\

Autonomous Systems

Lab # 2

Delivered by:

Muhammad Faran Akram
Raúl López Musito

Supervisor:

El Masri El Chaarani, Alaaeddine

Date of Submission:

06/11/2024

Table of Contents

1. Final Outcomes:

All the assigned tasks were completed and below are the results of each part with different maps.

1.1. Part 1(Bush-Fire Algorithm & Attraction Function):

The bush-fire algorithm computes the distance to an obstacle from each cell in the map. The attraction function computes the attraction to the goal for each cell in the map.

1.1.1. **Results on Map 0:**

The bush-fire map_0 shows the distances from obstacles across the grid, with colours ranging from dark (close to obstacles) to bright (far from obstacles). It shows how distance values propagate outward from obstacles. The attraction grid shows the attractive field, computed based on bush-fire distance values, the brighter region has more attraction as it is near to the goal and dark regions have less attraction.

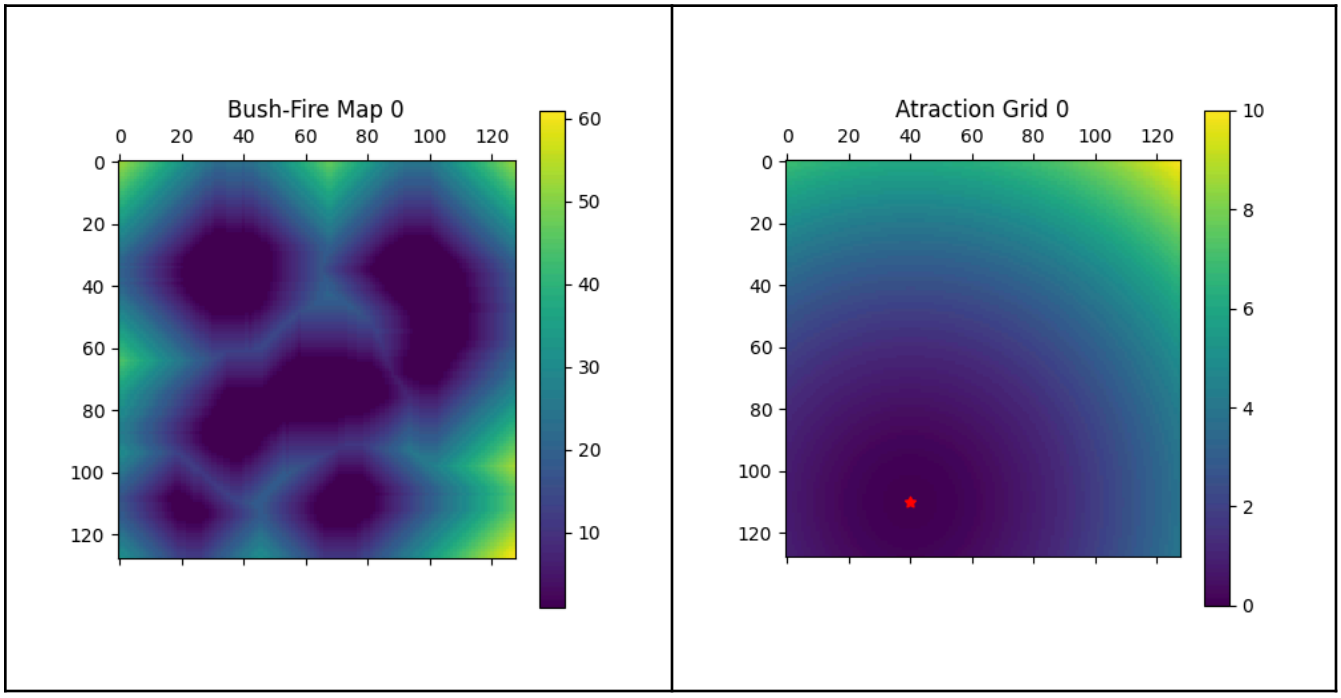


Fig. Bush-Fire & Attraction Grid with neighbours = 4

Using 4-neighbours caused the fire to spread slower and it partially retained the shape of the obstacle resulting in a smooth graph. While using 8-neighbours caused the fire to spread faster and more irregularly, resulting in the graph shown below.

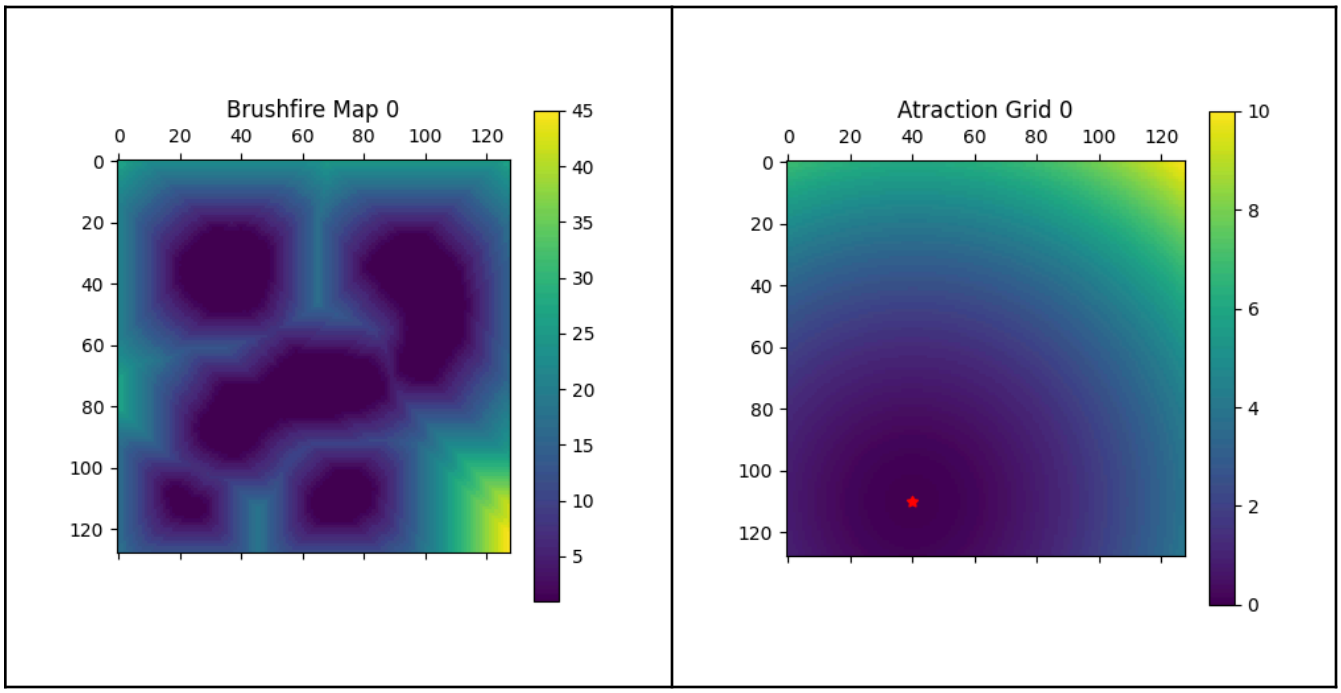


Fig. Bush-Fire & Attraction Grid with neighbours = 8

The repulsive grid displays obstacles' repulsive forces, with dark areas indicating no force and bright areas near obstacles indicating strong repulsion. The total potential grid combines both attractive and repulsive fields.

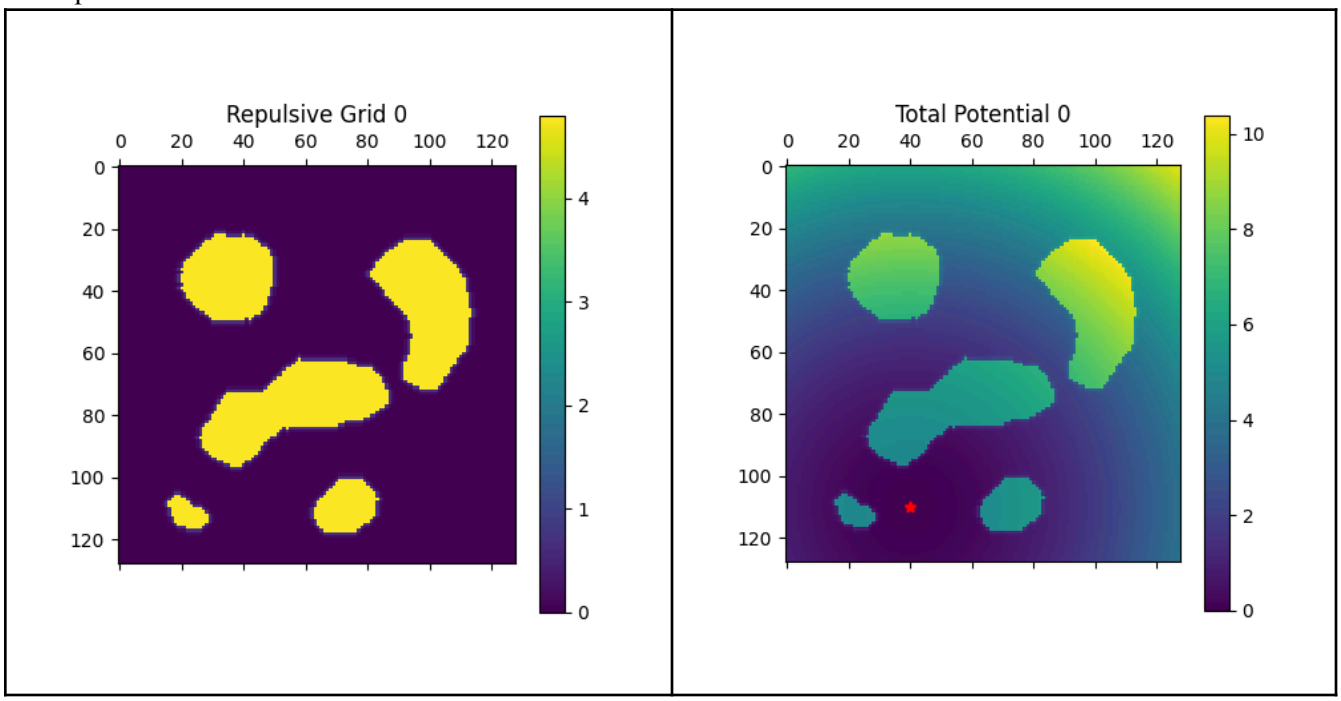


Fig. Repulsive & Total Potential with neighbours = 4

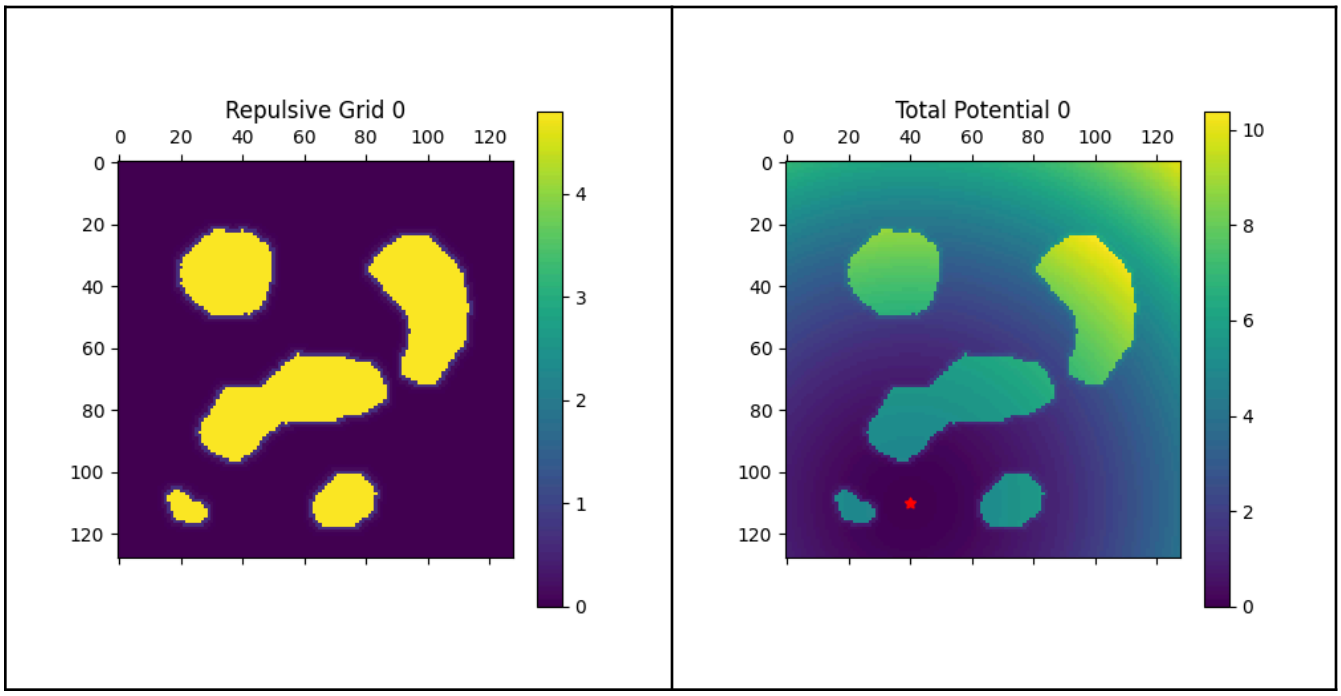


Fig. Repulsive & Total Potential with neighbours = 8

Using 4-neighbours created an irregular repulsive field around the complex-shaped obstacle, while 8-neighbours produced a smoother field, with Q remaining the same (Q=5) in both cases.

1.1.2. **Results on Map 1:**

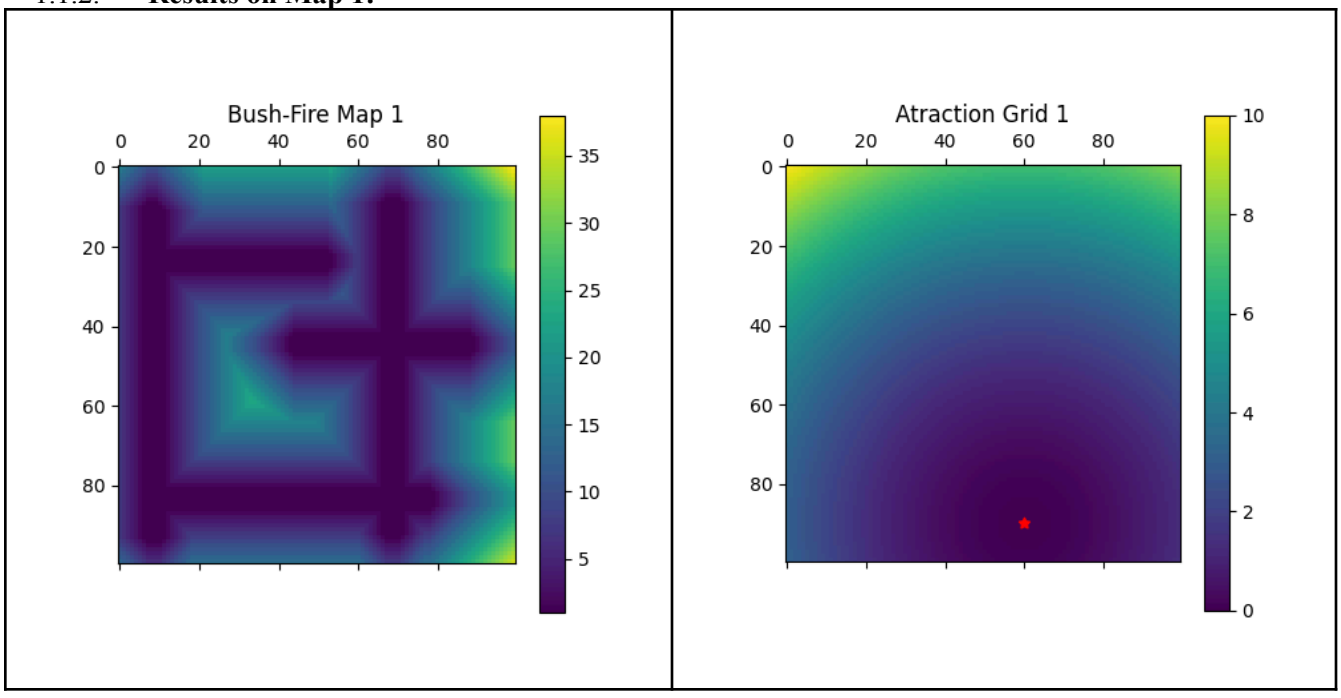


Fig. Bush-Fire & Attraction Grid with neighbours = 4

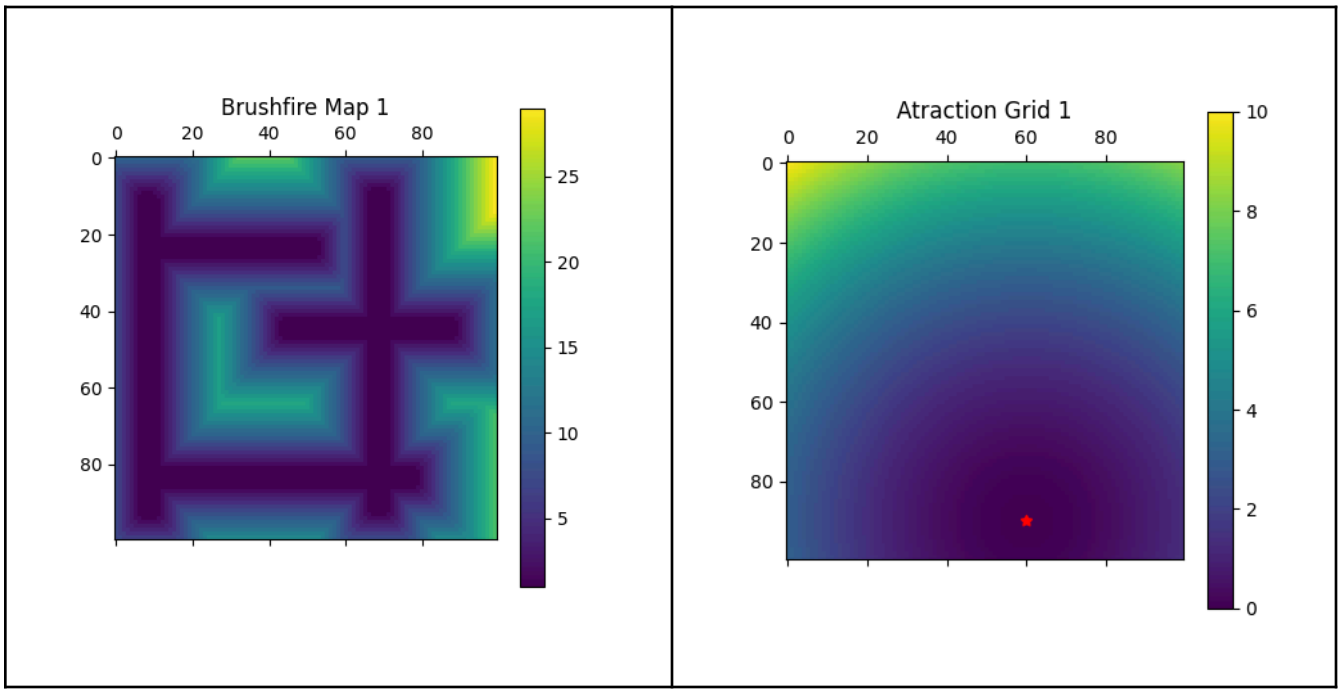


Fig. Bush-Fire & Attraction Grid with neighbours = 8

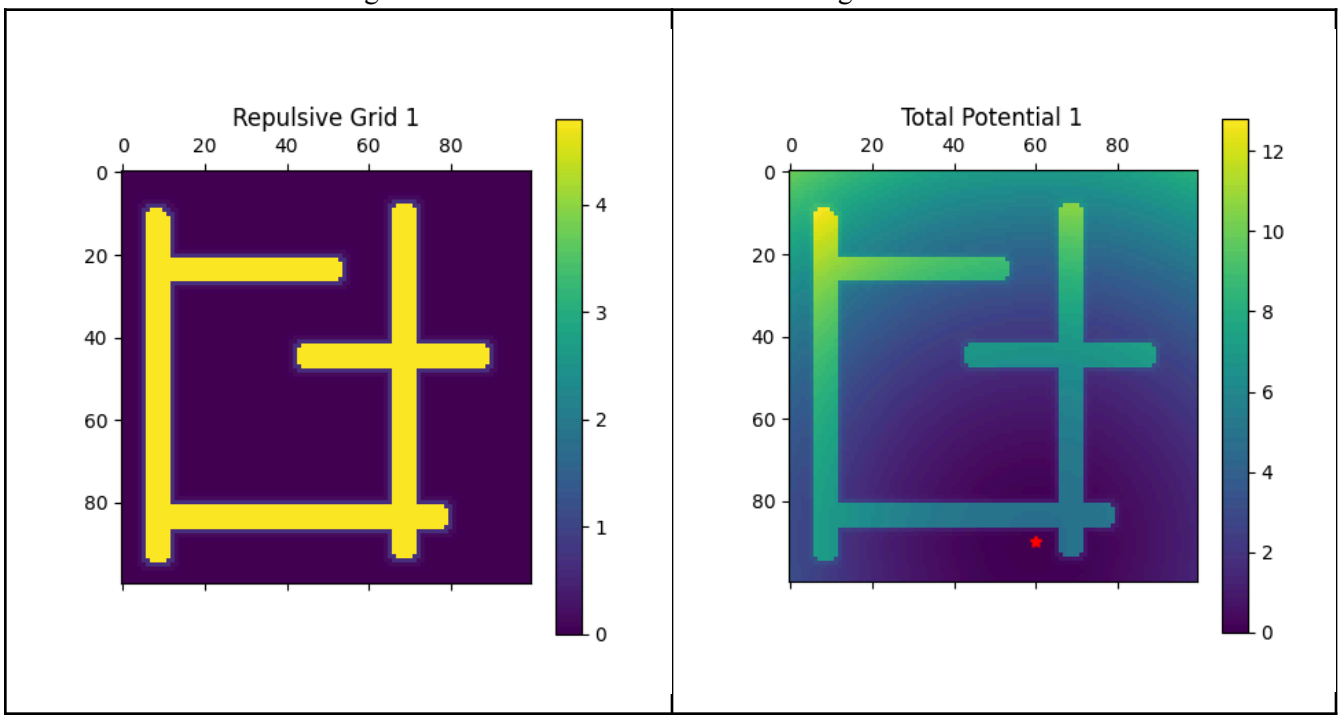


Fig. Repulsive & Total Potential with neighbours = 8

1.1.3. Results on Map 2:

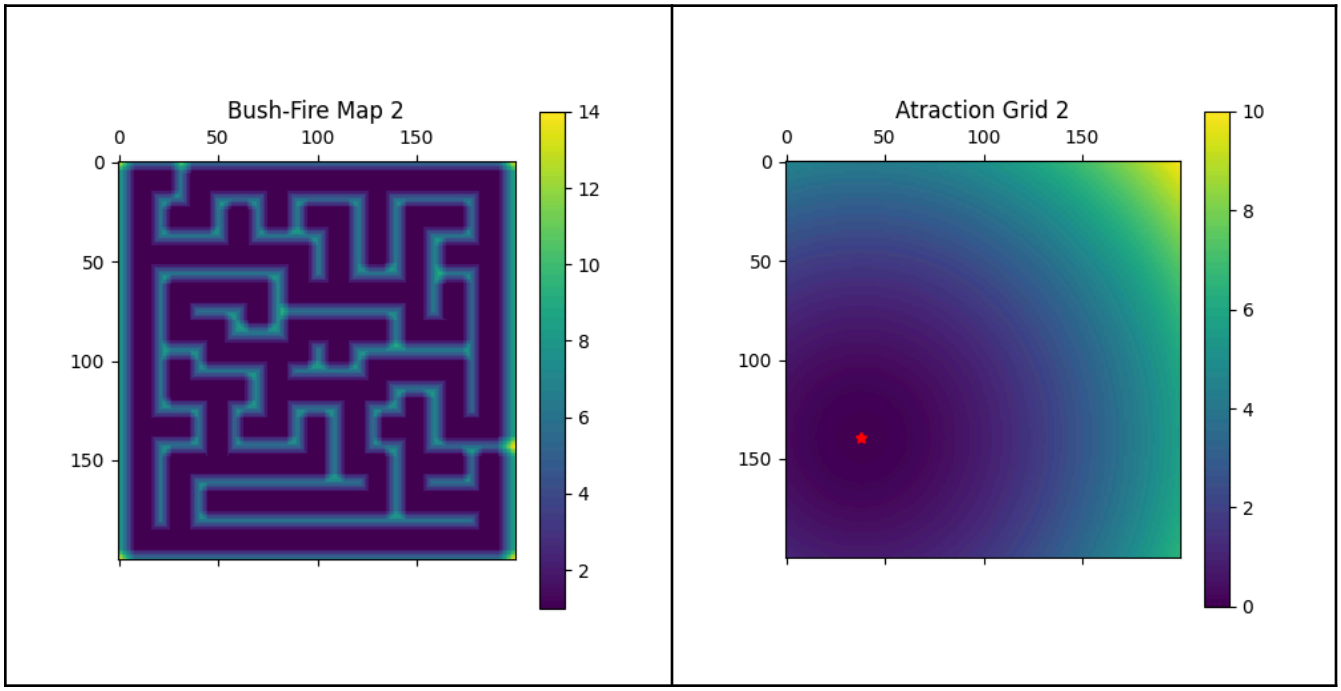


Fig. Bush-Fire & Attraction Grid with neighbours = 4

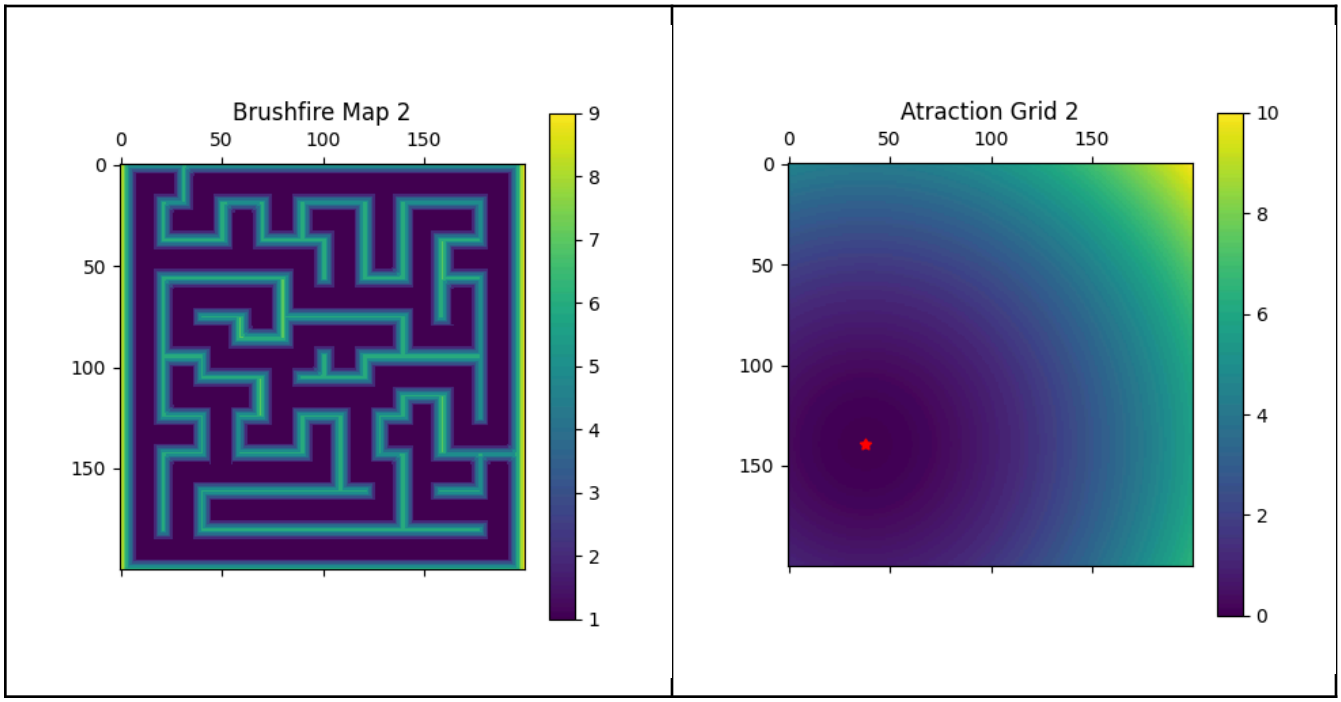


Fig. Bush-Fire & Attraction Grid with neighbours = 8

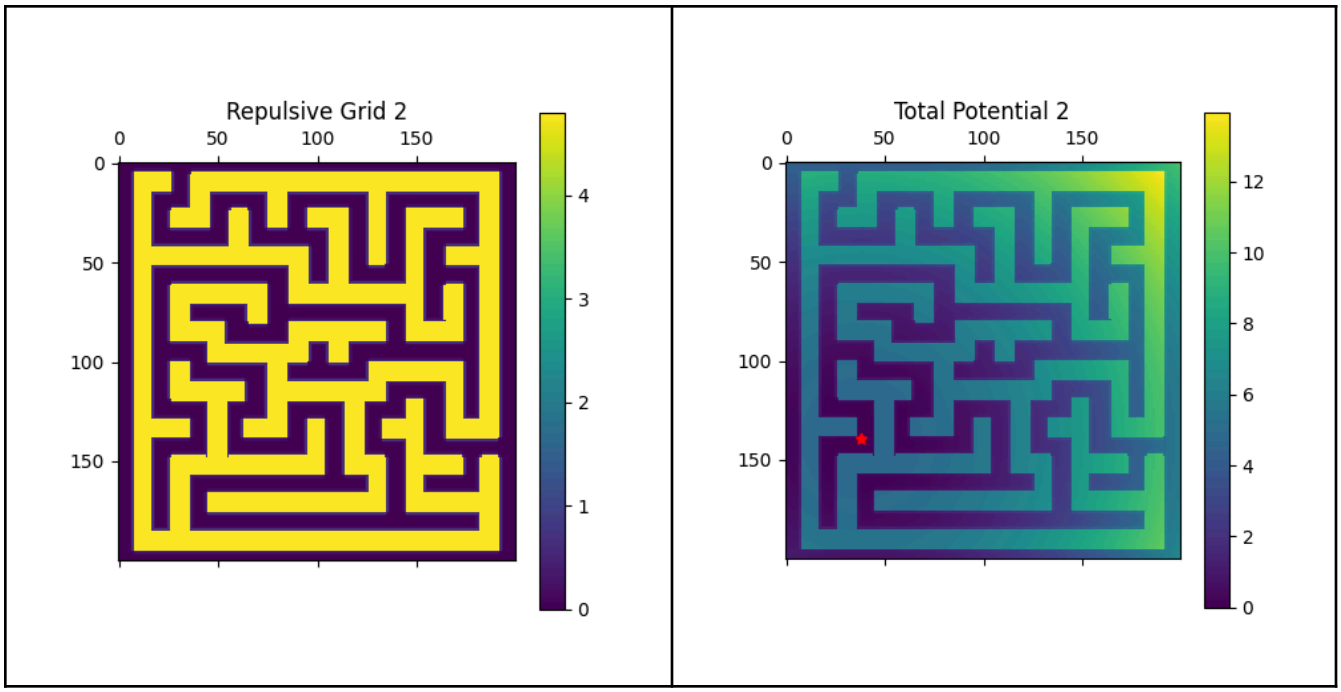


Fig. Repulsive & Total Potential with neighbours = 4

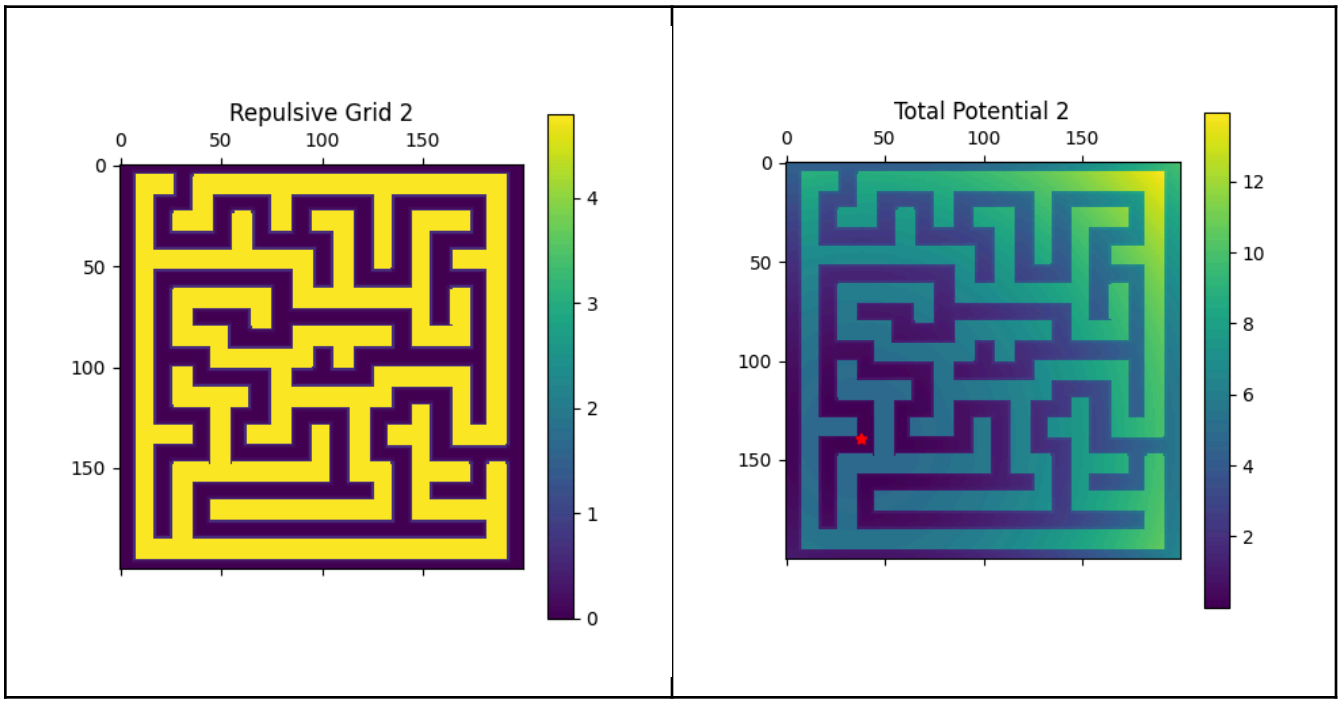


Fig. Repulsive & Total Potential with neighbours = 8

1.1.4. **Results on Map 3:**

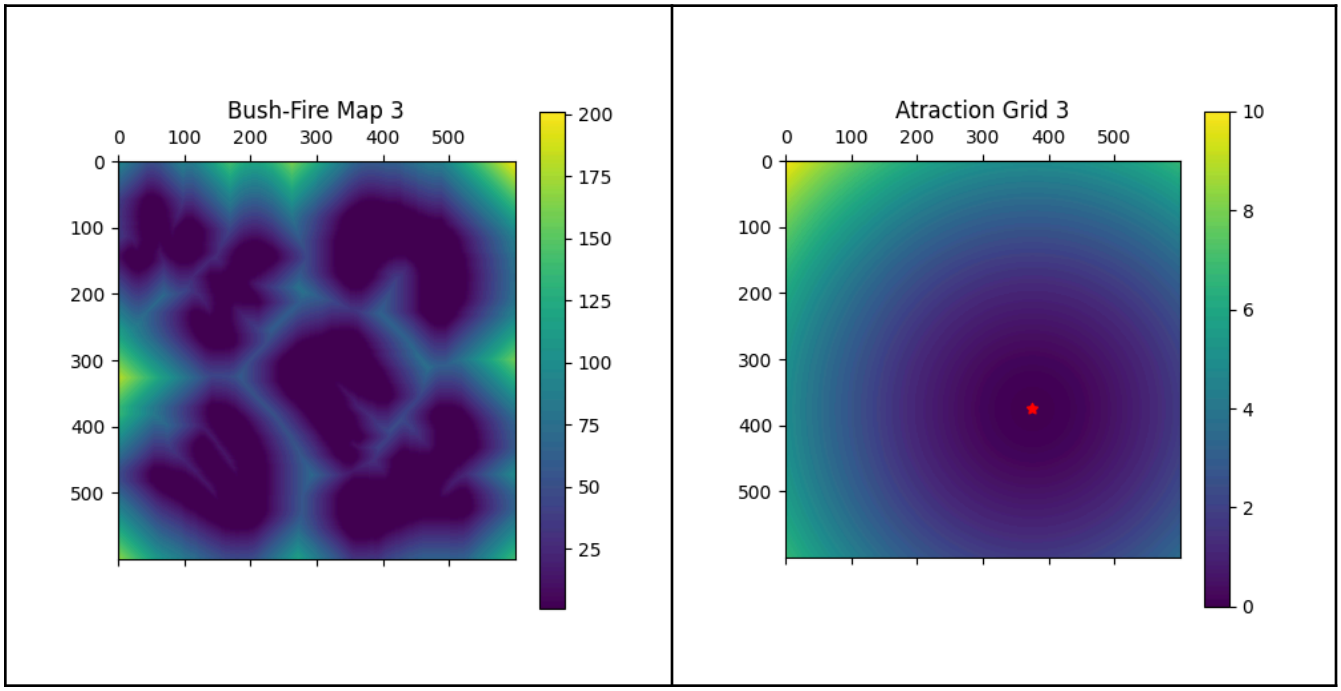


Fig. Bush-Fire & Attraction Grid with neighbours = 4

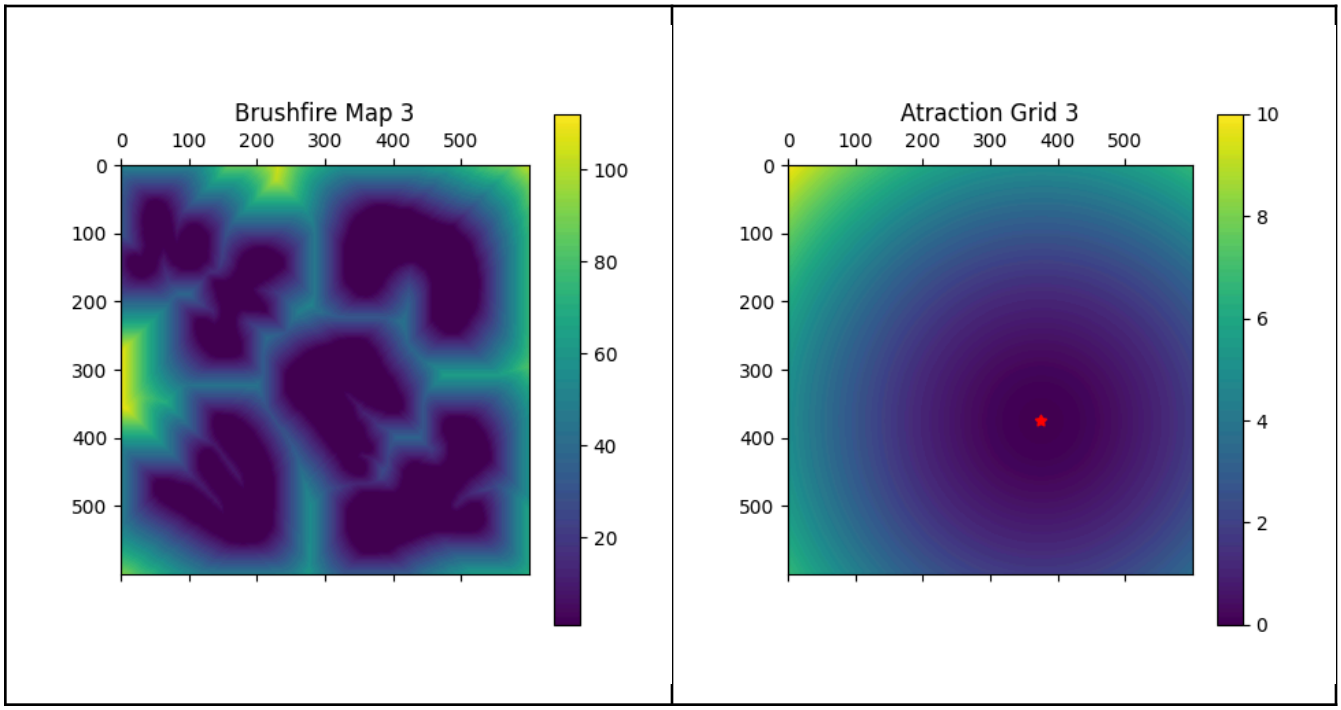


Fig. Bush-Fire & Attraction Grid with neighbours = 8

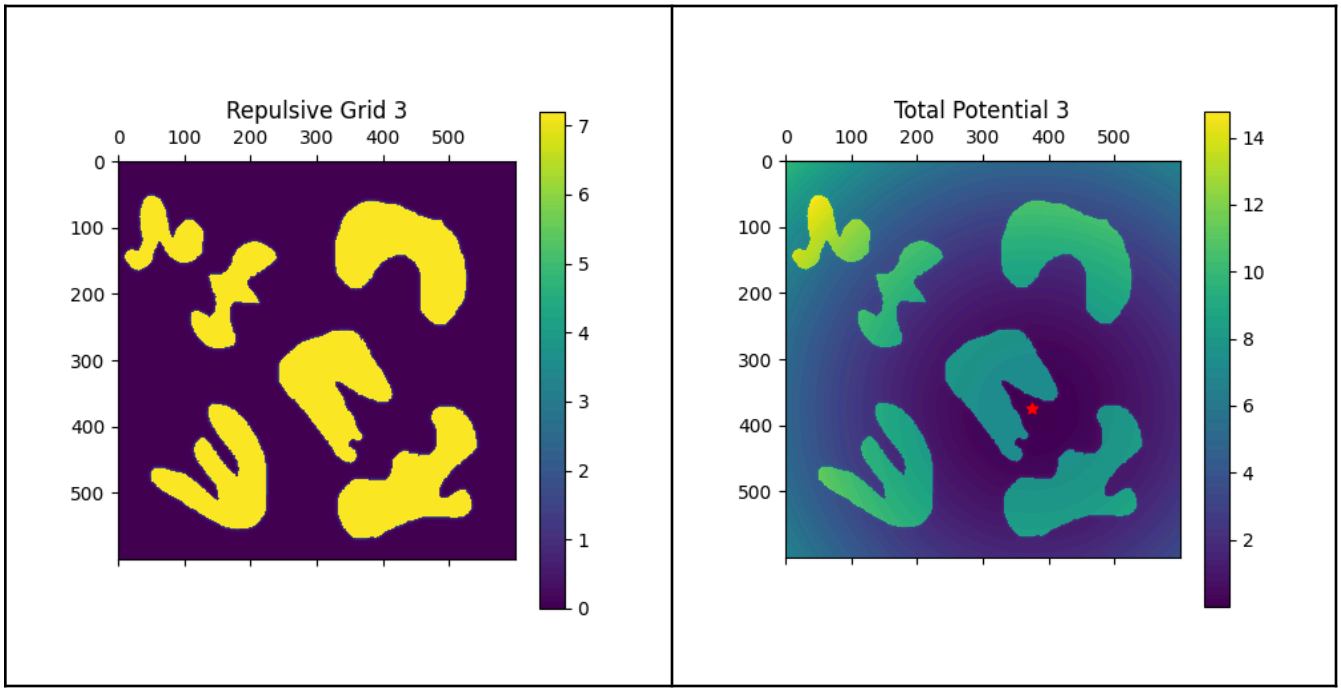


Fig. Repulsive & Total Potential with neighbours = 4

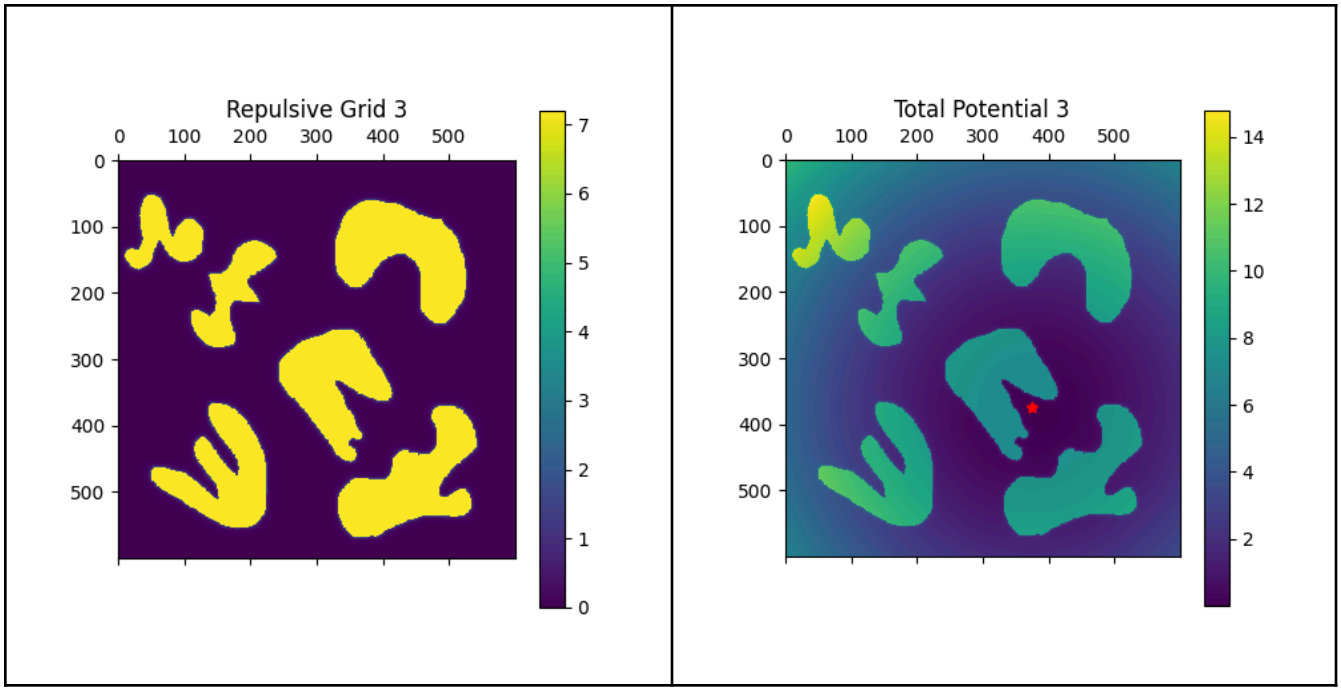


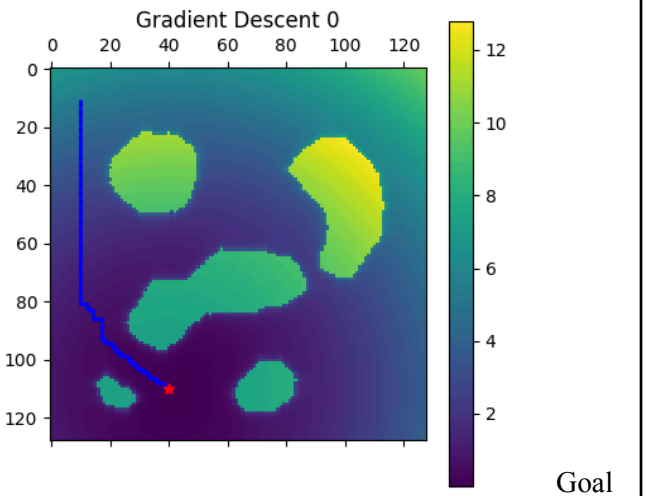
Fig. Repulsive & Total Potential with neighbours = 8

1.2. Part 2 (Gradient Descent):

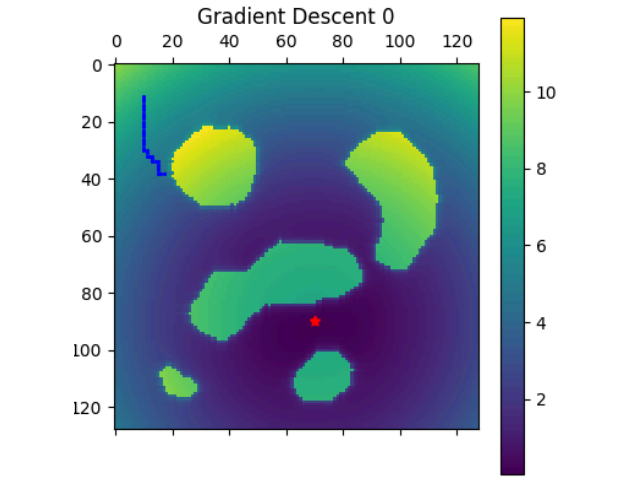
The gradient descent algorithm uses the total potential field to find the path from current position to goal.

1.2.1. Results on Map 0:

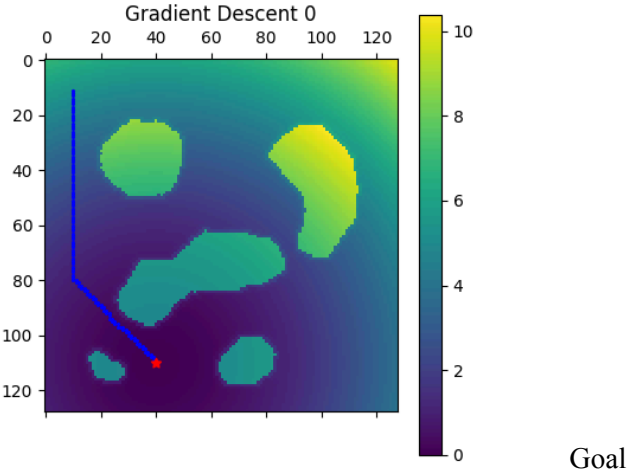
As gradient descent is not an optimal algorithm, It may be stuck in local minima, as shown in figure with Goal{90,70}.



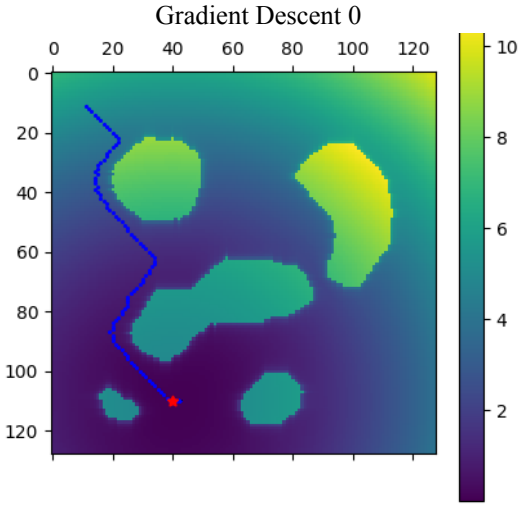
Goal {110,40} Neighbours = 4



Goal{90, 70} Neighbours = 4

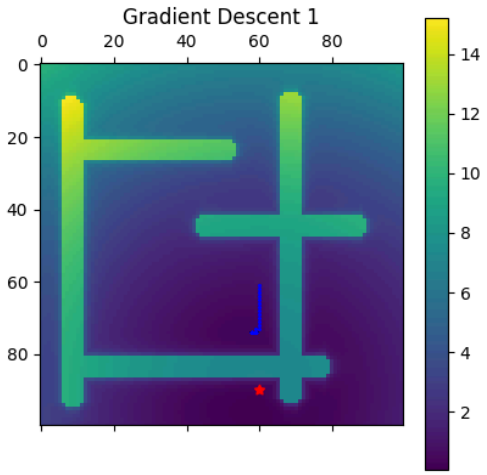


{110,40} Goal Neighbours = 8

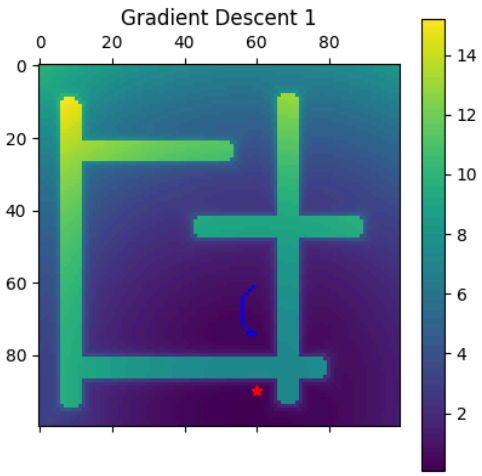


	Goal{90, 70}	Neighbours = 8

1.2.2. Results on Map 1:

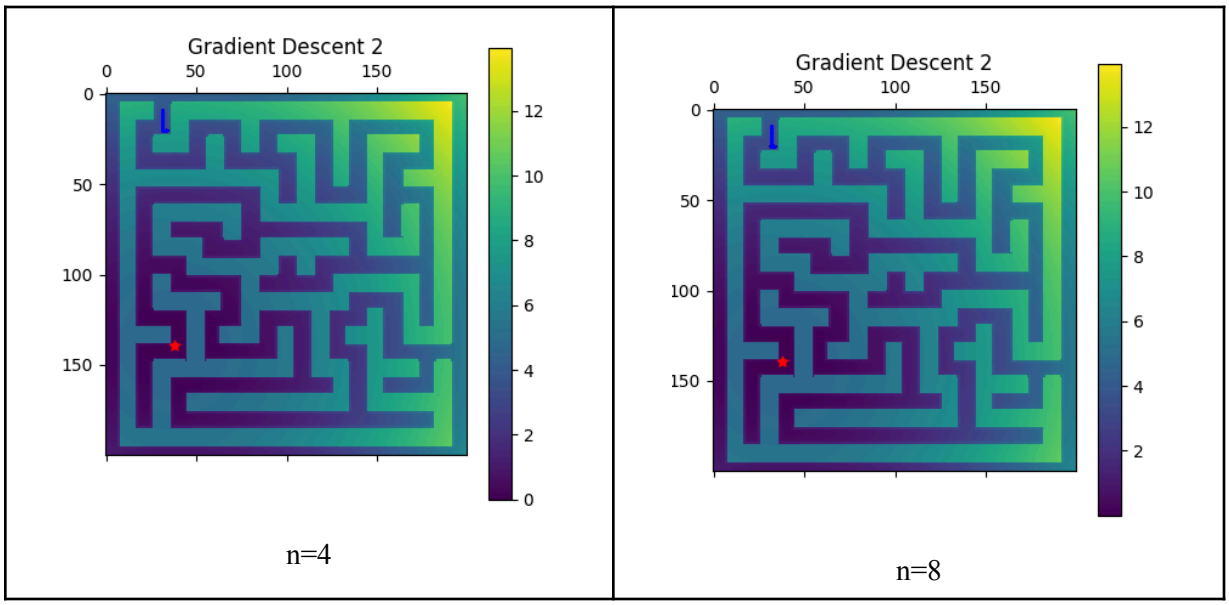


n=4



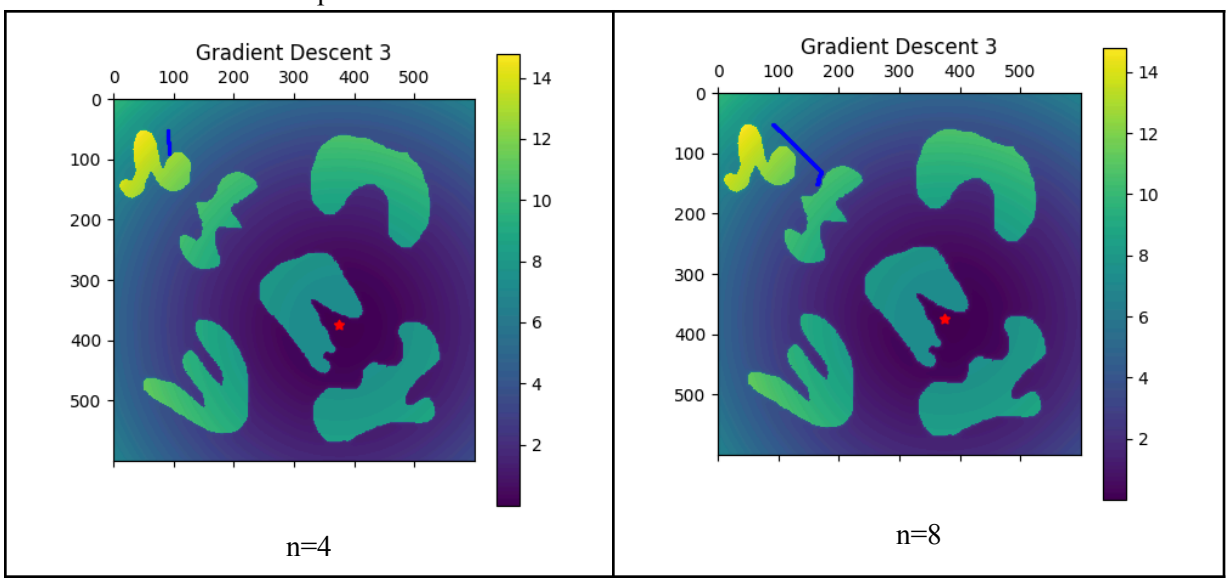
n=8

1.2.3. Results on Map 2:



n=4 n=8

1.2.4. Results on Map 3:



n=4 n=8

1.3. Part 3 (Wave-Front Planner & Path Finder)

The wavefront map illustrates distances from the goal point (red star), with darker colours near the goal and brighter colours farther away. This gradient guides a robot to navigate from the start toward the goal while avoiding obstacles.

1.3.1. Results on Map 0:

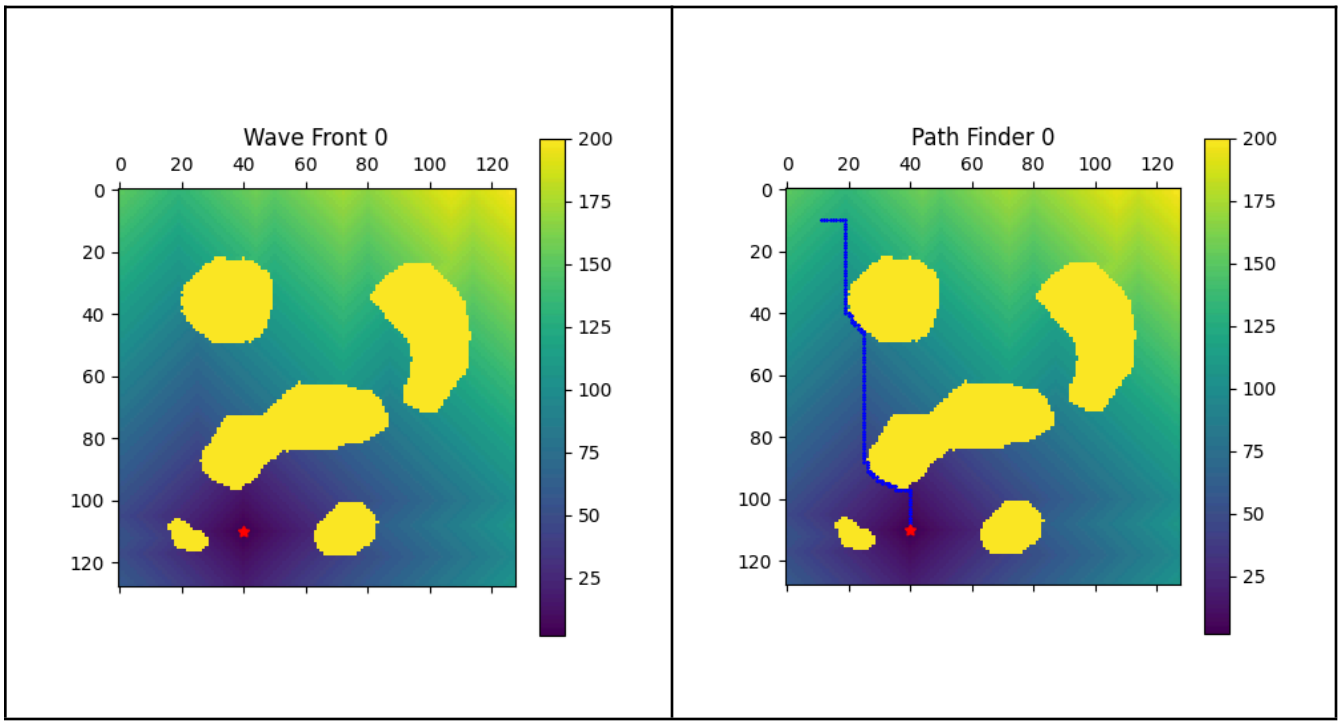


Fig : Wavefront planner with 4-neighbours

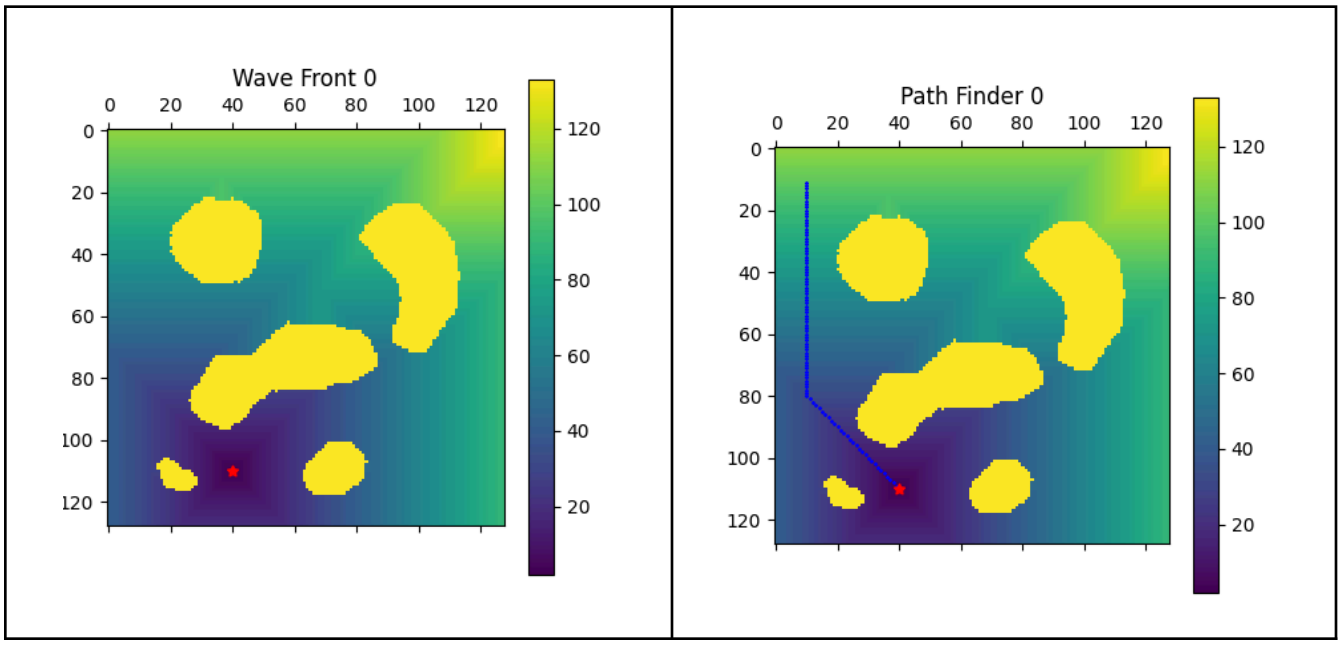


Fig : Wavefront planner with 8-neighbours

1.3.2. Results on Map 1:

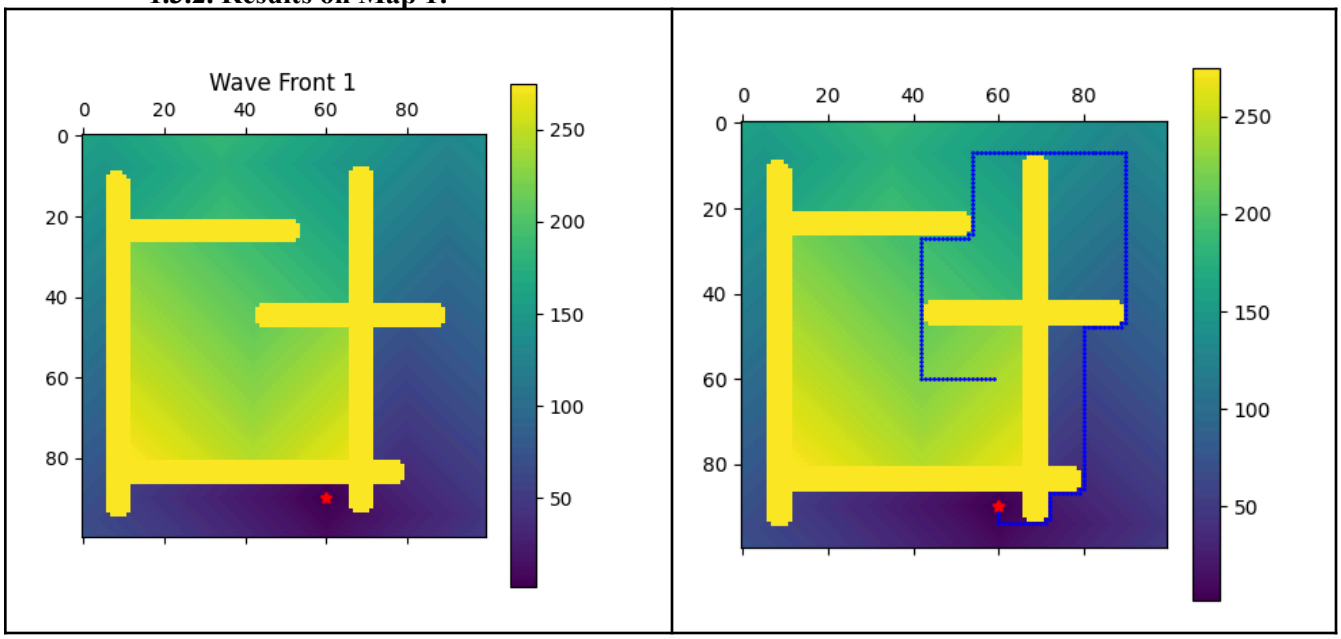


Fig. Wavefront and path finder with 4 neighbours

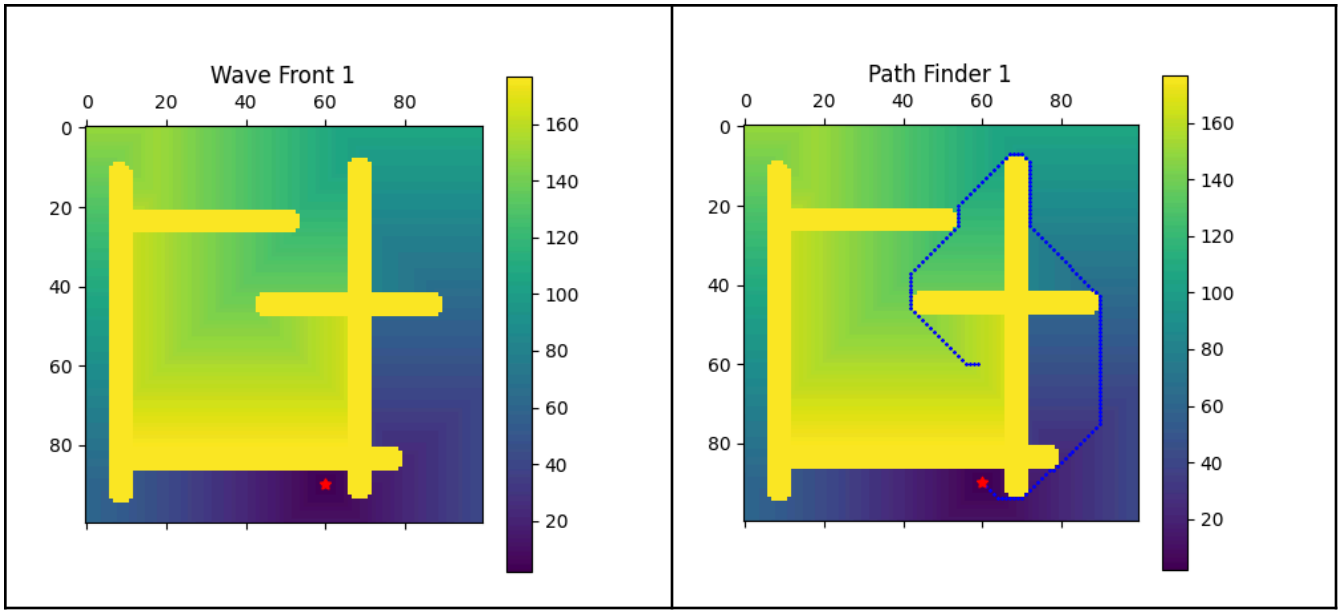


Fig. Wavefront and path finder with 8 neighbours

1.3.3. Results on Map 2:

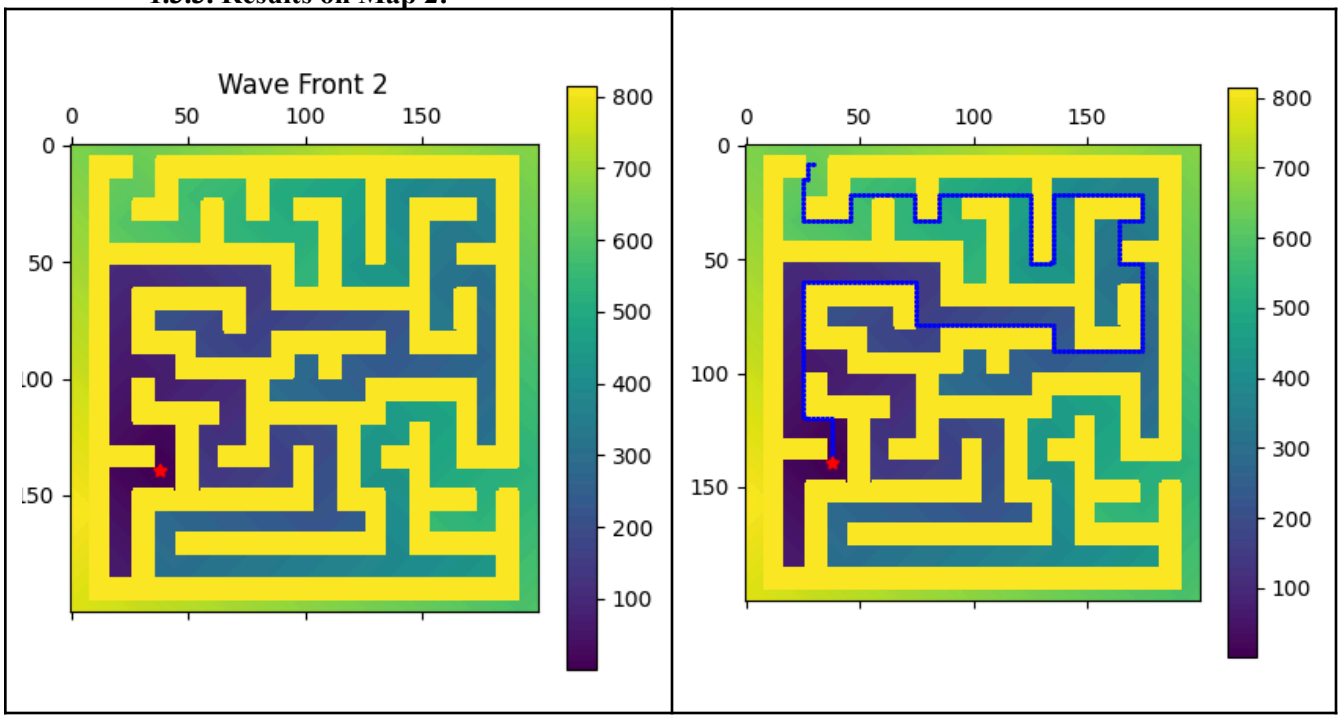


Fig. Wavefront and path finder with 4 neighbours

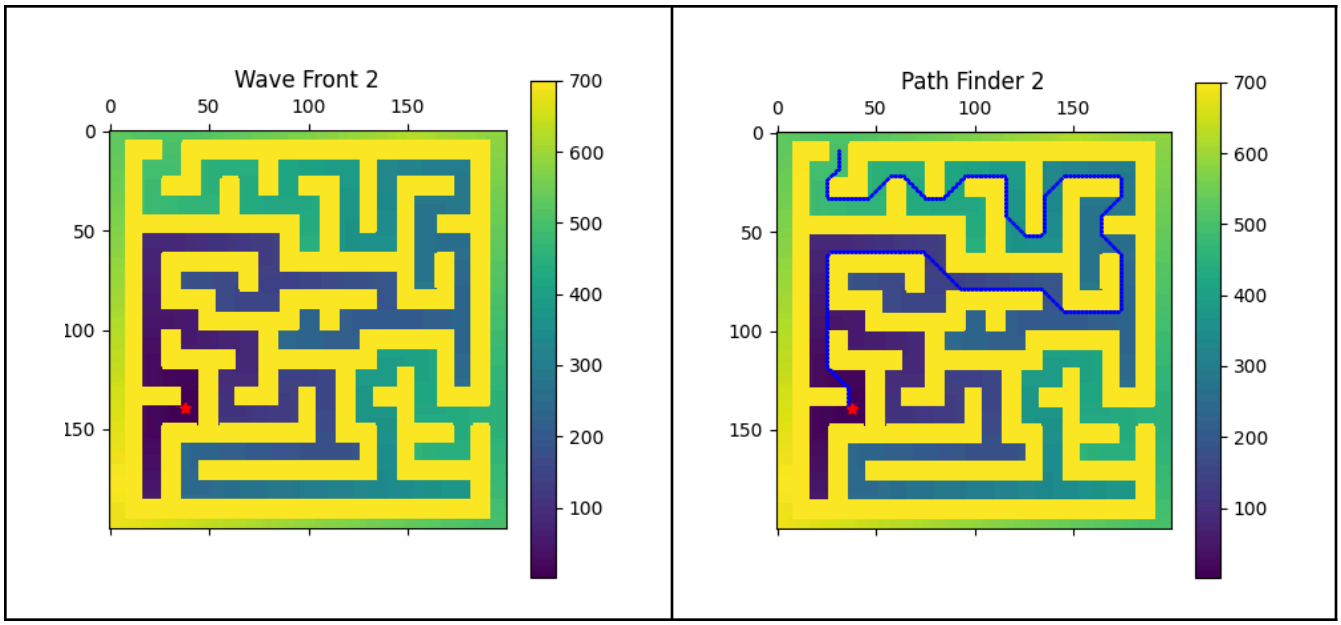


Fig. Wavefront and path finder with 8 neighbours

1.3.4. Results on Map 3:

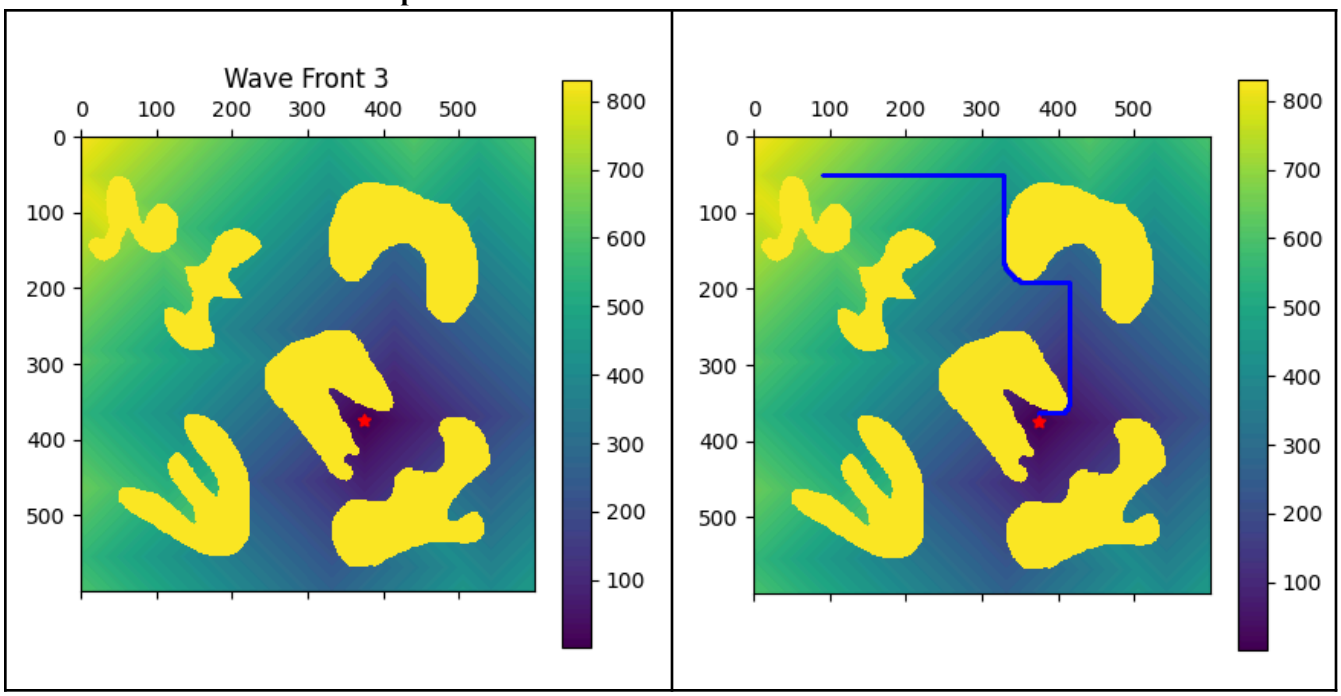


Fig. Wavefront and path finder with 4 neighbours

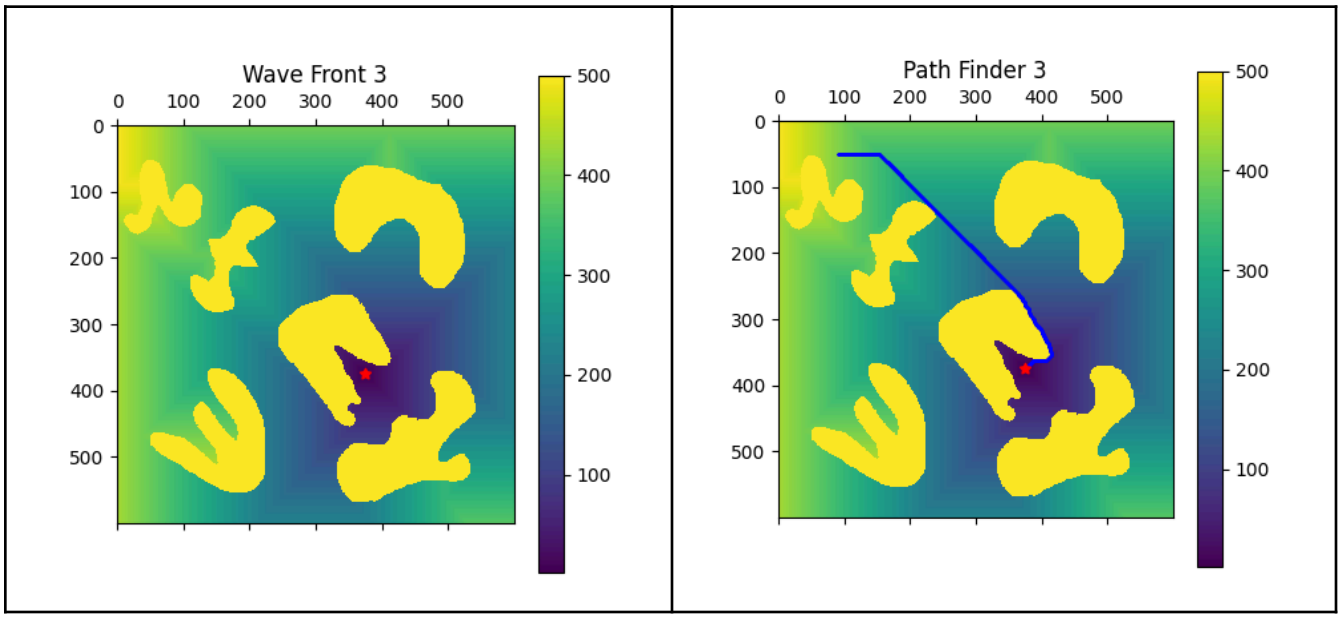


Fig. Wavefront and path finder with 8 neighbours

2. Implementations:

This section details the logic of the implemented algorithms, outlining each computational step. Figure 2.1 offers an overview of the full implementation, visually summarising the workflow and highlighting connections between components.

2.1. Bush Fire:

The bush-fire algorithm was implemented with the following logic;

Initialise a list L with all positions in the map where value = 1 (obstacles).
2. While L is not empty:
- Pop the last position from L as the current cell.
- For each neighbouring cell of the current cell:
- If the neighbour's value is 0 (unvisited), set it to the current cell's value + 1 and add it to the front of L.

2.2. Potential Fields:

2.2.1. Attraction Field

Attraction Field function was implemented with the following logic;

1. Get map dimensions and goal coordinates.
2. Create an empty attraction grid of the same size.

3. If distance is "d2", calculate quadratic Euclidean distance for each cell and update the grid.
4. Else if distance is "mht", calculate Manhattan distance for each cell and update the grid.

5. If apply_scaling is True, normalize the grid and multiply by the scale factor.

6. Return the attraction grid.

2.2.2. Repulsive Field

The repulsive field function was implemented with the following logic;

1. Get map dimensions and initialize a grid for repulsive potential values.
2. For each cell in the map:
- If the cell's distance is greater than Q:
- Set the repulsive value to 0.
- Otherwise:
- Calculate the repulsive potential using the function.
- Store the calculated repulsive value in the grid.
3. Return the repulsive potential grid.

2.3. Gradient Descent:

Gradient Descent function was implemented with the following logic;

1. Set a convergence threshold, initialize coordinates from the start point.
2. Initialize lists to store descent path coordinates, and set an initial difference for gradient check.
3. Set a counter to limit iterations.

4. While the difference is greater than the threshold:
- Find neighbouring cells based on the specified neighbour count.
- Create a temporary list to store neighbours' differences.

- For each neighbour:
- Calculate the difference between the neighbour and current cell value.
- Add the difference to the temporary list.

- Update the difference to the minimum value in the temporary list.
- Move to the neighbour with the smallest difference.
- Append the new coordinates to the descent path lists.

- If the counter exceeds the limit, exit the loop.
- Increment the counter.

5. Return the descent path coordinate lists.

2.4. Wave-Front Planner:

2.4.1. Wave-Front

The wave-Front algorithm was implemented with the following logic;

1. Create a copy of the map for wavefront values. Set the initial wavefront value at the goal cell.
2. Initialize a queue with the goal cell as the starting point.
3. While the queue is not empty:
- Increment the wavefront value for the new layer.
- Retrieve all cells at the current wavefront value.
- Initialize a list for neighbours of the next wavefront layer.

- For each cell in the current layer:
- Find the cell's neighbours.
- For each neighbour:
- If the neighbour is unvisited and not the goal:
- Assign the wavefront value to the neighbour.
- Add the neighbour to the next layer list.

- If there are valid neighbours, add them to the queue.

4. Set obstacle cells to a high value to mark them as untraversable.

5. If scaling is enabled, scale the wavefront values.
6. Return the updated wavefront map.

2.4.2. Path-Finder

Path-Finder algorithm was implemented with the following logic;

1. Initialize lists to store path coordinates.
2. While the start point is not equal to the goal point:
- Find neighbours of the current position.
- Select the neighbour with the smallest wavefront value (closest to the goal).
- Update the start point to this neighbour.
- Append the new position coordinates to the path lists.

3. Return the path coordinate lists.

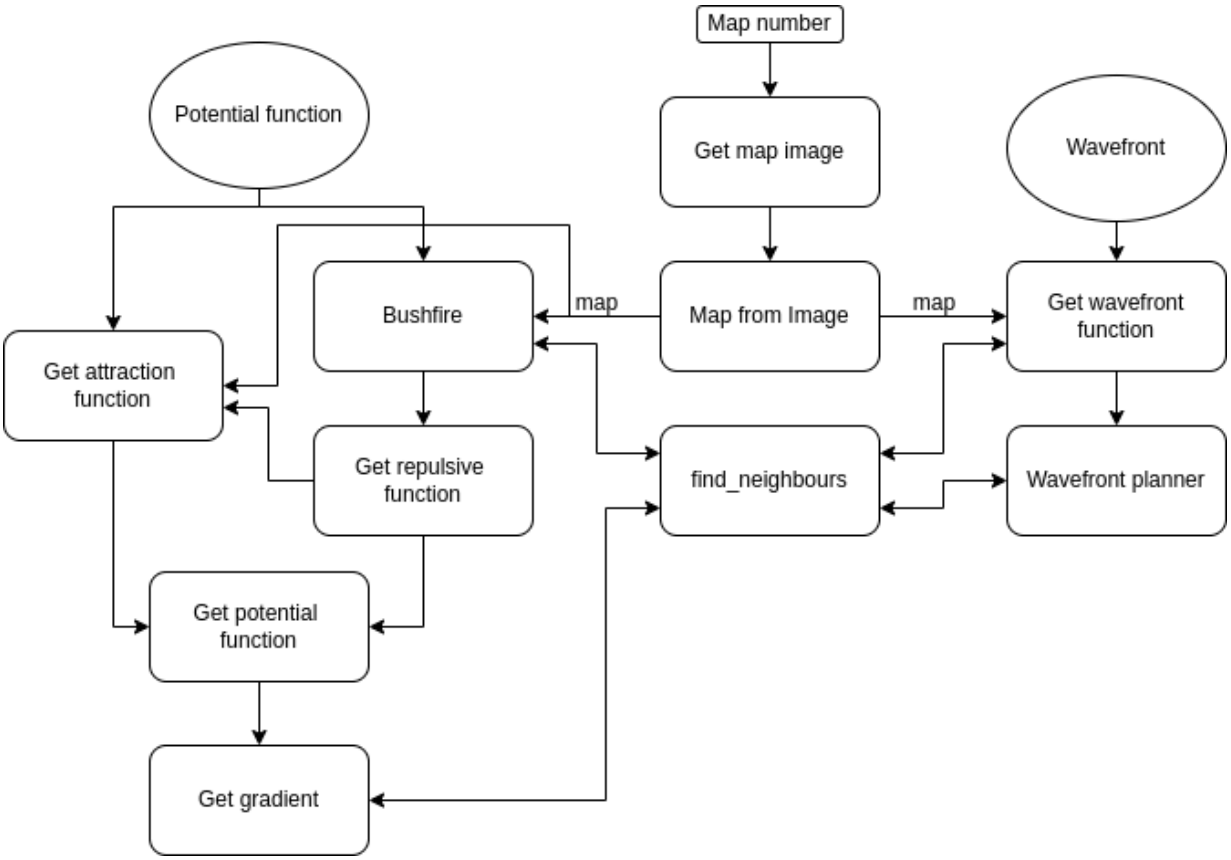


Fig.: The Overview of Implementation

2.5. Tuning parameter Q and deciding connectivity for each map:

For the first map we started with a pretty big Q value 100 which gave a not straightforward path to the goal, it had some imperfections as can be seen in the figure 2.2.

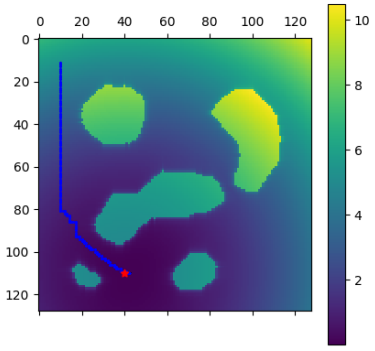


Fig. map 0, gradient descent connectivity 4 with Q = 100

So we decided to increase it to Q = 5. Giving a much better result, as seen in figure 2.3.

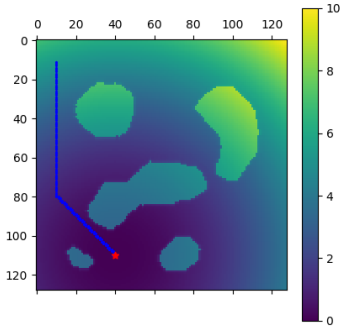


Fig. map 0, gradient descent connectivity 4 with Q = 5

However for connectivity 8 the results were different, when doing the Q = 5 the model was getting stuck in a local minima as seen in figure 2.4. And the Q = 100 was working fine.

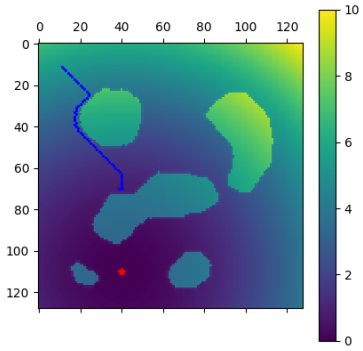


Fig. map 0, gradient descent connectivity 8 with Q = 5

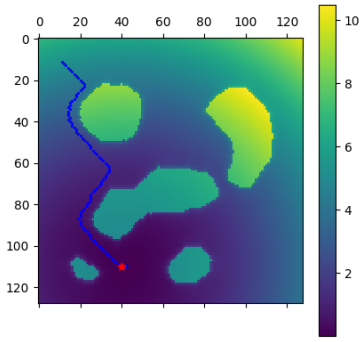


Fig. map 0, gradient descent connectivity 8 with Q = 100

For the rest of the maps these values are not relevant, since due to the topography of the map the model will always get stuck into a local minima no matter the parameters that are used.

3. Challenges:

3.1. Challenge one (Maps used on the examples and maps given to compute the algorithms were not the same):

The maps provided to test the algorithms did not have the same shape as the maps used for the examples. The example ones did not have any border. To match them we created the *remove_edges* function:

```python
def remove_edges (map, edge_size = 3):
    #   map: 2D n x m np.array from the map
    #   edge_size = int 3 as default, should be of the size of the wall

    # return a 2d n x m np.array with the first and last edge_size columns and rows = 0

    height, width = map.shape

    map[0:edge_size, :] = 0                 # first rows
    map[width-edge_size:width,:] = 0        # first columns
    map[:, 0:edge_size] = 0                 # last columns
    map[:,height -edge_size:height] = 0     # last rows

    return map
```

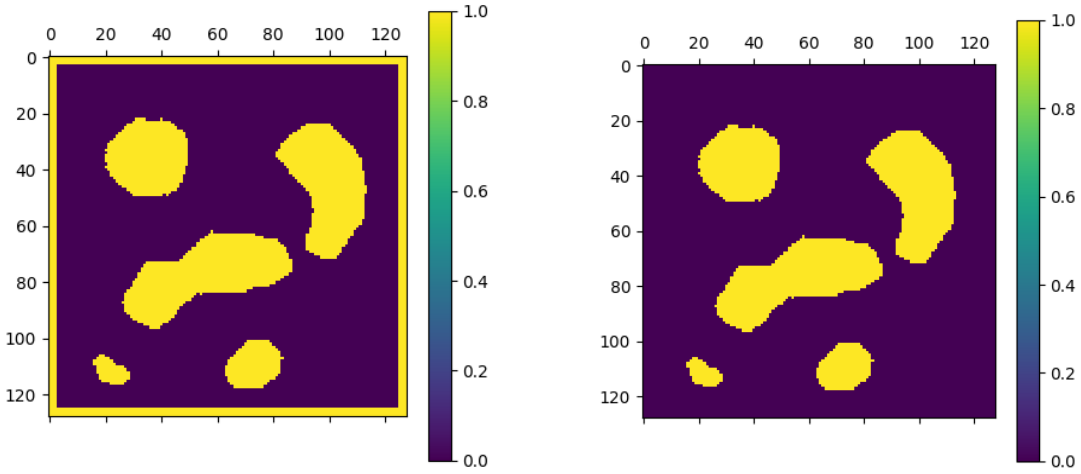


Fig. map0 with and without edges

3.2. Challenge two (Attraction function values):

When calculating attraction values across the grid, we observed quadratic growth, resulting in disproportionately large values for larger maps due to increased distances. To address this, we normalised all grid values and applied a scaling factor, ensuring they stayed within the desired maximum range:

```python
if apply_scaling == True: # normalize and multiply by the scale factor
    attraction_grid = attraction_grid/attraction_grid.max()*scale_factor
```

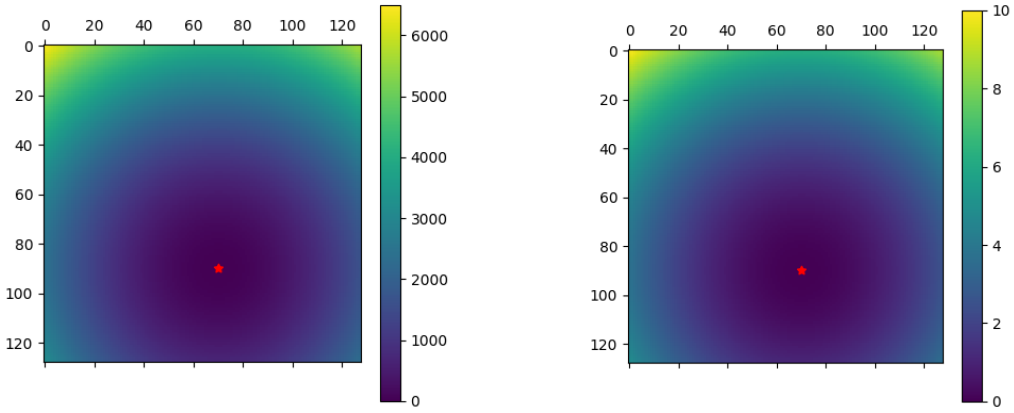


Fig. map0 attraction function normalised and unnormalised (see scaling)

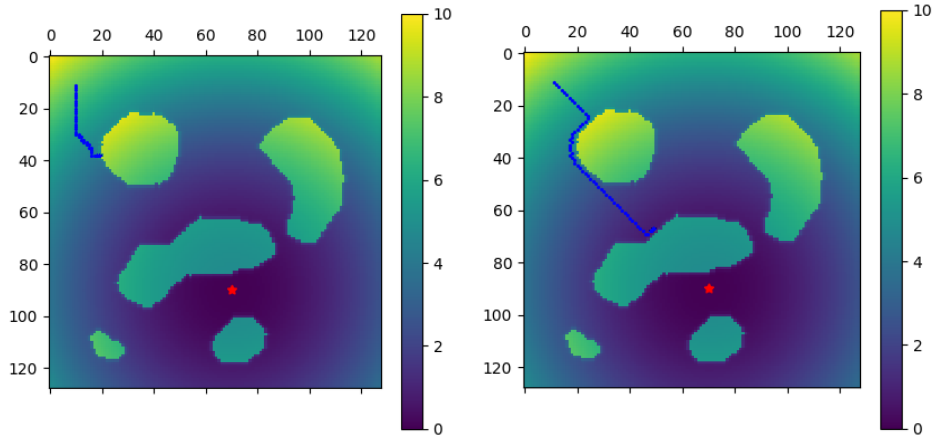
3.3. Challenge three (Neighbours out of the map):

When searching for neighbouring cells of a given cell, some neighbours fell outside the map boundaries, causing errors in the code. While using a clipping function could prevent this, we opted for conditional logic to manage it instead:

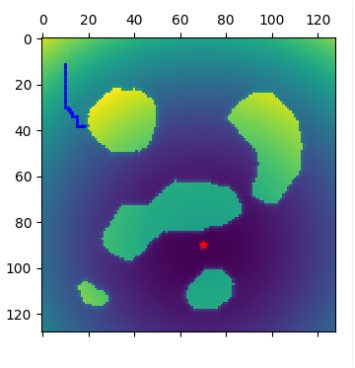
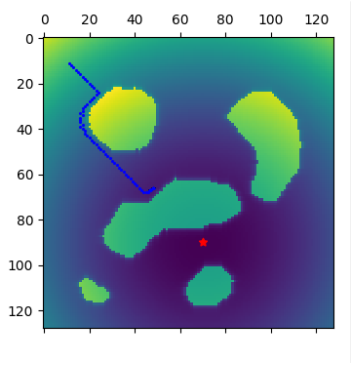
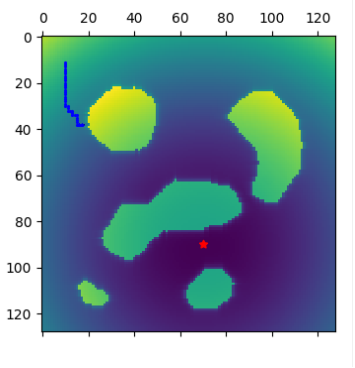
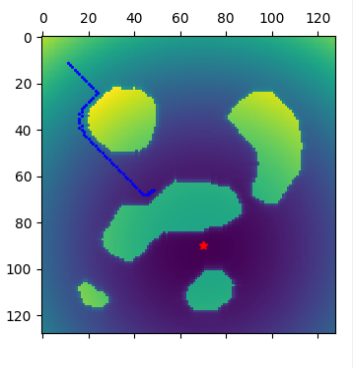
```
height, width = map.shape   # get dimensions of the map to define if a neighbour is valid or not
y,x = position              # depack the psotion
neighbours = []             # define empty list of neighbours

if neighbours_num == 4 or neighbours_num == 8:


    if neighbours_num == 4:                          # for connectivity 4
        if x-1 >= 0 :
            neighbours.append((position[0],position[1]-1))      # left

        if y-1 >= 0:
            neighbours.append((position[0]-1,position[1]))      # up

        if x+1 < width:
            neighbours.append((position[0],position[1]+1))      # right

        if y+1 < height:
            neighbours.append((position[0]+1,position[1]))      # down
```

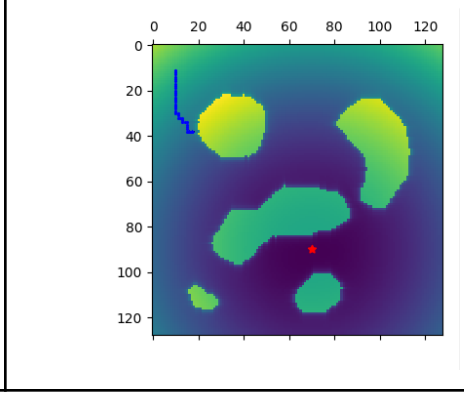
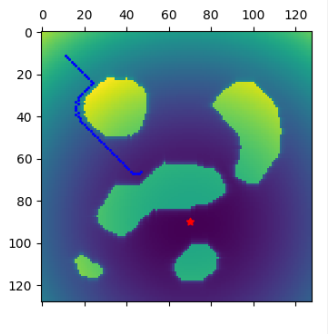
3.4. Challenge four (Local minimal in potential fields):

In some configurations of start/goal positions, repulsion range, and obstacle layout, the potential field may have local minima. These are points where the gradient descent method gets trapped, balancing attractive and repulsive forces, resulting in failure to reach the goal. Examples include being stuck at a local minimum with both 4 and 8 connectivity.

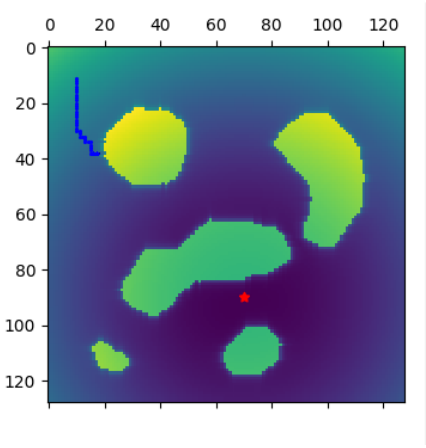
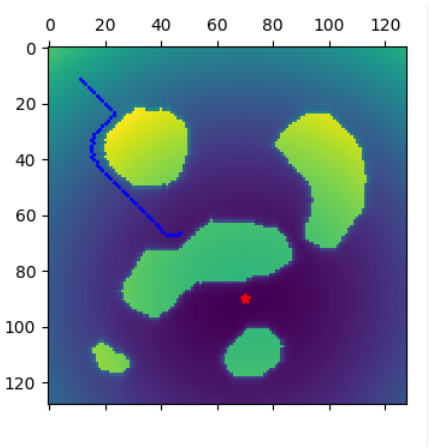


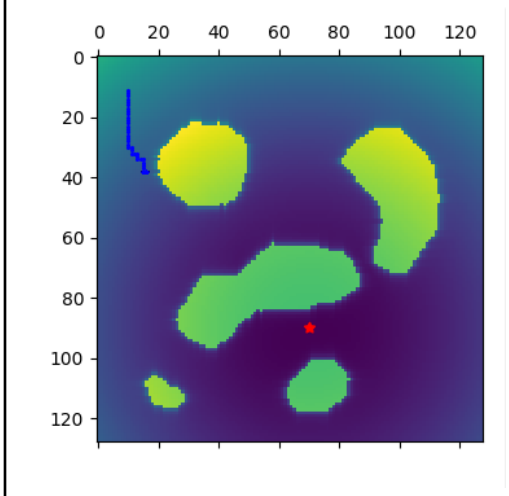
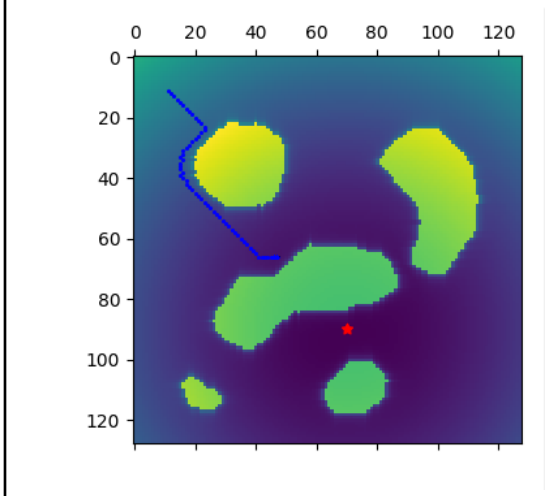
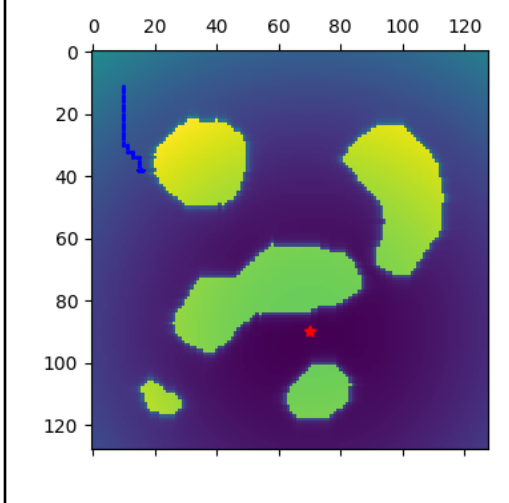
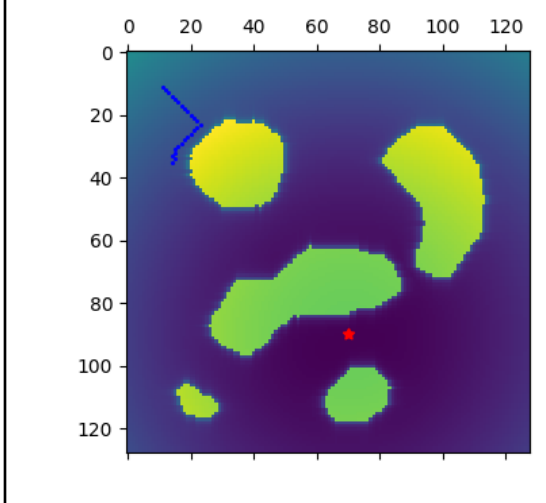
As a first approach, we tried to increase the range in which the repulsive function is applied aiming for a bigger range to avoid the local minima. In the previous example, the Q (distance where the repulsive value is not zero) was 5.

Q	Connectivity 4	Connectivity 8
10		
15		

20		

It is evident that this value is not affecting the local minima, however, if it is over increased the spaces between obstacles might get blocked. So we started to experiment with the repulsion force strength instead of the repulsion range. In the original example the Eta (repulsive force) was 15

Eta	Connectivity 4	Connectivity 8
20		

25		
35		

After modifying the Eta parameter we see that the gradient descent is not just not working if not is even behaving worse than before. So we proceed to modify the Eta and Q values at the same time. After doing this the same result appeared, the model wasn't able to get to the goal. This, as mentioned before, is due to the unavoidable problem of local minima present in potential fields.

4. Conclusion:

This lab report evaluates path planning algorithms, including the Bush-Fire Algorithm, potential fields, and the Wave-Front Planner. Each demonstrated strengths in guiding the robot toward the goal while avoiding obstacles. The Bush-Fire Algorithm effectively mapped distances, aiding potential field calculations, though local minima occasionally trapped the agent in non-goal positions. Parameter adjustments helped but risked blocking valid paths. The Wave-Front Planner proved more robust, reliably guiding the agent even in complex environments.