Biomedical Data Analytics Midterm: Finding the optimal trajectory using Wavefront Planning

Aly Khaled
1190156
aly.othman01@eng-st.cu.edu.eg

Maryam Moataz
1190522
maryam.fathy01@eng-st.cu.edu.eg

Mohamed Nasser
1190438
mohamed_gaafar@stud.cu.edu.eg

Mariam Aly
1190519
mariam.aly00@eng-st.cu.edu.eg

Cairo University, Faculty of Engineering, Biomedical Department

Abstract

This project aims to find the optimum collision free path(s) from a point to another one in a two dimensional (2D) static environment with fixed obstacles. For this objective the Wave Front algorithm is considered to tackle the problem successfully, as it gives optimal and fast results.

1 Introduction

The Wavefront algorithm finds a path from point S (start) to point G (goal) through a discretized workspace such as this (0 designates a cell of free space, 1 designates a cell fully occupied by an obstacle), The algorithm consists of a breadth-first search (BFS) on the graph induced by the neighbourhood connectivity (adjacency), As BFS traverses the space, each cell is assigned a value which corresponds to the number of moves required for the shortest path from that cell to the goal.

2 Problem

Given a 2D environment, containing some obstacles and a start and end point provided as input by the user, we were required to find the optimal trajectory towards the goal (end point), and output the weighted value map and trajectory map. Then visualise them in the form of a coloured image.

3 Steps

3.1 Step 1: Create a Discretized Map

Create a 2D array that contains the empty spaces and the static obstacles. Then define the end point (goal).

3.2 Step 2: Fill in Wavefront

Traverse through the array using a queue using the breadth first search algorithm as inspiration, we check node by node, starting at the goal. Ignore obstacles, look at nodes around your target node, then count up by adding a one to the neighbouring cells then incrementing the goal to be the old goal +1 and dequeuing the old goal that way we can move through the array and fill all of it. Of course, an if condition that eliminates any movement or involvement with the obstacles is important.

```
def wavefront_map(map, goal_row, goal_col):
    moves = [[ 0, 1], [0, -1], [1, 0],
             [-1, 0], [1,  1], [1, -1],
             [-1, 1], [-1, -1] ]

    queue = []
    # Start from the goal location
    ↪  (row,col,prevValue)
    queue.append([goal_row, goal_col, 2])

    while queue:
        current_row, current_col, prev_value =
        ↪  queue.pop(0)
        for move in moves:
            new_row = current_row + move[0]
            new_col = current_col + move[1]

            if  new_row < 0 or new_row >= len(map)
                or new_col < 0 or new_col >=
                ↪  len(map[0])
                or map[new_row][new_col] == 1
                or map[new_row][new_col] != 0:
                continue

            queue.append([new_row, new_col,
            ↪  prev_value+1])
            map[new_row][new_col] = prev_value+1

    return map
```

3.3 Step 3: Create a Discretized Map

After defining an array that contains all the possible moves like up, down, upper left, etc, ordered according to the fixed priority. We implemented a function that takes the map as a 2D array of 0's and 1's, row index of start, and column index of start. The function then checks for the validity of the chosen start and end points as they cannot be a boundary (obstacle;1), the start cannot equal a 2 as that means that the path is equal to 0 as the start and end are equal, nor can the start and end exceed the boundaries of the size of the given array. Inside the function is a while loop that breaks once the current value = 2, indicating we have reached our goal. For every current value we get its position in terms of rows and columns and then get all the values of its 8-neighbouring cells. We then choose the neighbouring cell with the minimum value that also agrees with our defined priority system and append its index into our trajectory array. The output of this function is our trajectory which is the shortest path from our start point to the goal.

```
def backtracking(map, start_row, start_col):
    moves = [[-1, -1], [1, -1], [1, 1], [-1, 1],
             [0,  -1], [1,  0], [0, 1], [-1, 0]]
    trajectory = []
    if map[start_row][start_col] == 0:
        print("No Solution")
        return trajectory
    current_row, current_col = start_row, start_col
    trajectory.append((current_row, current_col))
    current_value = map[current_row][current_col]
    while current_value != 2:
        current_value = map[current_row][current_col]
        minRow = current_row
        minCol = current_col
        for move in moves:
            new_row = current_row + move[0]
            new_col = current_col + move[1]

            if new_row < 0 or new_row >= len(map)
            or new_col < 0 or new_col >= len(map[0])
            or map[new_row][new_col] == 1:
                continue

            if current_value >=
            ↪  map[new_row][new_col]:
                current_value = map[new_row][new_col]
                minRow, minCol = new_row, new_col
        current_row, current_col = minRow, minCol
        trajectory.append((current_row, current_col))
    return trajectory
```

3.4 Step 4: Visualizing

For visualisation we used two methods, one for large size maps (over 200) and another for the smaller sized maps. Large sized maps were displayed as image values using "pillow" (python imaging library). The function scans along the image and sets pixel colours depending on whether the map value is a barrier (1) a goal (2) or else (0). This is then enlarged by a factor of 10 for better visibility, and then output as a .png file in the directory. The same is done for smaller images however smaller images have the wavefront algorithm values displayed with their corresponding numbers instead of as coloured pixels only using Matplotlib library.

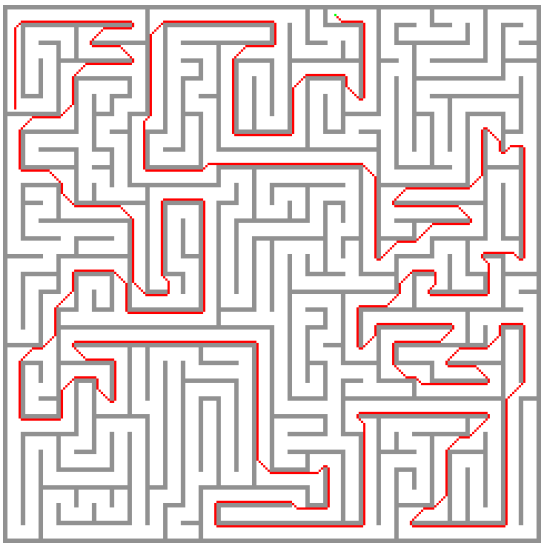


Figure 1: Large map with size (242 x 242) by Pillow

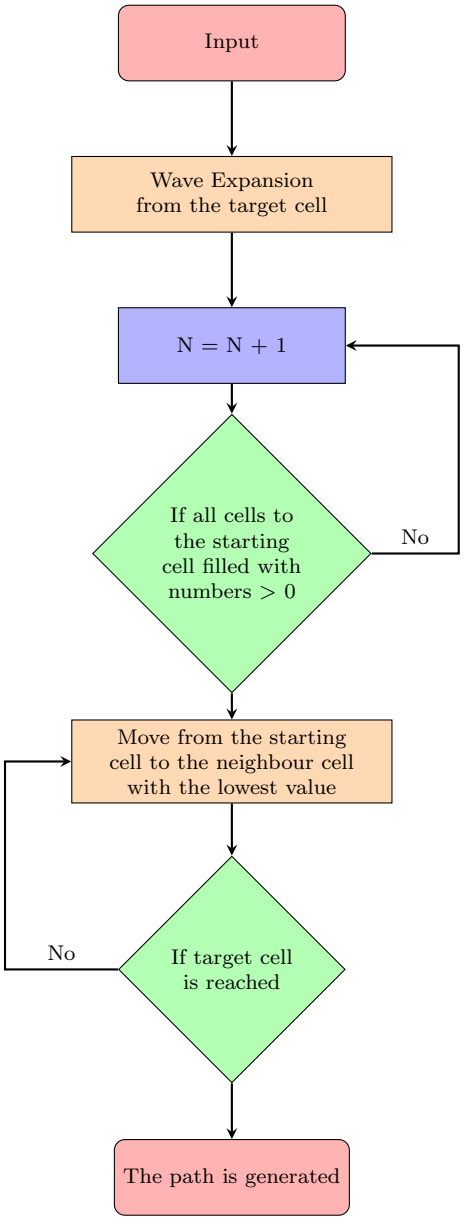
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	21	20	19	18	18	1	1	14	14	14	14	14	14	1	1	3	3	3	1
1	21	20	19	18	17	1	1	14	13	13	13	13	13	1	1	3	2	3	1
1	21	20	19	18	17	16	15	14	13	12	12	12	12	1	1	3	3	3	1
1	21	20	19	18	17	16	15	14	13	12	11	11	11	1	1	4	4	4	1
1	21	20	19	18	17	16	15	14	13	12	11	10	10	1	1	5	5	5	1
1	21	20	19	1	1	1	1	1	13	12	11	10	9	1	1	6	6	6	1
1	21	20	19	1	1	1	1	1	13	12	11	10	9	8	7	7	7	7	1
1	21	20	19	18	17	17	1	1	13	12	11	10	9	8	8	8	8	8	1
1	21	20	19	18	17	16	1	1	13	12	11	10	9	9	9	9	9	9	1
1	21	20	19	18	17	16	15	14	13	12	11	10	10	1	10	10	10	10	1
1	21	20	19	18	17	16	15	14	13	12	11	11	1	1	11	11	11	11	1
1	21	20	19	18	17	16	15	14	13	12	12	1	1	1	12	12	12	12	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 2: Small map displayed by Matplotlib

3.5 Step 5: Main loop (input/output)

For convenience, a main loop was created to allow user to be able to freely test the program, with 3 menu options: 1→ Reading a .mat file. 2→ Use a randomly generated test map. 0→ exit the program. Console output was handled using the print_output function where if the size of the map was too large the file would be saved instead for easier viewing. And the trajectory was displayed regardless. The exact required format was also taken into consideration.

4 Design



5 Testing & Validation

5.1 Generating random maps

We made a function that generates a random map of a given size. The function takes the size of the map as input and generates a random map of that size. The function uses the random library to generate a random number between 0 and 1. If the number is less than 0.3, the cell is an obstacle, otherwise it is a free cell. The function returns the generated map.

```
def generate_random_map(rows,cols):
    map = np.array([])
    map = np.random.randint(0, 2, (rows, cols))

    # set the goal location
    map[rows-1][cols-1] = 2

    return map
```

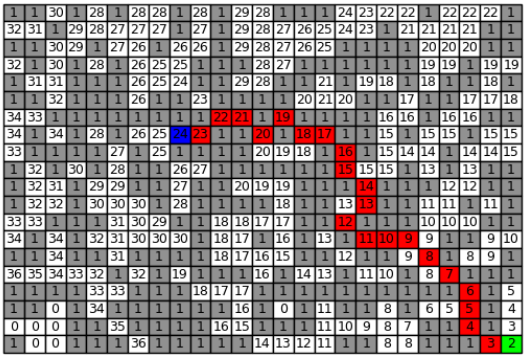


Figure 3: Randomly generated map

5.2 Testing with small map sizes

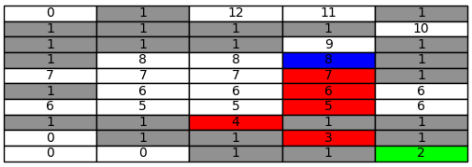


Figure 4: Randomly generated map with size (10 x 5)

5.3 Testing with large map sizes

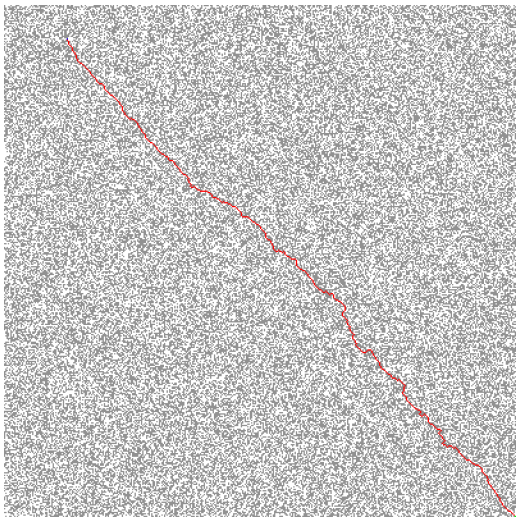


Figure 5: Randomly generated map with size (300 x 300)

5.4 Map with no solution

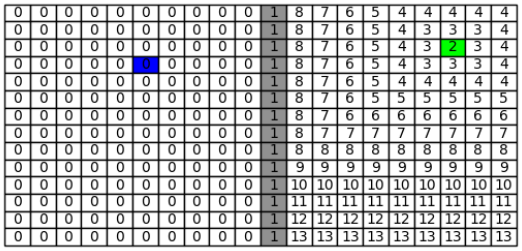


Figure 6: Map with no solution

6 Problems we faced

6.1 Problem 1: Implementation Wavefront

The first obstacle we faced when getting familiar with the implementation of the wavefront algorithm was the matrix filling (digitising the map), our understanding was that we had to start from the end point (the goal) and move through every consecutive 8 cell neighbourhood to calculate the number of steps away from the goal whilst also avoiding the obstacles represented as 1's in our matrix. At first we tried doing this by making a recursive function that would get the 8 neighbouring cells at each point and add 1 to the centre value, but that didn't seem quite right and was inefficient. We then opted to utilise the idea of the breadth first search (BST) algorithm as it seemed somewhat similar to the wavefront algorithm, this ended up working successfully.

6.2 Problem 2: Backtracking Moves Priority

The next, and main problem we faced was with backtracking to find the optimal trajectory; as we had to keep in mind the positional priority of the path.

```
moves = [[ 0,  1], [0, -1], [1, 0],[-1, 0],
         [1,  1], [1, -1], [-1, 1], [-1, -1] ]
```

figuring out the flow of iteration through the matrix to continuously calculate all the possible moves then getting the minimum while keeping in mind the priority was tricky at first and everytime we solved an error a new one came up but with organising our code in clear steps we worked it out using one for loop and an if condition.

6.3 Problem 3: Matrix Overflow

Another problem we faced was that multiple 2's were surprisingly showing up in our value map when there should only be one that represents the goal. Turns out that some array elements were overflowing the 8-bit unsigned integer type automatically allocated for them, so we opted to change them to 16-bit unsigned integers.

```
map = np.array(map)
map = map.astype(np.uint16)
```

6.4 Problem 4: Handling No Solution

The last problem we faced was that when there was no solution to the maze in another word when the goal was surrounded by obstacles, the program would run in infinite loop. We solved this by adding a condition to check if the starting point has value of 0 after the wavefront algorithm is done, if it does then there is no solution and will print "No solution" and exit.

```
def backtracking(map, start_row, start_col):
    trajectory = []
    if map[start_row][start_col] == 0:
        print("No Solution")
        return trajectory
    .
    .
    .
```

7 Conclusion

Through the course of this project we have gained insight regarding the multiple methods that are used for pathfinding and different approaches to solving this problem like breadth first search and depth first search methods, and learned about the real life applications of such algorithms in the field of robotics.

References

[Com79] Aya M. Zaian1, Abou-Hashema M. El-Sayed (2019). WAVEFRONT ALGORITHM FOR MOTION PLANNING AND OBSTECALES AVOIDING IN STATIC ENVIRONMENT. In *2019 Minia Journal of Engineering & Technology (MJET)*, volume 38.

[AdIn] Issa Zidane and Khalil Ali Khalil Ibrahim (2018). Wavefront and A-Star Algorithms for Mobile Robot Path Planning. In *International Conference on Advanced Intelligent Systems and Informatics*. In *2018 Advances in Intelligent Systems and Computing*, pages 1–6. DOI: $10.1007/978 - 3 - 319 - 64861 - 3_7$.