Student:
Precious I. Philip-Ifabiyi [philipifabiyiprecious@gmail.com]

Lab1: Potential Functions

1 Introduction

The aim of this report is to investigate the problem of path planning with the use of potential functions. A potential function is used to move a robot from a starting point (higher potential) to the goal point, which is at a lower potential. Both attractive and repulsive potentials are needed to develop a potential function. Attractive potential helps the robot locate the goal position, and repulsive potential helps the robot avoid obstacles along its path to the goal. In this report, a wavefront planner, brushfire algorithm, repulsive function and potential functions were implemented to solve the path planning problem in a grid map.
The report is organized in the following way: The algorithms developed are explained in the Methodology section (Section 2). The subsequent section on results (Section 3) demonstrates all the results obtained while implementing the attraction, repulsive, and potential functions. Finally, the results are discussed in the Discussion and Conclusions section (Section 4).

2 Methodology

This section of the report will discuss the programming approach, tools, and methods employed to implement the attraction function, brushfire algorithm, repulsive function, and potential function on a grid map. On the grid map, obstacles are represented as 1, and free spaces are represented as 0.

2.1 Attraction Function

An attraction function is used to guide the movement of a robot from a starting point to a goal position. This is created using the wavefront planner. The implementation discussed in this report uses Euclidean distance to compute the distance between cells. The following subsections discuss each function developed for the wavefront planner in detail.

2.1.1 Function: *wavefront_planner_euclidean*

The wavefront planner function receives a grid map and a goal position as input and then returns the attraction function for that grid map. A list is used to store cells that the wavefront planner has recently explored.
At the beginning of the function, the potential at the goal position is set to 2, and the goal position is placed into a list called *queue*. This *queue* will be used to explore the neighbouring cells in a wavefront manner.
The algorithm enters a loop that continues until *queue* is empty. Inside the loop, a new list called

new_ queue is created for the next wavefront expansion. The planner iterates over each position in *queue*. For each neighbour of the current position (obtained from the *possible_ motions* function), it calculates a value based on the sum of the current cell value and the Euclidean distance between the neighbour and the current position. If the neighbouring cell in the map is a free space and is unexplored, its potential is updated to the calculated value. If the neighbouring cell has been visited before, the value is updated to the minimum of the calculated value and the current value in the map. This means that the potential for positions that contain obstacles remains 1. The neighbour is then added to *new_ queue* for the next iteration.
After expanding the current wavefront, *queue* is updated with the new one i.e. *new_ queue*. Once the wavefront expansion is complete, the function returns the modified map, which now contains the attraction potential based on the wavefront expansion and Euclidean distance heuristic.
The Python implementation of the wavefront planner is shown in Listing 1.

2.1.2 Function: *possible_ motions*

This function receives a 2D map and a position as input and returns a list of possible neighbouring positions that can be reached from the given position in the map. It considers positions to be valid if they are unoccupied (i.e., have a value of 0 in the map) and within the bounds of the map. Neighbours of a cell are considered using 8-point connectivity.
The Python implementation of the possible_motions function is shown in Listing 2.

Listing 1: Wavefront planner function

Input : Map, Goal

Output: Attraction function

1

```python
# Wavefront Planner using Euclidean distance with Attraction Potential
def wavefront_planner_euclidean(map, goal):
    # Create an attraction potential map, initially identical to the given map
    attract_pot = map
    # Mark the goal cell in the attraction potential map with a value of 2
    attract_pot[goal[0], goal[1]] = 2
    # Initialize the queue with the goal position
    queue = [goal]

    # Continue the wavefront expansion until the queue is empty
    while len(queue) != 0:
        # Create a new queue for the next wavefront expansion
        new_queue = []

        # Iterate over each position in the current queue
        for position in queue:
            # Explore neighbors of the current position
            for neighbour in possible_motions(map, position):
                # Calculate the value for the current neighbor based on the Euclidean distance
                value = attract_pot[position[0], position[1]] + eucl_dis(neighbour, position)

                # Update the attraction potential map with the calculated value
                if attract_pot[neighbour[0], neighbour[1]] == 0:
                    # If the neighbor is unexplored, set its value to the calculated value
                    attract_pot[neighbour[0], neighbour[1]] = value
                else:
                    # If the neighbor has been visited before, update with the minimum value
                    attract_pot[neighbour[0], neighbour[1]] = min(value, attract_pot[neighbour
    [0], neighbour[1]])

                # Add the neighbor to the new queue for the next iteration
                new_queue.append(neighbour)

        # Update the current queue with the new queue
        queue = new_queue

    # Return the attraction potential map with updated values after wavefront expansion
    return attract_pot
```

Listing 2: possible_motions method

Input : 2D Map, Position in a map
Output: List of neighbouring cells

1

```python
# Function to find possible valid motions from a given position on the map
def possible_motions(map, position):
    # Get the dimensions of the map
    x = map.shape[1]
    y = map.shape[0]

    # Initialize an empty list to store valid neighboring positions
    possible = []

    # Check and append valid neighboring positions based on map boundaries and obstacle-free
    cells
    if position[1] - 1 >= 0 and map[position[0], position[1] - 1] == 0:
        possible.append((position[0], position[1] - 1))
    if position[0] - 1 >= 0 and map[position[0] - 1, position[1]] == 0:
        possible.append((position[0] - 1, position[1]))
    if position[1] + 1 < x and map[position[0], position[1] + 1] == 0:
        possible.append((position[0], position[1] + 1))
    if position[0] + 1 < y and map[position[0] + 1, position[1]] == 0:
        possible.append((position[0] + 1, position[1]))
    if position[1] - 1 >= 0 and position[0] - 1 >= 0 and map[position[0] - 1, position[1] - 1]
    == 0:
        possible.append((position[0] - 1, position[1] - 1))
    if position[1] + 1 < x and position[0] - 1 >= 0 and map[position[0] - 1, position[1] + 1]
    == 0:
        possible.append((position[0] - 1, position[1] + 1))
    if position[1] + 1 < x and position[0] + 1 < y and map[position[0] + 1, position[1] + 1] ==
     0:
        possible.append((position[0] + 1, position[1] + 1))
    if position[1] - 1 >= 0 and position[0] + 1 < y and map[position[0] + 1, position[1] - 1]
    == 0:
        possible.append((position[0] + 1, position[1] - 1))

    # Return the list of valid neighboring positions
    return possible
```

2.1.3 Function: *eucl_dis*

This function is used to calculate the Euclidean distance between 2 points. It is used by the *wavefront_planner_euclidean* function to obtain the attraction potential for a cell in a grid map. The Python implementation is shown in 3.

Listing 3: Eucl_dis method

Input : Cell1, Cell2
Output: Euclidean distance between Cell1 and Cell2

1

```python
def eucl_dis(point1,point2):
    return math.sqrt((point1[0]-point2[0])**2 + (point1[1]-point2[1])**2)
```

2.2 Path Finding

The path to the goal from any starting point can be found using the attraction or the potential function. This section describes the functions developed to find the path to the goal from any starting point.

2.2.1 Function: *find_the_path*

The function, *find_the_path*, takes an attraction or potential function and a starting position as input and returns a path to the goal from the starting position. The following ideas are used to get the path to goal; Apart from occupied positions, goal positions always have the lowest potential in an attraction function. This same idea works for potential functions.
So, the path to the goal is obtained by iteratively selecting the next position based on the minimum value in the attraction or potential function. Iteration is carried out by a *while* function that stops when it has reached a minimum. The modus operandi of this function is similar to that of the gradient descent algorithm. The cell with the minimum value is calculated and returned by the *find_minimum* function.
The implementation of this function in Python is shown in Listing 4.

Listing 4: find_the_path method

Input : Potential function, starting position
Output: Path from starting position to goal position

1

```python
def find_the_path(potential_function, start):
    path=[start]
    posit=start
    isGoal=False
    while isGoal == False:
        minimum=find_minimum(potential_function,posit)
        if minimum==posit:
            isGoal=True
        else:
            path.append(minimum)
            posit=minimum
    return path
```

2.2.2 Function: *find_minimum*

This function receives a potential function and a position and returns the neighbouring free cell of that position with the minimum potential. The neighbours considered are the left, up, right, down, and diagonal neighbours of a cell. Edges and obstacles were also taken into consideration. The Python implementation is shown in Listing 5.

Listing 5: find_minimum method

Input : Potential function, Position
Output: Neighbouring free cell with the minimum potential

1

```python
def find_minimum(attract_function, pos):
    minimum=pos
    if pos[1]-1 >= 0 and attract_function[pos[0],pos[1]-1]!=1 and attract_function[pos[0],pos
    [1]-1]<attract_function[pos[0],pos[1]]:
        minimum=(pos[0],pos[1]-1)
    if pos[0]-1 >=0 and attract_function[pos[0]-1, pos[1]]!=1 and attract_function[pos[0]-1,
    pos[1]]<attract_function[minimum[0],minimum[1]]:
        minimum=(pos[0]-1,pos[1])
    if pos[1]+1<attract_function.shape[1] and attract_function[pos[0],pos[1]+1]!=1 and
    attract_function[pos[0],pos[1]+1]<attract_function[minimum[0],minimum[1]]:
        minimum=(pos[0],pos[1]+1)
    if pos[0]+1<attract_function.shape[0] and attract_function[pos[0]+1,pos[1]]!=1 and
    attract_function[pos[0]+1,pos[1]]<attract_function[minimum[0],minimum[1]]:
        minimum=(pos[0]+1,pos[1])
    if pos[1]-1>=0 and pos[0]-1>=0 and attract_function[pos[0]-1,pos[1]-1]!=1 and
    attract_function[pos[0]-1,pos[1]-1]<attract_function[minimum[0],minimum[1]]:
        minimum=(pos[0]-1, pos[1]-1)
    if pos[0]-1>=0 and pos[1]+1<attract_function.shape[1] and attract_function[pos[0]-1,pos
    [1]+1]!=1 and attract_function[pos[0]-1,pos[1]+1]<attract_function[minimum[0],minimum[1]]:
        minimum=(pos[0]-1, pos[1]+1)
    if pos[1]+1<attract_function.shape[1] and pos[0]+1<attract_function.shape[0] and
    attract_function[pos[0]+1,pos[1]+1]!=1 and attract_function[pos[0]+1,pos[1]+1]<
    attract_function[minimum[0],minimum[1]]:
        minimum=(pos[0]+1,pos[1]+1)
    if pos[1]-1>=0 and pos[0]+1<attract_function.shape[0] and attract_function[pos[0]+1,pos
    [1]-1]!=1 and attract_function[pos[0]+1,pos[1]-1]<attract_function[minimum[0],minimum[1]]:
        minimum=(pos[0]+1,pos[1]-1)
    return minimum
```

2.3 Brushfire Algorithm

The brushfire algorithm is used for computing the distance of cells to obstacles in a grid map ($D(q)$). This is very useful for obtaining the repulsive function of a grid map. The brushfire algorithm is similar to the wavefront planner except that the obstacle positions are placed inside the queue instead of the goal position.
The *brushfire* function will be discussed in the following subsection. All other functions used by the brushfire function have already been discussed in Section 2.1.

2.3.1 Function: *brushfire*

This function implements the brushfire algorithm. As mentioned before, it is similar to the wavefront planner discussed in Section 2.1.1. The differences will be noted in this section. At the start

of the algorithm, all the positions of the cells with obstacles are obtained. This is achieved by using the *numpy.where* function. The obstacles are placed in a list called *queue*, which will be used to propagate the values.
The implementation of this function on Python is shown in Listing 6.

Listing 6: brushfire function

Input : Binary map
Output: Brushfire algorithm output

1

```python
# Function to perform Brushfire algorithm on a binary map
def brushfire(map):
    # Find the coordinates of obstacle cells (value 1) in the map
    x, y = np.where(map == 1)

    # Initialize a queue with obstacle coordinates
    queue = []
    for f, b in zip(x, y):
        queue.append((f, b))

    # Continue the Brushfire algorithm until the queue is empty
    while len(queue) != 0:
        # Create a new queue for the next iteration
        new_queue = []

        # Iterate over each position in the current queue
        for pos in queue:
            # Explore neighbors of the current position
            for neigh in possible_motions(map, pos):
                # Calculate the value for the current neighbor based on the Euclidean distance
                val = map[pos[0], pos[1]] + eucl_dis(neigh, pos)

                # Update the map with the calculated value if the neighbor is unexplored (map
    value is 0)
                if map[neigh[0], neigh[1]] == 0:
                    map[neigh[0], neigh[1]] = val
                else:
                    # If the neighbor has been visited before, update with the minimum value
                    map[neigh[0], neigh[1]] = min(map[neigh[0], neigh[1]], val)

                # Add the neighbor to the new queue for the next iteration
                new_queue.append(neigh)

        # Update the current queue with the new queue
        queue = new_queue

    # Return the map with updated values after Brushfire algorithm
    return map
```

2.4 Repulsive Function

Given an array of distances of each cell in a grid map to obstacles ($D(q)$), the function, *repulsive_function*, calculates the repulsive potential for all the positions in a grid map. The following formula is used to calculate the repulsive function ($rep(q)$).

$$rep(q) = \begin{cases} 1, & \text{if } D(q) = 1 \\ 4 \cdot (\frac{1}{D(q)} - \frac{1}{Q})^2, & \text{if } D(q) \leq Q \\ 0, & \text{if } D(q) > Q \end{cases}$$

Where Q is the desired radius of repulsion with the obstacles.
It can be concluded from the formula that the cells whose distances from obstacles are greater than Q will look flat, while areas whose distances to the obstacles are less than Q will look like a mountain. This gives a repulsive effect and makes the robot avoid areas with high potential.
The implementation of the *repulsive_function* function is shown in Listing 7.

Listing 7: repulsive_function

Input : Array of distance of cells to obstacles, Radius of repulsion
Output: Repulsive function

1

```python
def repulsive_function(dis_obs, Q):
    dis_obs[dis_obs>Q]=0
    dis_obs[dis_obs==1]=1
    dis_obs[(dis_obs<=Q) & (dis_obs>1)]=4*((1/dis_obs[(dis_obs<=Q) & (dis_obs>1)])-(1/Q))**2
    return dis_obs
```

2.5 Potential Function

A potential function is a combination of an attractive function and a repulsive function. The attraction function obtained using the wavefront planner and the repulsive function are utilised to get the potential function of a grid map.
In the attractive potential obtained using the wavefront planner, the obstacles are represented with a 1, and the goal position is represented with a 2. The attraction potential for the unoccupied cells is usually greater than 2. So, to merge both the attraction and repulsive potential, the potential of occupied cells is changed to the $max\ value\ in\ the\ attraction\ function + 1$. The attractive function is then normalized so that potentials are between 0 and 1. The normalization that is used is the min-max normalization.
After normalising the attraction function, the attraction and repulsive functions are added together to obtain the potential function.
The implementation of this function in Python is shown in Listing 8.

Listing 8: potential_function

Input : Attraction function, Repulsive function

Output: Potential function

1

```
def potential_function(attraction, repulsion):
    attraction[attraction==1]= attraction.max()+1
    attraction=(attraction-attraction.min())/(attraction.max()-attraction.min())
    potential=attraction+repulsion
    return potential
```

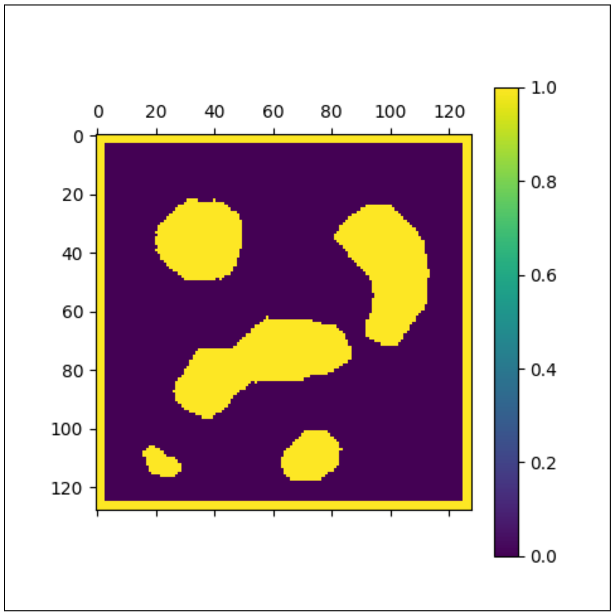
3 Results

This section shows the results of using the wavefront planner and brushfire algorithm to get the attraction, repulsive, and potential functions. It is divided into four subsections where the results of four different maps are displayed individually.

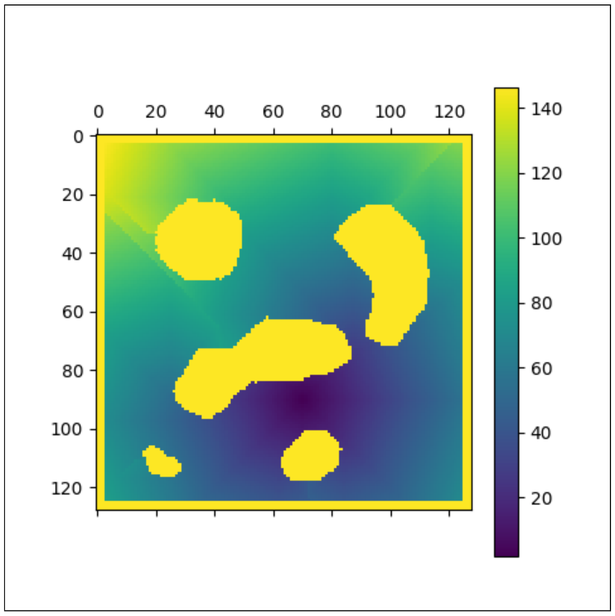
Each subsection contains the attraction function, path from start to goal using only the attraction function, brushfire, repulsive function, potential function (i.e., attraction function + repulsive function normalized), and the path from start to goal using the combined attraction and repulsive functions for a single map.

3.1 First grid map

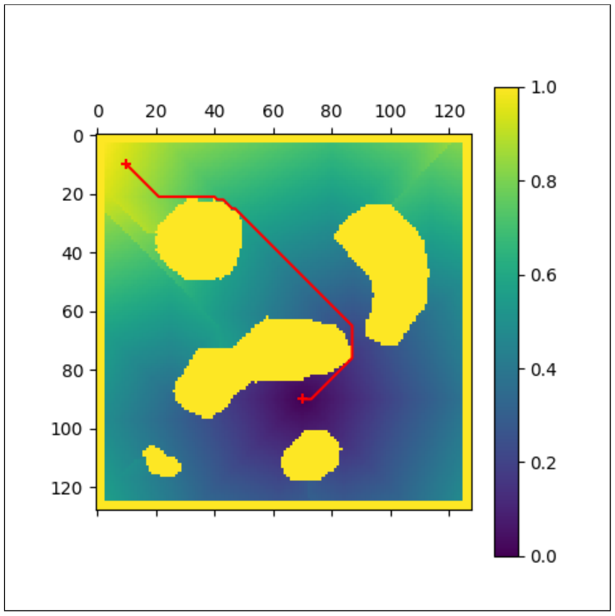
Figure 1 and 2 show the results of obtaining the potential function on the first grid map. The radius of repulsion used is 5.



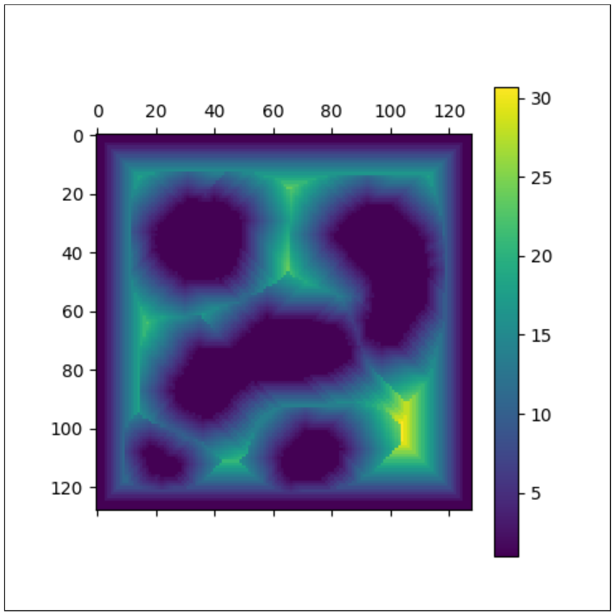
(a) Original Grid map



(b) Attraction function

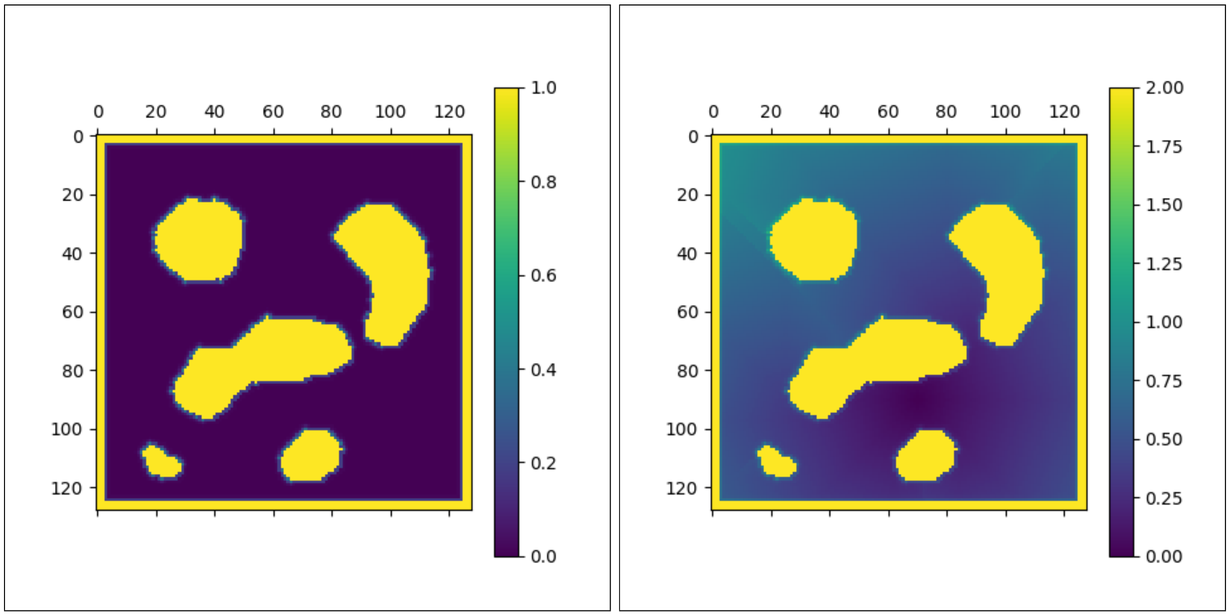


(c) Path on Attraction function

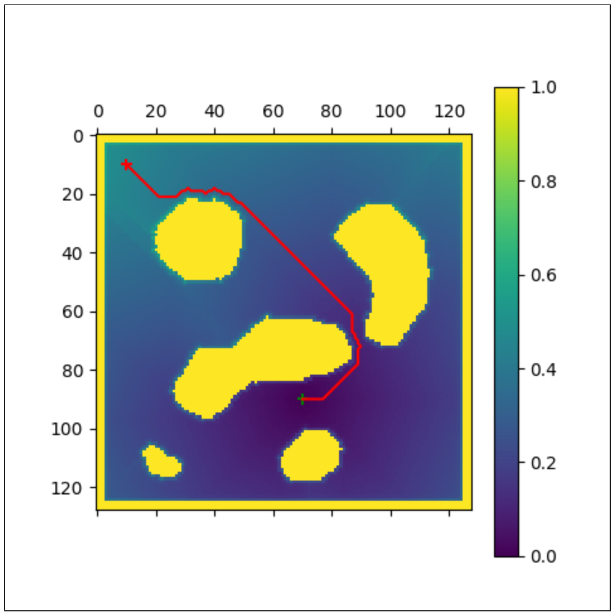


(d) Brushfire

Figure 1: Results for grid map 1



(a) Repulsive function (b) Potential function

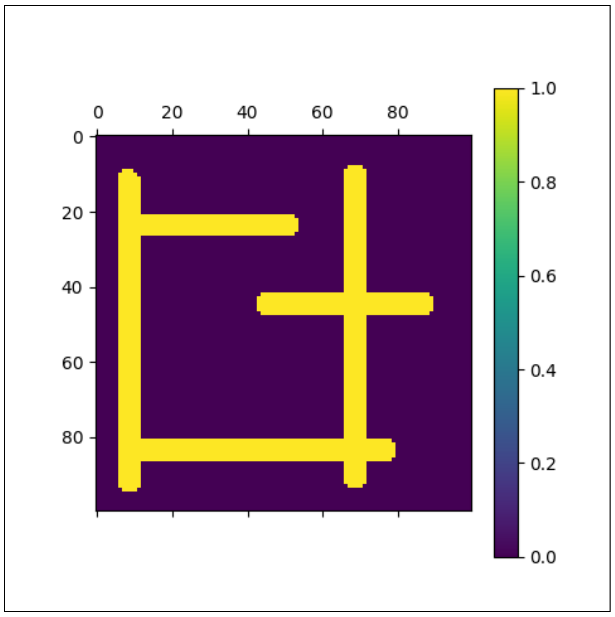


(c) Path on potential function

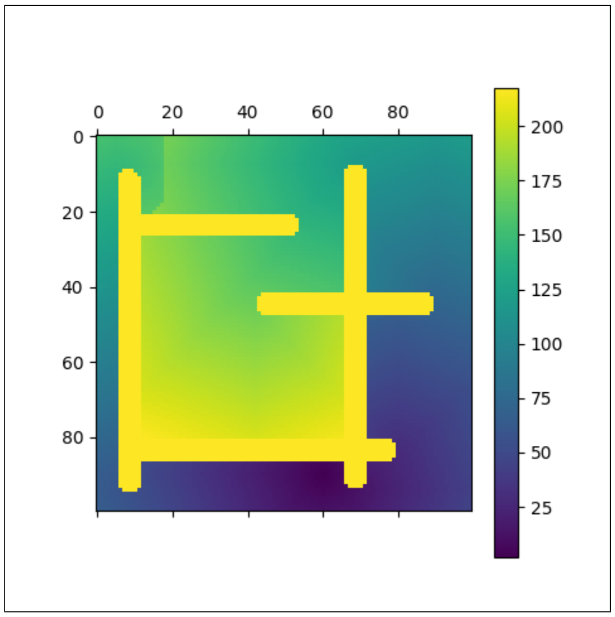
Figure 2: Results for Map 1

3.2 Second grid map

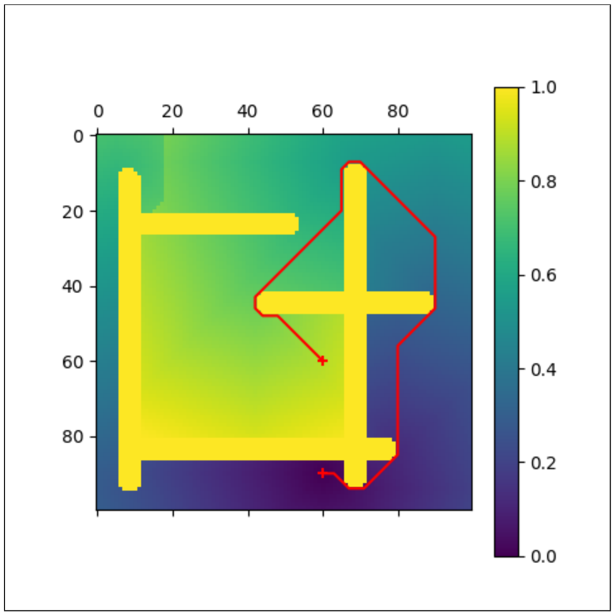
Figure 3 and 4 show the results of obtaining the potential function on the second grid map. The radius of repulsion used is 4.



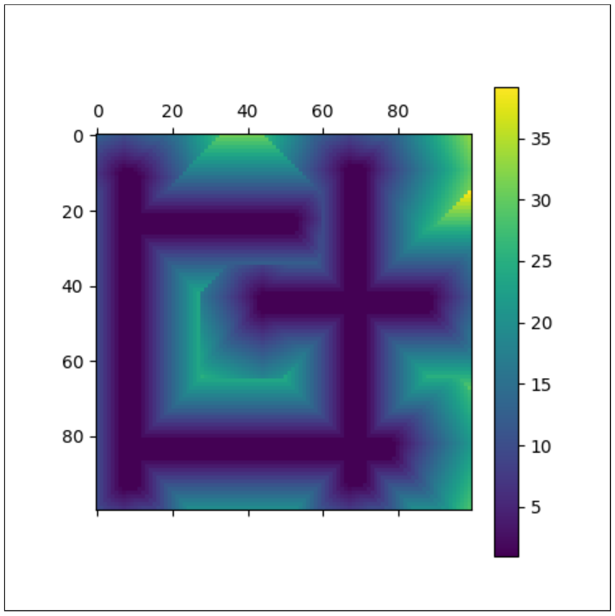
(a) Original Grid map



(b) Attraction function

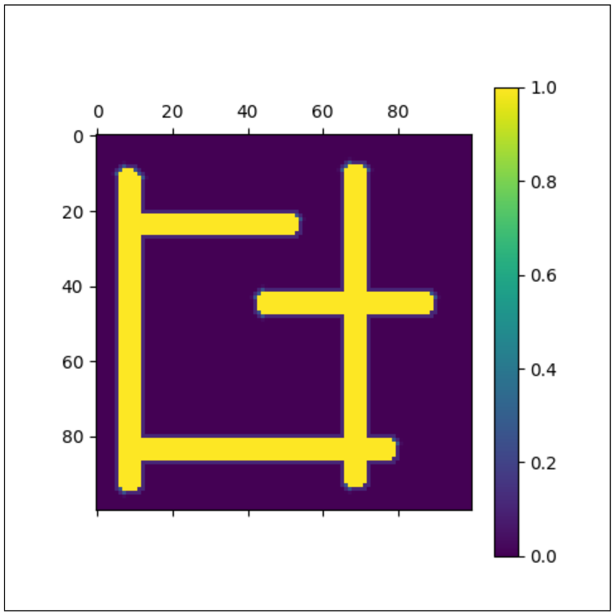


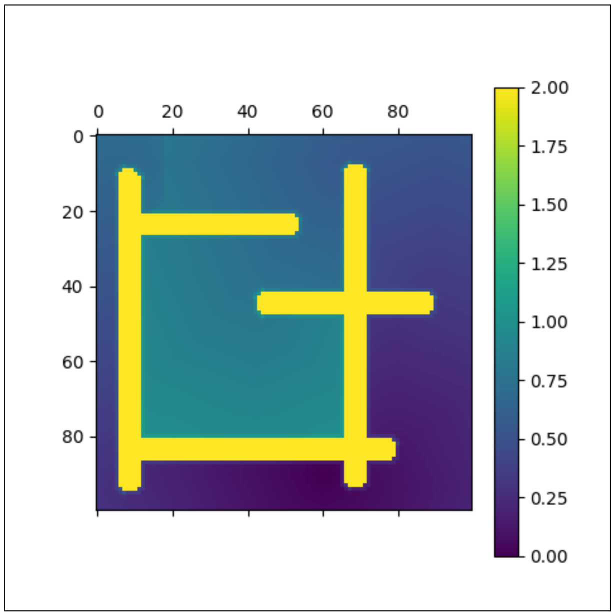
(c) Path on attraction function



(d) Brushfire

Figure 3: Results for map 2


(a) Repulsive function


(b) Potential function

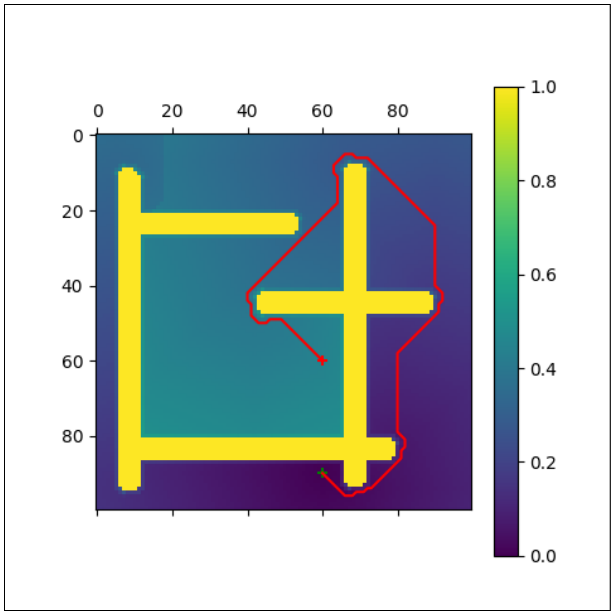
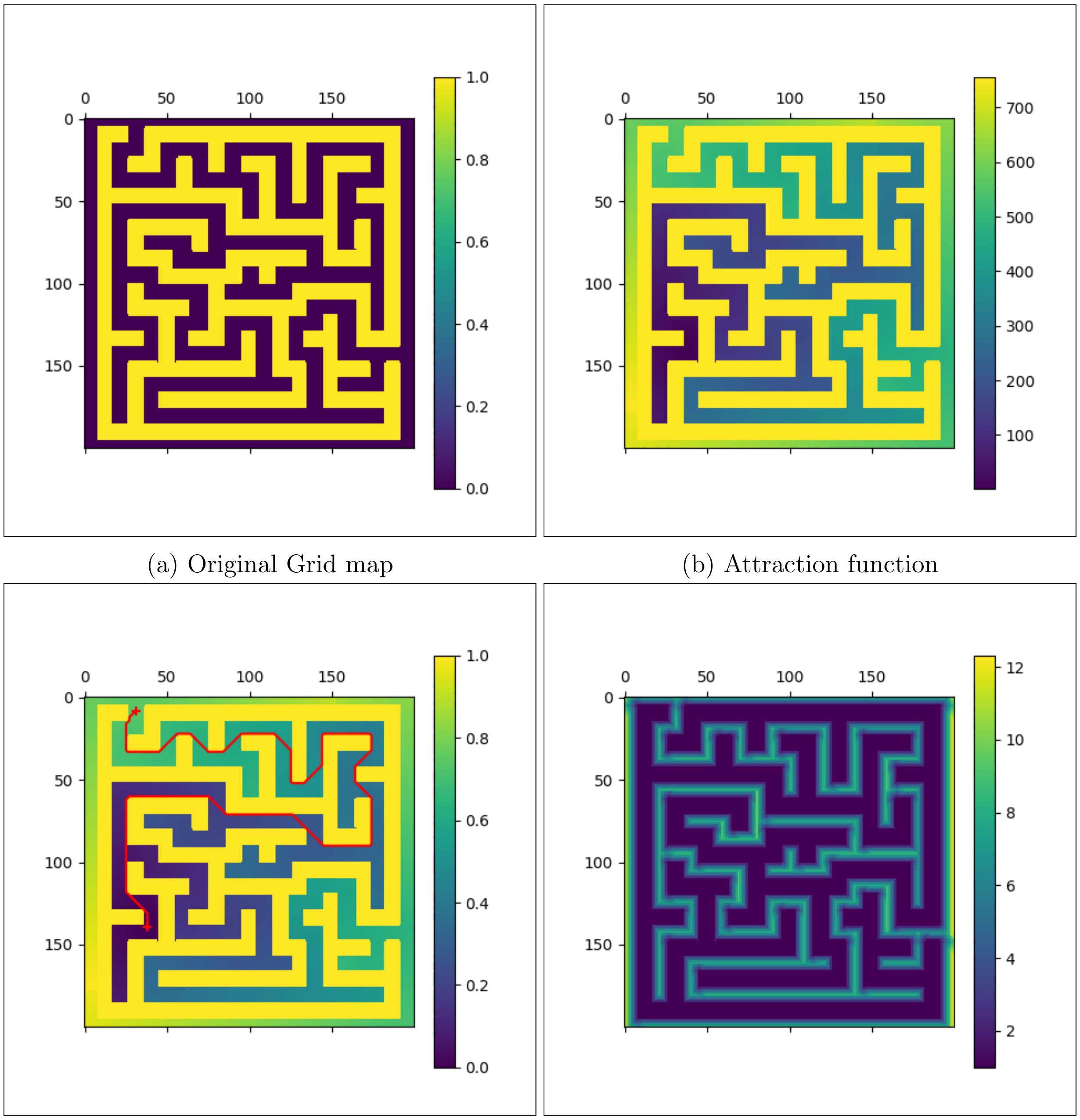

(c) Path on potential function

Figure 4: Results on Map 2

3.3 Third grid map

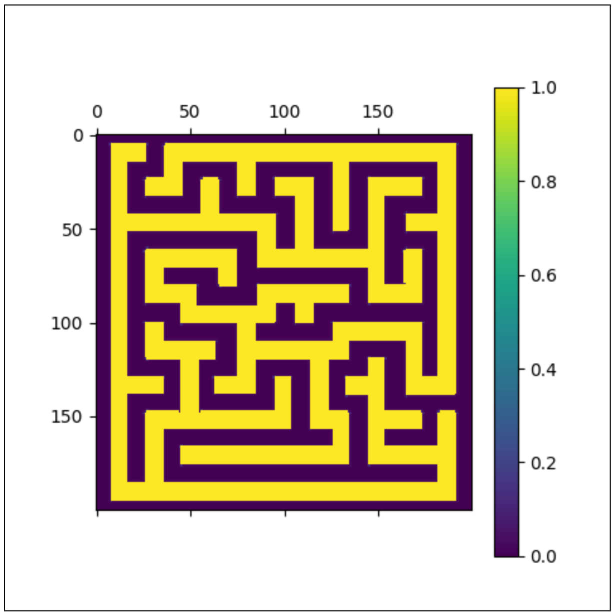
Figure 5 and 6 show the results of obtaining the potential function on the third grid map. The radius of repulsion used is 3.



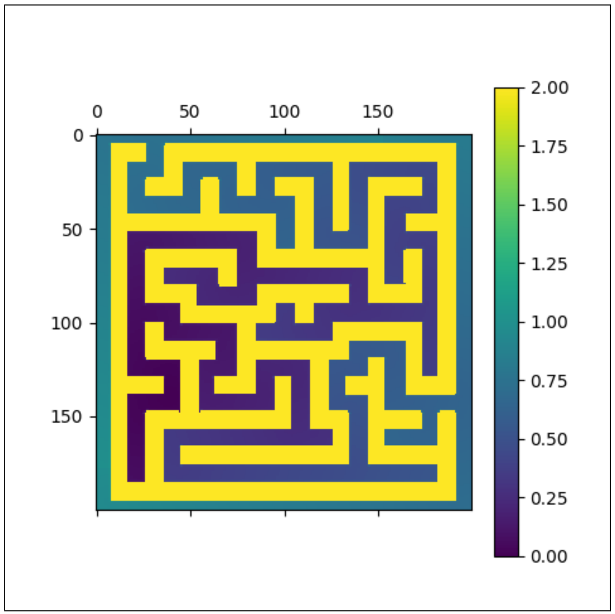
(a) Original Grid map (b) Attraction function

(c) Path on attraction function (d) Brushfire

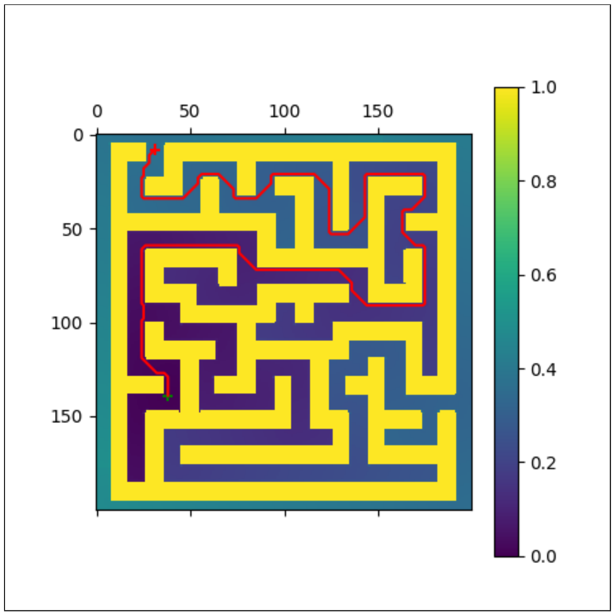
Figure 5: Results for map 3



(a) Repulsive function



(b) Potential function

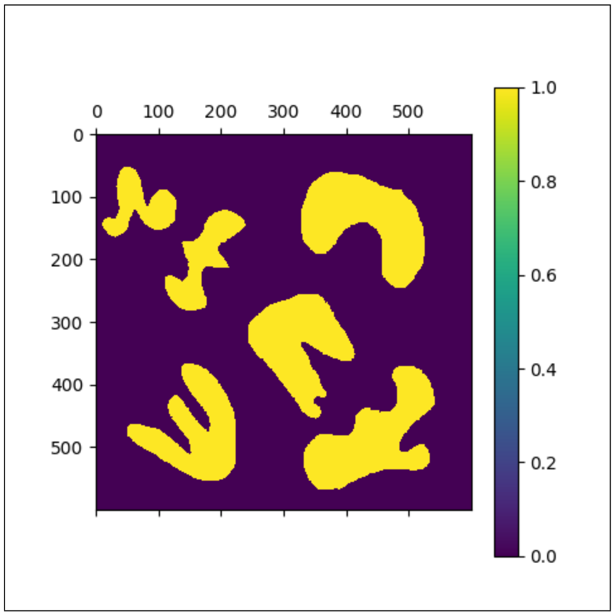


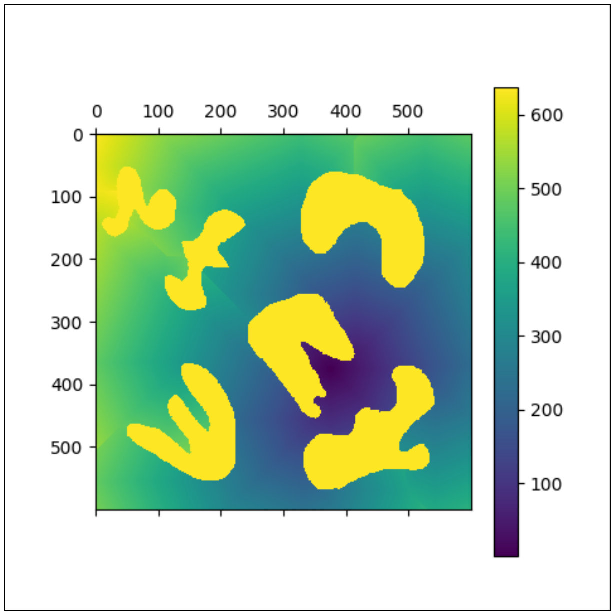
(c) Path on potential function

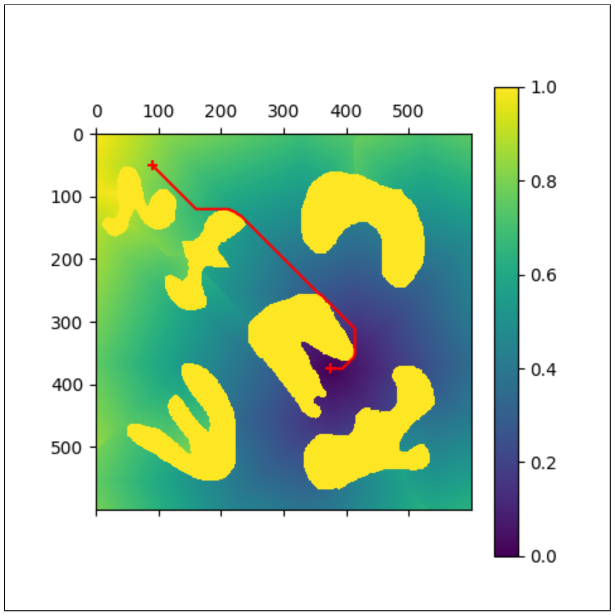
Figure 6: Results on Map 3

3.4 Fourth grid map

Figure 7 and 8 show the results of obtaining the potential function on the fourth grid map. The radius of repulsion used (Q) is 10.


(a) Original Grid map


(b) Attraction function


(c) Path on attraction function

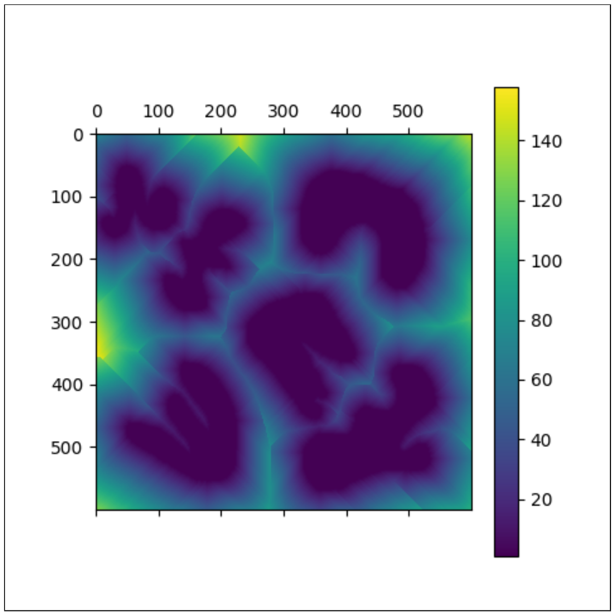
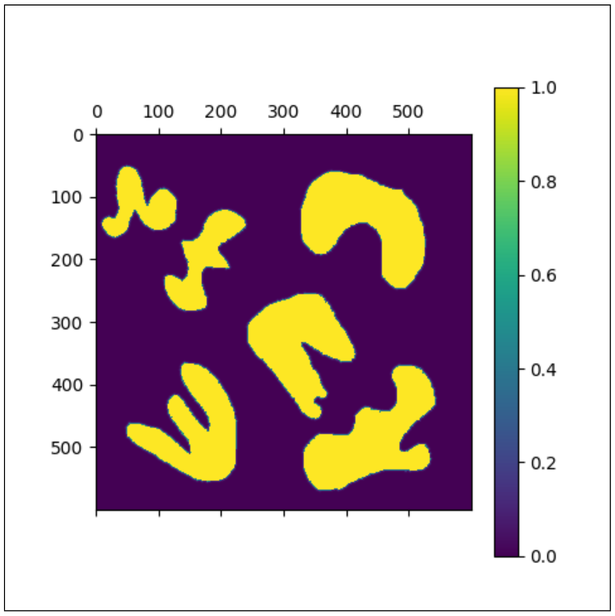
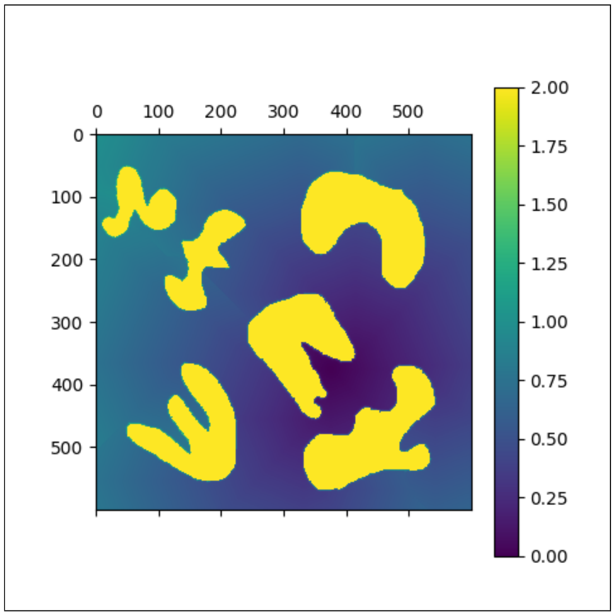

(d) Brushfire

Figure 7: Results for map 4


(a) Repulsive function


(b) Potential function

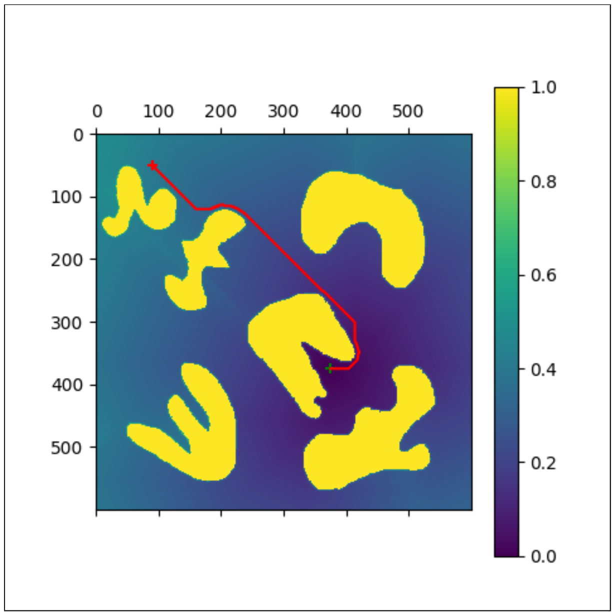

(c) Path on potential function

Figure 8: Results on Map 4

4 Discussion & Conclusions

It can be observed from the results that the paths generated using the attraction function tend to follow the boundary of the obstacles closely. By combining the repulsive function with the attraction function, the distance of the path to the obstacles is regulated by the parameter, Q.

When a large Q value was used, the distance of the generated path from the obstacle increased.

The choice of the repulsion radius, Q, depends on the environment, the size of the obstacles, and the desired behaviour. A small Q should be used when you want the robot to follow obstacles closely. A large Q will suffice if you want a significant buffer zone between the robot and the obstacles. Empirically testing different Q values can also help in decision-making.

The main challenge that was faced was the issue of local minima in the potential function. This only happened when the Q value was large.

The implementation of the attraction and repulsive functions used Euclidean distance. The distance of the 8 neighbours of a cell should be set to 1 if connect-8 is to be used. For connect-4, the up, down, left and right neighbours should only be considered.

In conclusion, this lab report focused on exploring the application of potential functions in the context of robotic path planning. The utilization of potential functions, particularly attraction and repulsive potentials, has proven to be a useful tool in path planning.