RescueBot

6th Marwa Mohammed Nabawey Hassan
Hochschule Hamm-Lippstadt
Electronic Engineering Depertment
Prototyping and Systems Engineering Course
marwa-mohammed-nabawey.hassan@stud.hshl.de

***Index Terms*—Coding, simulation, maps, programming**

I. Prototyping Programming

During the programming and simulation phases, the robot starts to develop its own view of the world using few symbols to represent each object, we have been asked to deal with different environments and program the robot to reach its target. In the next section, we are going to explain how we dealt with each required task, so that we end up having a robot that meet all the requirements. We have developed our code functions in C using Visual Studio Code tool.

A. Different Targets

Throughout the representation of the Robot's world by a single map defined with character array, the Robot was defined by character 'R', and asked to reach either a land target defined by 'T' or sea target 't', and return to its home base which defined by 'X' after the robot leaves the initial position taking into consideration that, after the robot pick its target the position will turn into 'O' and the robot has a limitation of maximum 200 steps.

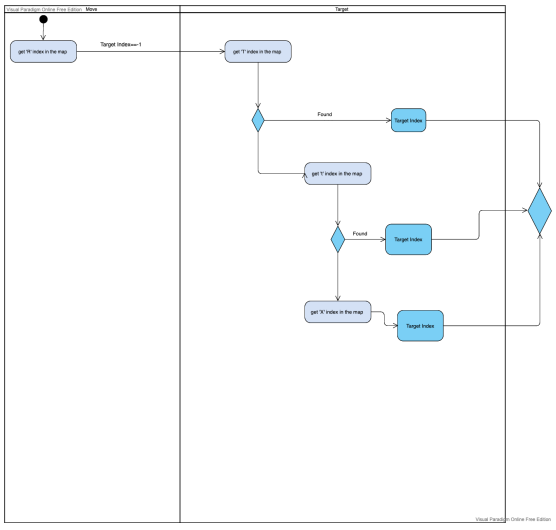

Fig. 1. Different Target's Activity Diagram

The Fig.1 shows the activity diagram of our approach we used to deal with different Targets.

Our approach to solve this challenge, was to initialize a Target index by "-1" after getting the Robot index in the map, the function"get_pos" start to search for first the target 'T' if found it will then consider it as the current Target index, if not then as next step it searches for target 't' and considered it as current target index, as a third step the function will search for the target 'X' which is the home base and considered it as the current target. Fig. 2shows the code implementation of this task.

```
// Dealing with different Targets
int robot_index, width, target_index = -1;
int bestRoute = 0;
width = get_pos(world, '\n') + 1;
robot_index = get_pos(world, 'R');
target_index = get_pos(world, 'T');
if(target_index == -1)
    target_index = get_pos(world, 't');
if(target_index == -1)
    target_index = get_pos(world, 'X');
```

Fig. 2. Targets code.

B. Avoiding Walls

As a next task, the robot were asked to avoid crashing into walls, which is represented by '#' , to develop that, a function ' move ' should return an integer that represents the direction, where the robot should navigate as follows : for North return 1, East return 2, South return 3 and West return 4, the Fig .3 shows the activity diagram of our approach to avoid the wall.

Our approach to solve this challenge, was to get Min. steps in each direction by passing the next step to function called "minEnergy" and returning the value and store it in four variable "ifUp indicate North direction ", "ifDown for South direction ", "ifRight for East " and "IfLeft for West". The Fig.4shows the activity diagram of the "minEnergy " Function.

The minEnergy function check if the next potential step is wall, it will return then a '1000' which can be considered as great value, so the robot won't ever consider this direction, or the other option is that it returns the actual number of steps when taking this direction, the Fig 5 shows the code implementation of minEnergy function.

After the calculation of the potential value in each direction, the values are get compared and store in variable called "bestRoute" which hold one of the direction NORTH, SOUTH,

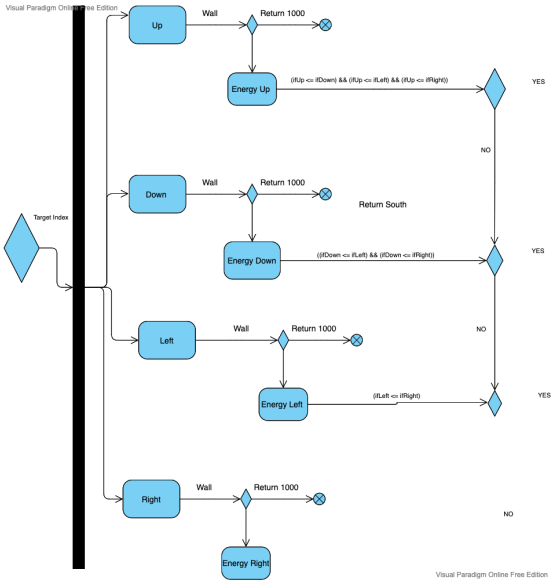

Fig. 3. Avoiding walls Activity Diagram

Fig. 4. min.Energy Activity Diagram .

```
int minEnergy(char *world, int initial_position, int width, int target_index)
{
    int totalEnergy = 1;     //counter for the steps to go to the target
    int steps[200];
    for (int i = 0; i < 200; i++)   //Clear the steps counter
        steps[i] = 0;
    steps[initial_position] = ((world[initial_position] == '~' && current_mode == 0) ||
        ((world[initial_position] == 'O' && current_mode == 1) || world[initial_position] == '*')  ? 4 : 1);
    int targetFound = 0;
    while(targetFound == 0)
    {
        if(steps[target_index] != 0){
            return steps[target_index];
        }
        for(int i = 0; i < 200; i++)
        {
            if(steps[i] == totalEnergy)
            {
                if((world[i - width] != '#') && (steps[i - width] == 0))
                    steps[i - width] = newCellValue(world, steps, i, i - width, totalEnergy);
                if((world[i + width] != '#') && (steps[i + width] == 0))
                    steps[i + width] = newCellValue(world, steps, i, i + width, totalEnergy);
                if((world[i + 1] != '#') && (steps[i + 1] == 0))
                    steps[i + 1] = newCellValue(world, steps, i, i + 1, totalEnergy);
                if((world[i - 1] != '#') && (steps[i - 1] == 0))
                    steps[i - 1] = newCellValue(world, steps, i, i - 1, totalEnergy);
            }
        }
        totalEnergy ++;
    }
    return totalEnergy;
}
```

Fig. 5. The implementation of minEnergy function .

EAST and WEST and considered it as the best route to reach the target, Fig 6 shows the implementation of this step.

C. Water Mode

One of the Robot requirement is the ability to drive on water, which represented by (∼) symbol . To drive through

```
//get min steps in every direction
int ifUp = ((world[robot_index - width] == '#') ? 1000 : minEnergy(world, robot_index - width, width, target_index));
int ifDown = ((world[robot_index + width] == '#') ? 1000 : minEnergy(world, robot_index + width, width, target_index));
int ifLeft = ((world[robot_index - 1] == '#') ? 1000 : minEnergy(world, robot_index - 1, width, target_index));
int ifRight = ((world[robot_index + 1] == '#') ? 1000 : minEnergy(world, robot_index + 1, width, target_index));
if((ifUp <= ifDown) && (ifUp <= ifLeft) && (ifUp <= ifRight))
    bestRoute = North;
else if((ifDown <= ifLeft) && (ifDown <= ifRight))
    bestRoute = South;
else if(ifLeft <= ifRight)
    bestRoute = West;
else
    bestRoute = East;
```

Fig. 6. The implementation of bestRoute .

water, the Robot should have to toggle from Land to Water Mode and vice versa. In addition to returning 1,2,3,4 to the directions the robot should return 5 in the move function to toggle the driving mode, the Fig.7 shows our Approach to switch between the Two modes.

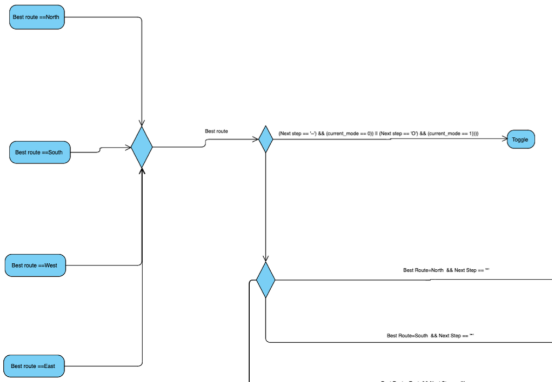

Fig. 7. Activity diagram of Toggle scenario .

After the best route is defined, the move function check if the next step is water and the current mode is Land, and return a Toggle. As well as checking the other scenario if the next step is land and the current mode is water, it will toggle the mode. The Fig.8 shows the code implementation of this task.

```
//decide if robot has to toggle
if(((bestRoute == North)
    &&((world[robot_index - width] == '~') && (current_mode == 0)) || ((world[robot_index - width] == 'O') && (current_mode == 1)))) ||
    ((bestRoute == South)
    && ((world[robot_index + width] == '~') && (current_mode == 0)) || ((world[robot_index + width] == 'O') && (current_mode == 1)))) ||
    ((bestRoute == West)
    &&((world[robot_index - 1] == '~') && (current_mode == 0)) || ((world[robot_index - 1] == 'O') && (current_mode == 1)))) ||
    ((bestRoute == East)
    && ((world[robot_index + 1] == '~') && (current_mode == 0)) || ((world[robot_index + 1] == 'O') && (current_mode == 1))))
    {
        current_mode = (current_mode == 0 ? 1 : 0);
        return Toggle;
    }
```

Fig. 8. The implementation of Toggle scenario .

D. Dealing with Obstacles

As a last Task the Robot's world gets more complicated to consider new barriers, beside the Wall the map will contain obstacles described with ('*') symbol, that can be destroyed, The robot will consume energy based on 10 unit energy per movement, 30 unit energy per toggle and 70 unit energy for trying to destroy an obstacle, and we will have to use min Energy to fulfill the task.

In addition, if we choose to destroy the obstacle, the function will return a new value as 6 to destroy an obstacle to the north, 7 to destroy an obstacle to the east,8 to destroy an obstacle to the south and 9 to destroy an obstacle to the west. The Fig. 9 shows our approach to solve this task .

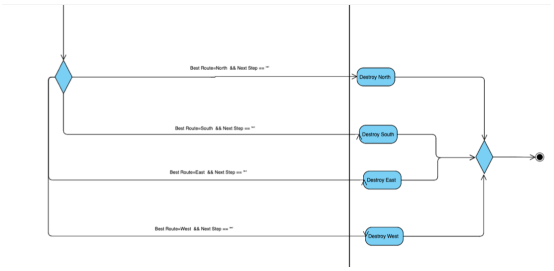

Fig. 9. Activity diagram of dealing with obstecles .

After getting the best route, the function newCellValue will return either Energy+ 1, when they meet any Target, or Energy+4, if it faces any Obstacle in any direction or if faces a water or land, when need to toggle. The fact that it will return only 4 when destroying an obstacle, because the robot will consume only one time the amount of energy to destroy the obstacles in the two-way path. The Fig. 10 shows the activity diagram of newCellValue and how it returns either Energy+1 or Energy+4.

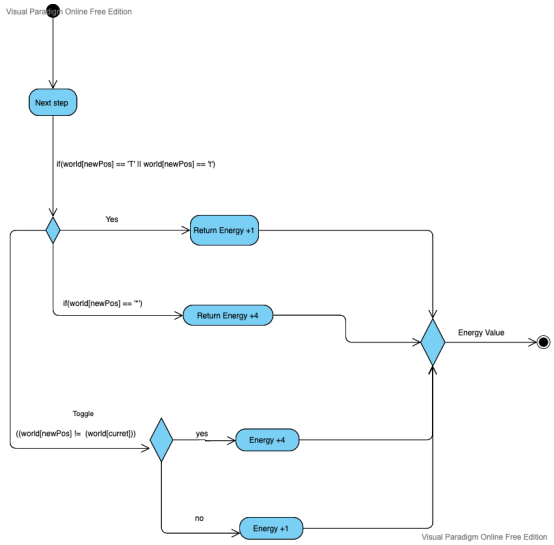

Fig. 10. Activity diagram of newcellValue function .

The implementation of the function is shown in Fig. 11

```
int newCellValue(char *world, int *steps, int curret, int newPos, int actualEnergy)
{
    if(world[newPos] == 'T' || world[newPos] == 't')
        return actualEnergy + 1;
    if(world[newPos] == '*')
        return actualEnergy + 4;
    return ((world[newPos] !=  (world[curret])) ? actualEnergy + 4 : actualEnergy + 1); //toggle
}
```

Fig. 11. Implementation of newcellValue function .

After getting the best route and the potential next step contains any obstacle, the function will then return to destroy in this direction to deal with the obstacle. The checking of conditions is done in four direction North, South, East, west, If the robot's best route doesn't contain any obstacles, it will send then the best Route. The Fig. 12 shows the code implementation when destroying obstacle.

```
    //dealing with obstecles
if(bestRoute == North && world[robot_index - width] == '*')
    return DestroyNort;
if(bestRoute == South && world[robot_index + width] == '*')
    return DestroySouth;
if(bestRoute == West && world[robot_index - 1] == '*')
    return DestroyWest;
if(bestRoute == East && world[robot_index + 1] == '*')
    return DestroyEast;
return bestRoute; // REPLACE THE RETURN VALUE WITH YOUR CALCULATED RETURN VALUE
```

Fig. 12. Implementation of destroying obstacles scenario .