
        {
            Avoid(RLocation);//mark location as void location
            //reset
            moveUp = 0;
            moveDown = 0;
            //check for method change
            if (ifToChange(world[RLocation + 1]))
            {

```

```

            return 5;
        }
        return east;
    }
    //stuck at upper left conner
    else if ((world[RLocation - 21] == '#' ||
AvoidLocationChecker(RLocation - 21)) &&
(world[RLocation + 21] == 'O' || world[RLocation + 21] ==
'~'))
    {
        moveDown = 1;//countinue moving down for next
step
        moveUp = 0;
        Avoid(RLocation);//mark location as void location
    }
    //stuck at lower left coner
    else if ((world[RLocation + 21] == '#' ||
AvoidLocationChecker(RLocation + 21)) &&
(world[RLocation - 21] == 'O' || world[RLocation - 21] ==
'~'))
    {
        moveUp = 1;//countinue moving up for next step
        moveDown = 0;
        Avoid(RLocation);//mark location as void location
    }
    //move down option
    else if (world[RLocation + 21] != '#' ||
AvoidLocationChecker(RLocation + 21) != 1)
    {
        moveDown = 1;
    }
    //move up option
    else if (world[RLocation - 21] != '#' ||
AvoidLocationChecker(RLocation - 21) != 1)
    {
        moveUp = 1;
    }
    if (moveDown == 1)
    {
        //check for method change
        if (ifToChange(world[RLocation + 21]))
        {
            return 5;
        }
        return south;
    }
    else if (moveUp == 1)
    {
        //check for method change
        if (ifToChange(world[RLocation - 21]))
        {
            return 5;
        }
        return north;
    }
}
//default when there is no obstacle
moveUp = 0;
moveDown = 0;
vLock = 0;
//check for method change

```

```

    if (ifToChange(world[RLocation - 1]))
    {
        return 5;
    }
    return west;
}
//temporary lock for horizontal move algorithm
else
{
    hLock = 1;
}

//if deadlock occur
if (vLock == 1 && hLock == 1 )
{

```

```

printf("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
    vLock = 0;
    hLock = 0;
    moveLeft = 0;
    moveRight = 0;
    moveUp = 0;
    moveDown = 0;
    goto start;
}
}

```

### C. Task 3

#### 1) Source file

```

#include "robohood.h"

void initialize()
{
    printf("\n -----> initializing for
map : %d <----- ", map);
    for (int i = 0; i < 200; i++)
    {
        visitedNode[i] = 0;
    }
    for (int i = 0; i < 200; i++)
    {
        parentNode[i] = 0;
    }
    for (int i = 0; i < 200; i++)
    {
        queue[i] = 0;
    }
    for (int i = 0; i < 200; i++)
    {
        path[i] = 0;
    }
    for (int i = 0; i < 200; i++)
    {
        returnPath[i] = 0;
    }

    found = 0;
    counter = 1;

```

```

currentNode = 0;
indexQ = 0;
Path = 0;
goTarget = 0;
goInitial = 0;
currentPosition = 0;
nextPosition = 0;
next = 1;
current = 0;
direction = 0;
TargetNode = 0;
currentSurface = 0;
}

```

```

void check()
{
    for (int i = 0; i < 200; i++)
    {
        printf("\n      visitednode      %d      ",
visitedNode[i]);
    }
    for (int i = 0; i < 200 ; i++)
    {
        printf("\n      parentnode      %d      ",
parentNode[i]);
    }
    for (int i = 0; queue[i] != '\0'; i++)
    {
        printf("\n queue %d ", queue[i]);
    }
}

void findReversePath()
{
    returnPath[0] = TargetNode;
    int CurrentNode = TargetNode;

    for (int x = 1; CurrentNode != RobotNode; x++) {
        returnPath[x] = parentNode[CurrentNode];
        CurrentNode = returnPath[x];
    }
}

void findPath()
{
    int FoundIndex;
    path[0] = RobotNode;
    int temp = '\0';

    for (int i = 0; returnPath[i] != 0; i++)
    {
        if (returnPath[i] == RobotNode)
        {
            FoundIndex = i;
            break;
        }
    }

    for (int j = FoundIndex, k = 0; j >= 0; j--, k++)
    {
        temp = returnPath[j];
        path[k] = temp;
    }
}

```



```

    }
}
int move(char* world, int map_id)
{
    if (map_id != map)
    {
        map = map_id;
        initialize();
    }
    //find path
    if (Path == 0)
    {
        initialize();
        //find R and T location
        for (; world[currentNode] != 'R';
currentNode++) {
            RobotNode = currentNode;
            for (; world[TargetNode] != 'T' &&
TargetNode < 147; TargetNode++) {
                if (TargetNode == 147)
                {
                    TargetNode = 0;
                    for (; world[TargetNode] != 't'
&& TargetNode < 147; TargetNode++) {
                        }
                    //initial positio parent is null(robot
position)
                    parentNode[currentNode] = '\0';
                    //put current node into que node
                    queue[indexQ++] = currentNode;

                    while (target != found)
                    {
                        //set current node as visited note
                        visitedNode[currentNode] =

visited;

                        //check neighbour
                        for (int direction = 0; direction <

4; direction++)
                        {
                            if (world[currentNode +
neighbour[direction]] != '#' && world[currentNode +
neighbour[direction]] != '*' && visitedNode[currentNode +
neighbour[direction]] == notVisited)
                            {
                                //put neighbour
                                in queue
                                queue[indexQ++] =
currentNode +
neighbour[direction];

                                //set node to
                                visited node
                                visitedNode[currentNode + neighbour[direction]] =
visited;

                                //store parent
                                node for neighbour node

```

```

        parentNode[currentNode + neighbour[direction]] =
currentNode;

        //stop if T is
        found
        if
        (world[currentNode + neighbour[direction]] == 'T' ||
world[currentNode + neighbour[direction]] == 't')
        {
            found
            = 1;
            break;
        }
    }
    //move to next node in queue
    currentNode = queue[counter++];

    }
    findReversePath();
    findPath();
    Path = 1;
}
//debug
/*
for (int i = 0; i < 200; i++)
{
    printf("\n returnPath %d\n", returnPath[i]);
}*/

//Return direction to robot to go to target
if (goTarget == 0)
{
    currentPosition = path[current++];
    printf("\n current position %d",
currentPosition);
    nextPosition = path[next++];
    //printf("\n next %d", next);
    //printf("\n next %d", next-1);

    printf("\n next position %d",
nextPosition);
    direction = currentPosition - nextPosition;
    printf("\n direction %d", direction);
    if (direction == -1)
    {
        printf("\n right \n");
        if ((currentSurface == 0 &&
world[path[next - 1]] == '~') || (currentSurface == 1 &&
world[path[next - 1]] == 'O'))
        {
            if (currentSurface == 0)
            {
                currentSurface
                = 1;
            }
            else if (currentSurface
            {
                currentSurface
                = 0;
            }
        }
    }
}

```

```

        }
        next -= 1;
        current -= 1;
        return 5;
    }
    return 2;
}
else if (direction == 1)
{
    printf("\n left \n");
    if ((currentSurface == 0 &&
world[path[next - 1]] == '~') || (currentSurface == 1 &&
world[path[next - 1]] == 'O'))
    {
        if (currentSurface == 0)
        {
            currentSurface
= 1;
        }
        else if (currentSurface
== 1)
        {
            currentSurface
= 0;
        }
        next -= 1;
        current -= 1;
        return 5;
    }
    return 4;
}
else if (direction == -21)
{
    printf("\n below \n");
    if ((currentSurface == 0 &&
world[path[next - 1]] == '~') || (currentSurface == 1 &&
world[path[next - 1]] == 'O'))
    {
        if (currentSurface == 0)
        {
            currentSurface
= 1;
        }
        else if (currentSurface
== 1)
        {
            currentSurface
= 0;
        }
        next -= 1;
        current -= 1;
        return 5;
    }
    return 3;
}
else if (direction == 21)
{
    printf("\n up \n");
    if ((currentSurface == 0 &&
world[path[next - 1]] == '~') || (currentSurface == 1 &&
world[path[next - 1]] == 'O'))
    {
        if (currentSurface == 0)
        {
            currentSurface
= 1;
        }
        else if (currentSurface
== 1)
        {
            currentSurface
= 0;
        }
        next -= 1;
        current -= 1;
        return 5;
    }
    return 1;
}

goTarget = 1;
// reset for future use
currentPosition = 0;
nextPosition = 0;
current = 0;
next = 1;
}
//Return direction to robot to go to initial position
if (goInitial == 0)
{
    currentPosition = returnPath[current++];
    printf("\n current position %d",
currentPosition);
    nextPosition = returnPath[next++];
    printf("\n next position %d",
nextPosition);
    direction = currentPosition - nextPosition;
    printf("\n direction %d", direction);
    if (direction == -1)
    {
        printf("\n right \n");
        if ((currentSurface == 1 &&
world[nextPosition] == 'O') || (currentSurface == 0 &&
world[nextPosition] == '~'))
        {
            printf("\n
masukkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkg
dfghfghfddghjh \n");
        }
        if (currentSurface == 0)
        {
            currentSurface
= 1;
        }
        else if (currentSurface
== 1)
        {
            printf("\n
masukkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk
\n");
            currentSurface
= 0;
        }
    }
}

```

```

        next -= 1;
        current -= 1;
        return 5;
    }
    return 2;
}
else if (direction == 1)
{
    printf("\n left \n");
    if ((currentSurface == 0 &&
world[returnPath[next - 1]] == '~') || (currentSurface == 1 &&
world[returnPath[next - 1]] == 'O'))
    {
        if (currentSurface == 0)
        {
            currentSurface
= 1;
        }
        else if (currentSurface
== 1)
        {
            currentSurface
= 0;
        }
        next -= 1;
        current -= 1;
        return 5;
    }
    return 4;
}
else if (direction == -21)
{
    printf("\n below \n");
    if ((currentSurface == 0 &&
world[returnPath[next - 1]] == '~') || (currentSurface == 1 &&
world[returnPath[next - 1]] == 'O'))
    {
        if (currentSurface == 0)
        {
            currentSurface
= 1;
        }
        else if (currentSurface
== 1)
        {
            currentSurface
= 0;
        }
        next -= 1;
        current -= 1;
        return 5;
    }
    return 3;
}
else if (direction == 21)
{
    printf("\n up \n");
    if ((currentSurface == 0 &&
world[returnPath[next - 1]] == '~') || (currentSurface == 1 &&
world[returnPath[next - 1]] == 'O'))
    {
        if (currentSurface == 0)

```

```

        {
            currentSurface
= 1;
        }
        else if (currentSurface
== 1)
        {
            currentSurface
= 0;
        }
        next -= 1;
        current -= 1;
        return 5;
    }
    return 1;
}
goInitial = 1;
// reset for future use
}
}

```

#### D. Task 4

##### 1) Source file

```

#include "robohood4.h"

void initialize()
{
    printf("\n -----> initializing for
map : %d <----- ",map);
    for (int i = 0; i < 200; i++)
    {
        visitedNode[i] = 0;
    }
    for (int i = 0; i < 200; i++)
    {
        parentNode[i] = 0;
    }
    for (int i = 0; i < 200; i++)
    {
        queue[i] = 0;
    }
    for (int i = 0; i < 200; i++)
    {
        path[i] = 0;
    }
    for (int i = 0; i < 200; i++)
    {
        returnPath[i] = 0;
    }

    found = 0;
    counter = 1;
    currentNode = 0;
    indexQ = 0;
    Path = 0;
    goTarget = 0;
    goInitial = 0;
    currentPosition = 0;
    nextPosition = 0;
}

```

```

        next = 1;
        current = 0;
        direction = 0;
        TargetNode = 0;
        currentSurface = 0;
    }

//to detect obstacle
int ifObstacle(char obstacle)
{
    if (obstacle == '*')
    {
        return 1;
    }
    return 0;
}

//for debugging
void check()
{
    for (int i = 0; i < 200; i++)
    {
        printf("\n    visitednode    %d    ",
visitedNode[i]);
    }
    for (int i = 0; i < 200; i++)
    {
        printf("\n    parentnode    %d    ",
parentNode[i]);
    }
    for (int i = 0; queue[i] != '\0'; i++)
    {
        printf("\n queue %d ", queue[i]);
    }
}

//find path
void findReversePath()
{
    returnPath[0] = TargetNode;
    int CurrentNode = TargetNode;

    for (int x = 1; CurrentNode != RobotNode; x++) {
        returnPath[x] = parentNode[CurrentNode];
        CurrentNode = returnPath[x];
    }
}

//find path (reversed)
void findPath()
{
    int FoundIndex=0;
    path[0] = RobotNode;
    int temp = '\0';

    for (int i = 0; returnPath[i] != 0; i++)
    {
        if (returnPath[i] == RobotNode)
        {
            FoundIndex = i;
            break;
        }
    }

    for (int j = FoundIndex, k = 0; j >= 0; j--, k++)
    {
        temp = returnPath[j];
        path[k] = temp;
    }
}

int move(char* world, int map_id)
{
    if (map_id != map)
    {
        map = map_id;
        initialize();
    }
    //find path
    if (Path == 0)
    {
        initialize();
        //find R and T location
        for (; world[currentNode] != 'R';
currentNode++) {}
        RobotNode = currentNode;

        for (; world[TargetNode] != 'T' &&
TargetNode < 147; TargetNode++) {}
        if (TargetNode == 147)
        {
            TargetNode = 0;
            for (; world[TargetNode] != 't'
&& TargetNode < 147; TargetNode++) {}
        }

        //initial positio parent is null(robot
position)

        parentNode[currentNode] = '\0';
        //put current node into que node
        queue[indexQ++] = currentNode;

        while (target != found)
        {
            //set current node as visited note
            visitedNode[currentNode] =
visited;

            //check neighbour
            for (int direction = 0; direction <
4; direction++)
            {
                if (world[currentNode +
neighbour[direction]] != '#' /* && world[currentNode +
neighbour[direction]] != '*'/* && visitedNode[currentNode
+ neighbour[direction]] == notVisited)
                {
                    //put neighbour
                    in queue

                    queue[indexQ++] =
currentNode +
neighbour[direction];

```

	//set node to		currentSurface
visitedNode	= land;	}	
		} next -= 1;	
visited[currentNode + neighbour[direction]] = visited;		current -= 1;	
	//store parent	return changeMethod;	
node for neighbour node		}	
	parentNode[currentNode + neighbour[direction]] = currentNode;	//detroy obstacle	
		if	
	(ifObstacle(world[nextPosition]))	{	
			next -= 1;
found	//stop if T is		current -= 1;
	if		return destroyEast;
(world[currentNode + neighbour[direction]] == 'T'    world[currentNode + neighbour[direction]] == 't')	{		
		return moveEast;	
	found	}	
= 1;	break;	else if (direction == west)	
		{	
		printf("\n left \n");	
		if ((currentSurface == land && world[nextPosition] == '~')    (currentSurface == water && world[nextPosition] == 'O'))	
		{	
			if (currentSurface ==
			land)
			{
			currentSurface
			== water;
			}
			else if (currentSurface
			{
			currentSurface
			}
			next -= 1;
			current -= 1;
			return 5;
		}	
		if	
		(ifObstacle(world[nextPosition]))	
		{	
			next -= 1;
			current -= 1;
			return destroyWest;
		}	
		return moveWest;	
		}	
		else if (direction == south)	
		{	
			printf("\n below \n");
			if ((currentSurface == land && world[nextPosition] == '~')    (currentSurface == water && world[nextPosition] == 'O'))
			{
			if (currentSurface ==
land)			land)
			{
			currentSurface
			== water;
			}
			else if (currentSurface
			{
			currentSurface
			== water;



```

    }
    else if (currentSurface == water)
    {
        currentSurface
    }
    next -= 1;
    current -= 1;
    return 5;
}
if
(ifObstacle(world[nextPosition]))
{
    next -= 1;
    current -= 1;
    return destroySouth;
}
return moveSouth;
}
else if (direction == north)
{
    printf("\n up \n");
    if ((currentSurface == land &&
world[nextPosition] == '~') || (currentSurface == water &&
world[nextPosition] == 'O'))
    {
        if (currentSurface ==
land)
        {
            currentSurface
        }
        else if (currentSurface
== water)
        {
            currentSurface
        }
        next -= 1;
        current -= 1;
        return 5;
    }
    if
(ifObstacle(world[nextPosition]))
    {
        next -= 1;
        current -= 1;
        return destroyEast;
    }
    return moveEast;
}
else if (direction == west)
{
    printf("\n left \n");
    if ((currentSurface == land &&
world[nextPosition] == '~') || (currentSurface == water &&
world[nextPosition] == 'O'))
    {
        if (currentSurface ==
land)
        {
            currentSurface
        }
        else if (currentSurface
== water)
        {
            currentSurface
        }
        next -= 1;
        current -= 1;
    }
}
}
goTarget = 1;
// reset for future use
currentPosition = 0;
nextPosition = 0;
current = 0;
next = 1;
}

//Return direction to robot to go to initial position
if (goInitial == 0)

```

```

        }
        if
(ifObstacle(world[nextPosition]))
        {
            next -= 1;
            current -= 1;
            return destroyWest;
        }
        return moveWest;
    }
    else if (direction == south)
    {
        printf("\n below \n");
        if ((currentSurface == land &&
world[nextPosition] == '~') || (currentSurface == water &&
world[nextPosition] == 'O'))
        {
            if (currentSurface ==
land)
            {
                currentSurface
= water;
            }
            else if (currentSurface
== water)
            {
                currentSurface
= land;
            }
            next -= 1;
            current -= 1;
            return 5;
        }
        if
(ifObstacle(world[nextPosition]))
        {
            next -= 1;

```

```

        current -= 1;
        return destroySouth;
    }
    return moveSouth;
}
else if (direction == north)
{
    printf("\n up \n");
    if ((currentSurface == land &&
world[nextPosition] == '~') || (currentSurface == water &&
world[nextPosition] == 'O'))
    {
        if (currentSurface ==
land)
        {
            currentSurface
= water;
        }
        else if (currentSurface
== water)
        {
            currentSurface
= land;
        }
        next -= 1;
        current -= 1;
        return 5;
    }
    if
(ifObstacle(world[nextPosition]))
    {
        next -= 1;
        current -= 1;
        return destroyNorth;
    }
    return moveNorth;
}
}
}

```

## Eidesstattliche Erklärung

Hiermit bestätige ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen sowie Hilfsmittel genutzt habe. Alle Ausführungen, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind deutlich kenntlich gemacht. Außerdem versichere ich, dass die vorliegende Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

## Affidavit

I hereby confirm that I have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, I assure that this paper has not been part of a course or examination in the same or a similar version.

Mohamad Zaidi, Zafirul Izzat Bin      Soest, 16.07.2021



Name, Vorname

Ort, Datum

Unterschrift

Last Name, First Name

Location, Date

Signature

Bin Sh Abu Bakar,

Sheikh Muhammad Adib

Soest, 16.07.2021



Name, Vorname

Ort, Datum

Unterschrift

Last Name, First Name

Location, Date

Signature

Bin Md Radzi, Hadi Imran

Soest, 16.07.2021



Name, Vorname

Ort, Datum

Unterschrift

Last Name, First Name

Location, Date

Signature

Ammar Haziq, Bin Mohd Halim

Soest, 16.07.2021



Name, Vorname

Ort, Datum

Unterschrift

Last Name, First Name

Location, Date

Signature

Sulaimon Adijat Ajoke

Soest, 16.07.2021



Name, Vorname

Ort, Datum

Unterschrift

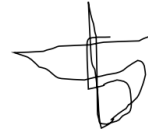
Last Name, First Name

Location, Date

Signature

Habeeb Rilwan, Giwa

Soest, 16.07.2021



---

Name, Vorname

Ort, Datum

Unterschrift

Last Name, First Name

Location, Date

Signature