Name: Jayesh Bhagyesh Gajbhar
Unity ID: jgajbha

Homework 3 : Writeup

I. First Steps

i. Small graph:

The small graph has been modeled on the city of Pune, India (or atleast a part of it, since the city is huge).

The structure of the graph is as follows,
source_name >> source_x_coordinate >> source_y_coordinate >> destination_name >> destination_x_coordinate >> destination_y_coordinate >> distance(by road between the source and destination) >> time_taken(by road according to google maps, during peak traffic hours)

The source and destination coordinates have been based on the real-life coordinates taken from google maps, for example:
Airport (18.579546376032056, 73.90897044170211)

So, the final structure of the graph is like so:
Airport 18.579546376032056 73.90897044170211 Cantt 18.607615207948363 73.87509718082079 7 15

The graph consists of about 23 nodes with 82 edges (considering two way roads connecting the nodes to a from the source and destination, and each edge being a one way road)

The graph was modeled on Pune city as this city is densely packed, i.e., there are multiple roads and connections to each node, and there are a lot of subdivisions(areas) in the city which are at suitable distances from one another, which makes it easy to model a graph on it. There is also a lot of data available on google maps, such as distance by road from one point to another (acting as edge weights), coordinates of a place and the time taken to travel by car from one point to another during peak traffic hours(to use as potential heuristics).

The graphical representation of the graph has been attached to the appendix (I.i).

ii. Large graph:

The large graph has been generated using a python script which is as follows:
```
import networkx as nx
```

```
# Generate a random graph with NetworkX
large_graph = nx.gnm_random_graph(100000, 500000)

# Export the graph to a text file
nx.write_edgelist(large_graph, "graph2.txt", data=False)
```

The NetworkX library in python has been used to generate this graph. The gnm_random_graph(100000, 500000) generates a digraph, with 100000 nodes and 500000 edges. This graph is then exported to a txt file.

Please note that this script generates an unweighted graph. To add weights to the graph, the following python script has been used:

```
import random

# Open the original edge list file for reading
with open("graph.txt", "r") as original_file:
  # Open a new file to write the modified edge list with weights
  with open("graph_with_weights.txt", "w") as new_file:
    # Iterate over each line in the original file
    for line in original_file:
      # Split the line into source and target nodes
      source, target = map(int, line.strip().split())
      # Generate a random whole number between 1 and 20
      weight = random.randint(1, 20)
      # Write the line with the weight included
      new_file.write(f"{source} {target} {weight}\n")
```

The above script generates random weights between 1 and 20 and assigns it to each edge.

To make use of a heuristic which is sensible, random coordinates have been added to each node and to make sure that no two points have the same coordinates in the graph, the following python script has been used:

```
import random

def generate_unique_coordinates(num_nodes):
  # Generate all possible coordinates within the range
  all_coordinates = [(x, y) for x in range(1000) for y in range(1000)]
  # Shuffle the list of coordinates randomly
  random.shuffle(all_coordinates)
  # Yield unique coordinates for the required number of nodes
```

```
    for i in range(num_nodes):
        yield all_coordinates[i]

def add_random_unique_coordinates(input_file, output_file):
    num_nodes = 0
    with open(input_file, 'r') as f_in:
        # Count the number of nodes in the input file
        for _ in f_in:
            num_nodes += 1

    print(num_nodes)

    # Ensure that there are enough unique coordinates for each node
    if num_nodes * 2 > 1000 * 1000:
        print("Error: Number of nodes exceeds available unique coordinates.")
        return

    # Generate unique random coordinates for all nodes
    unique_coordinates = generate_unique_coordinates(num_nodes * 2)

    with open(input_file, 'r') as f_in, open(output_file, 'w') as f_out:
        for line in f_in:
            fields = line.strip().split()
            node1 = fields[0]
            node2 = fields[1]
            distance = fields[2]
            # Get unique random coordinates for each node
            coord1_x, coord1_y = next(unique_coordinates)
            coord2_x, coord2_y = next(unique_coordinates)
            f_out.write(f"{node1} {coord1_x} {coord1_y} {node2} {coord2_x} {coord2_y}
{distance}\n")

input_file = "large_graph.txt"
output_file = "large_graph_with_coordinates.txt"
add_random_unique_coordinates(input_file, output_file)
```

In the subsequent parts of the assignment, the Euclidean distance heuristic has been used.

The final structure of the large_graph.txt file is like so:
0 179 731 41762 715 370 3
which is similar to the structure of the small graph, i.e.,
source_name >> source_x_coordinate >> source_y_coordinate >> destination_name >> destination_x_coordinate >> destination_y_coordinate >> edge_weight

II. Dijkstra's Algorithm and A* Algorithm:

Choice of heuristics:

For the small graph – Euclidean distance
For the large graph – Euclidean distance

A. Dijkstra's algorithm:

i. Small graph:
For the small graph, 23*23 combinations of source and destination have been tested, i.e., all the nodes have been tested, once as the source and once as the destination.

The results were as follows:
Average time taken by Dijkstra: 194.063 microseconds

The fringe is generated like so(example):
Source: Airport, Destination: Railway
Fringe: Airport(0)
Fringe: Viman(1) Tingre(2) Kalyani(3) Cantt(7)
Fringe: Ngr_Rd(2) Tingre(2) Kalyani(3) Cantt(7) Wagholi(9)
Fringe: Tingre(2) Kalyani(3) Kharadi(3) Wdsheri(3) Cantt(7) Wagholi(8) Wagholi(9)
Fringe: Kalyani(3) Kharadi(3) Wdsheri(3) Vishrant(4) Cantt(7) Wagholi(8) Wagholi(9)
Fringe: Kharadi(3) Wdsheri(3) Vishrant(4) Yerwada(4) KP(6) Cantt(7) Bund(8) Wagholi(8) Wagholi(9)
Fringe: Wdsheri(3) Vishrant(4) Yerwada(4) Wagholi(5) KP(6) Mundhwa(6) Cantt(7) Bund(8) Wagholi(8) Wagholi(9)
Fringe: Vishrant(4) Yerwada(4) Mundhwa(5) Wagholi(5) KP(6) Mundhwa(6) Cantt(7) Bund(8) Wagholi(8) Wagholi(9)
Fringe: Yerwada(4) Mundhwa(5) Wagholi(5) KP(6) Mundhwa(6) Sangam(6) Cantt(7) Bund(8) Wagholi(8) Wagholi(9)
Fringe: Mundhwa(5) Wagholi(5) KP(6) Mundhwa(6) Sangam(6) Cantt(7) Bund(8) Wagholi(8) Wagholi(9)
Fringe: Wagholi(5) KP(6) Mundhwa(6) Sangam(6) Cantt(7) Bund(8) Magar(8) Wagholi(8) Wagholi(9)
Fringe: KP(6) Mundhwa(6) Sangam(6) Cantt(7) Bund(8) Magar(8) Wagholi(8) Wagholi(9)
Fringe: Mundhwa(6) Sangam(6) Cantt(7) Bund(8) Magar(8) Wagholi(8) Camp(9) Wagholi(9)
Fringe: Sangam(6) Cantt(7) Bund(8) Magar(8) Wagholi(8) Camp(9) Wagholi(9)
Fringe: Cantt(7) Bund(8) Magar(8) Railway(8) Wagholi(8) Camp(9) Wagholi(9) Deccan(11) Old_Pune(11)

```
Fringe: Bund(8) Magar(8) Railway(8) Wagholi(8) Camp(9) Wagholi(9) Deccan(11)
Old_Pune(11)
Fringe: Magar(8) Railway(8) Wagholi(8) Camp(9) Wagholi(9) Deccan(11)
Old_Pune(11)
Fringe: Railway(8) Wagholi(8) Camp(9) Wagholi(9) Deccan(11) Old_Pune(11)
Shortest path found using Dijkstra's algorithm: Airport Tingre Vishrant Sangam
Railway
Dijkstra's algorithm took 290 microseconds.
```

It is evident that the fringe in dijkstra's algorithm only takes into account the edge weights to decide which node it will fill next. Which means that the fringe in Dijkstra's algorithm expands uniformly in all directions, considering all neighboring nodes without any prioritization based on the estimated distance to the destination. As a result, the fringe tends to grow uniformly outward from the source node.

Since Dijkstra's algorithm explores all reachable nodes uniformly, it tends to fill the entire reachable space from the source node. This can result in a larger fill, especially in graphs with many nodes and edges, as it explores all possible paths.

Please note that the machine used for testing these algorithms is a Macbook Air with an M2 chip and 16GB RAM, the run times may vary based on the specifications of the machine.

ii. Large graph:
Dijkstra's algorithm takes significantly longer to execute on the large graph, which is as expected since the graph has 100000 nodes and 500000 edges.

The results were as follows:

```
Dijkstras on large graph
Shortest path found using Dijkstra's algorithm: 100 29818 36159 27953 4416
33696 33300 89778 200
Time taken by dijkstra's algorithm: 178344 microseconds
Cost: 25
```

Please note that the above result is considering just one iteration of the dijkstras algorithm on the large graph, but all the other variations, i.e., dijkstras algorithm on different source and destination nodes takes about the same amount of time, i.e., 150000 – 200000 microseconds.

Similar to the dijkstras on the small graph, the fringe only accounts for the edge weight and grows uniformly outward from the source node.

As stated above, the fill is much larger in this case as the graph is much larger.

The fringe has not been printed out in this case as it would significantly add to the run time of the program and therefore, would be much slower than it already is.

B. A* algorithm:

i. Small graph:
For the small graph, 23*23 combinations of source and destination have been tested, i.e., all the nodes have been tested, once as the source and once as the destination.

The results were as follows:
Average time taken by A*: 373.051 microseconds.

The A* algorithm takes significantly longer to execute, which is to be expected as the algorithm performs a map lookup to find the coordinates to calculate the Euclidean distance between two points.

In this case, the fringe is generated like so:

```
Source: Airport, Destination: Railway
Fringe: Airport(0)
Fringe: Viman(1.06714) Tingre(2.06429) Kalyani(3.0442) Cantt(7.08623)
Fringe: Tingre(2.06429) Ngr_Rd(2.06634) Kalyani(3.0442) Cantt(7.08623)
Wagholi(9.12708)
Fringe: Ngr_Rd(2.06634) Kalyani(3.0442) Vishrant(4.05227) Cantt(7.08623)
Wagholi(9.12708)
Fringe: Kalyani(3.0442) Wdsheri(3.06261) Kharadi(3.08971) Vishrant(4.05227)
Cantt(7.08623) Wagholi(8.12708) Wagholi(9.12708)
Fringe: Wdsheri(3.06261) Kharadi(3.08971) Yerwada(4.0442) Vishrant(4.05227)
KP(6.03856) Cantt(7.08623) Bund(8.02609) Wagholi(8.12708) Wagholi(9.12708)
Fringe: Kharadi(3.08971) Yerwada(4.0442) Vishrant(4.05227) Mundhwa(5.04816)
KP(6.03856) Cantt(7.08623) Bund(8.02609) Wagholi(8.12708) Wagholi(9.12708)
Fringe: Yerwada(4.0442) Vishrant(4.05227) Mundhwa(5.04816)
Wagholi(5.12708) KP(6.03856) Cantt(7.08623) Bund(8.02609) Wagholi(8.12708)
Wagholi(9.12708)
Fringe: Vishrant(4.05227) Mundhwa(5.04816) Wagholi(5.12708)
Sangam(6.02053) KP(6.03856) Cantt(7.08623) Bund(8.02609) Wagholi(8.12708)
Wagholi(9.12708)
Fringe: Mundhwa(5.04816) Wagholi(5.12708) Sangam(6.02053) KP(6.03856)
Cantt(7.08623) Bund(8.02609) Wagholi(8.12708) Wagholi(9.12708)
Fringe: Wagholi(5.12708) Sangam(6.02053) KP(6.03856) Cantt(7.08623)
Bund(8.02609) Magar(8.06088) Wagholi(8.12708) Wagholi(9.12708)
```

```
Fringe: Sangam(6.02053) KP(6.03856) Cantt(7.08623) Bund(8.02609)
Magar(8.06088) Wagholi(8.12708) Wagholi(9.12708)
Fringe: KP(6.03856) Cantt(7.08623) Railway(8) Bund(8.02609) Magar(8.06088)
Wagholi(8.12708) Wagholi(9.12708) Deccan(11.0246) Old_Pune(11.0339)
Fringe: Cantt(7.08623) Railway(8) Bund(8.02609) Magar(8.06088)
Wagholi(8.12708) Camp(9.02205) Wagholi(9.12708) Deccan(11.0246)
Old_Pune(11.0339)
Fringe: Railway(8) Bund(8.02609) Magar(8.06088) Wagholi(8.12708)
Camp(9.02205) Wagholi(9.12708) Deccan(11.0246) Old_Pune(11.0339)
Path found: Airport Kalyani Yerwada Sangam Railway
A* algorithm took 525 microseconds.
```

Evidently, the A* algorithm considers the heuristic as well as the edge weight while generating the fringe. In the A* algorithm, the fringe expands based on a heuristic function that estimates the cost from the current node to the destination. This allows the algorithm to prioritize nodes that are more likely to lead to the goal. As a result, the fringe tends to focus on areas closer to the destination, leading to a more directed search.

The fill in A* algorithm depends on the heuristic function. If the heuristic is consistent and accurately estimates the cost to reach the destination, the fill may be smaller compared to Dijkstra's algorithm. This is because A* focuses its exploration on nodes that are more likely to lead to the goal, potentially ignoring areas that are unlikely to contain the optimal path. In this case, the fill may be smaller as the heuristic here is admissible and consistent.

ii. A* algorithm:
A* takes significantly longer than Dijkstra's algorithm on the large graph. This is because of the lookups it performs while calculating the heuristic value for each entry of the fringe.

The results were as follows:

```
A* on large graph
Shortest path found using A* algorithm: 100 18508 61647 32852 5369 1743
19024 48563 67055 98439 1247 74133 5154 51056 28243 89778 200
Time taken by dijkstra's algorithm: 49329223 microseconds
Cost: 105
```

The A* algorithm, in this case, has generated a longer path from the source to the destination node because the heuristic in this case is overestimating and inadmissible, i.e., the coordinates assigned to the nodes generate Euclidean distances that are orders of magnitude greater than the actual distance between the source and destination nodes.

(Note how the coordinates are in the 10^2 order of magnitude and the distance between the nodes is in the 10^1 order or magnitude in the following edge example:
0 179 731 41762 715 370 3)

In this case, even though the fringe is smaller than dijkstras algorithm for the same graph considering the same source and destination nodes, the fill is much larger, because the heuristic is overestimating in nature.

Examples and logs are available in the Part2 folder under the names part2_examples.txt and part2_logs.txt respectively.

III. Heuristics

For this part of the assignment, I have used 4 heuristics, which are as follows:

i. **Euclidean distance:**
Based on the real life coordinates (taken from google maps) of the locations, Euclidean distance was calculated.
This heuristic is underestimating, admissible and consistent in this case as Euclidean distance calculated based on the coordinates gives a value much lower than the actual distance(in km, by road).

ii. **Time taken, by road during peak traffic hours:**
Based on the real life travel time(in minutes) between these nodes(taken from google maps).
This heuristic is overestimating in nature and inadmissible.

iii. **Euclidean distance squared:**
Euclidean distance from i. but squared.
This is an underestimating heuristic, contrary to intuition, as the value given by Euclidean distance in this case is in the order of ~10^{-2}, and squaring that value yields an even lower value.

iv. **(1000*(Manhattan distance))^4:**
Based on Manhattan distance between the source and destination nodes, but multiplied by a factor of 1000 and then raised to the power of 4.
This is an extremely overestimating heuristic as it yields a value far greater than the actual distance between two nodes(by road).

Results of running the A* algorithm with all of the above heuristics are listed below:

Example 1:

Source: Yerwada, Destination: Wagholi

A* with Euclidean distance: (underestimating)
Path found: Yerwada Kalyani Ngr_Rd Kharadi Wagholi
Cost : 5
A* algorithm took 125 microseconds.

A* with time: (overestimating, since timetaken + actual distance > total distance required)
Path found: Yerwada Kalyani Wdsheri Kharadi Wagholi
Cost : 5
A* algorithm with time heuristic took 79 microseconds.
(Takes lesser time to execute compared to the Euclidean distance heuristic, as there is no computation needed to be performed, data is provided in the graph structure itself)

A* with Euclidean distance squared: (underestimating, because the distance calculated by running euclidean distance is in 10^-2 order of magnitude and squaring this would make it even smaller)
Path found: Yerwada Kalyani Ngr_Rd Kharadi Wagholi
Cost : 5
A* algorithm took 126 microseconds.
(Takes a similar amount of time to execute compared to the Euclidean distance heuristic)

A* with (1000*(manhattan))^4: (overestimating, since (1000*(manhattan))^4 is much larger than the actual distance)
Path found: Yerwada Vishrant Tingre Viman Wagholi
Cost : 15
A* algorithm took 62 microseconds.
(Runs faster than all of the above due to the following reasons:

a. It is computationally less expensive to calculate Manhattan distance compared to Euclidean distance
b. Since the heuristic overestimates the distance, the algorithm may find a suboptimal path that is longer than the optimal path. Longer paths typically require exploring fewer nodes compared to shorter paths, leading to faster execution.)

The output paths in each case reflect how the heuristic influences the search algorithm's decision-making process. Heuristics that underestimate or overestimate the actual distance can lead to suboptimal or inefficient paths. In the case of an overestimating heuristic like the one using $(1000*(\text{Manhattan}))^4$, the algorithm may choose longer paths to avoid overestimating the actual distance.

Conversely, underestimating heuristics like the Euclidean distance tend to produce shorter but potentially suboptimal paths. The optimal heuristic strikes a balance between underestimating and overestimating the actual distance to guide the algorithm efficiently toward the goal.

It is worth mentioning that the time heuristic may not produce accurate results as the time taken is bound to the edges in the graph structure, and there is no data structure that stores time taken from each node to each other node, leading to potential inaccuracies. Hence, the inclusion of (1000*(Manhattan distance))^4 as an inadmissible heuristic.

Example 2:

Source: Wdsheri, Destination: Kothrud

```
A* with Euclidean distance:
Path found: Wdsheri Kalyani Yerwada Sangam Old_Pune Kothrud
Cost : 12
A* algorithm took 189 microseconds.

A* with time:
Path found: Wdsheri Kalyani Yerwada Sangam Deccan Kothrud
Cost : 12
A* algorithm with time heuristic took 106 microseconds.

A* with Euclidean distance squared:
Path found: Wdsheri Kalyani Yerwada Sangam Old_Pune Kothrud
Cost : 12
A* algorithm took 193 microseconds.

A* with manhattan^4:
Path found: Wdsheri Mundhwa KP Camp Railway Deccan Kothrud
Cost : 16
A* algorithm took 87 microseconds.
```

In summary, the choice of heuristic significantly affects the paths found by the A* algorithm. Underestimating heuristics tend to produce shorter but potentially suboptimal paths, while overestimating heuristics may lead to longer but potentially more optimal paths. The optimal heuristic strikes a balance between underestimating and overestimating the actual distance to guide the algorithm efficiently toward the goal.

IV. Putting it All Together

To create an indoor environment, I created a grid system of 50 x 50 with an individual grid size of 15 x 15, to achieve a window size of 750 x 750.
Next, I created a vector of vectors to implement the grid system in code.
std::vector<std::vector<int>> grid(GridSizeX, std::vector<int>(GridSizeY, 0)); // creating a 2D grid with GridSizeX rows and GridSizeY columns

In this system, a grid cell with the value of 1 is considered to be invalid, i.e., an obstacle, and will not be considered when the A* algorithm is executed. For example, grid[0][0] = 1 means that the cell at (0,0) is inaccessible, i.e., it is an obstacle.

The path is generated by the A* algorithm as a vector of waypoints, i.e., coordinates on the grid system.

For the path following algorithm, I have used a seek behavior for movement and orientation matching for aligning the boid in the direction of the movement, defined in the SteeringBehavior.hpp file.

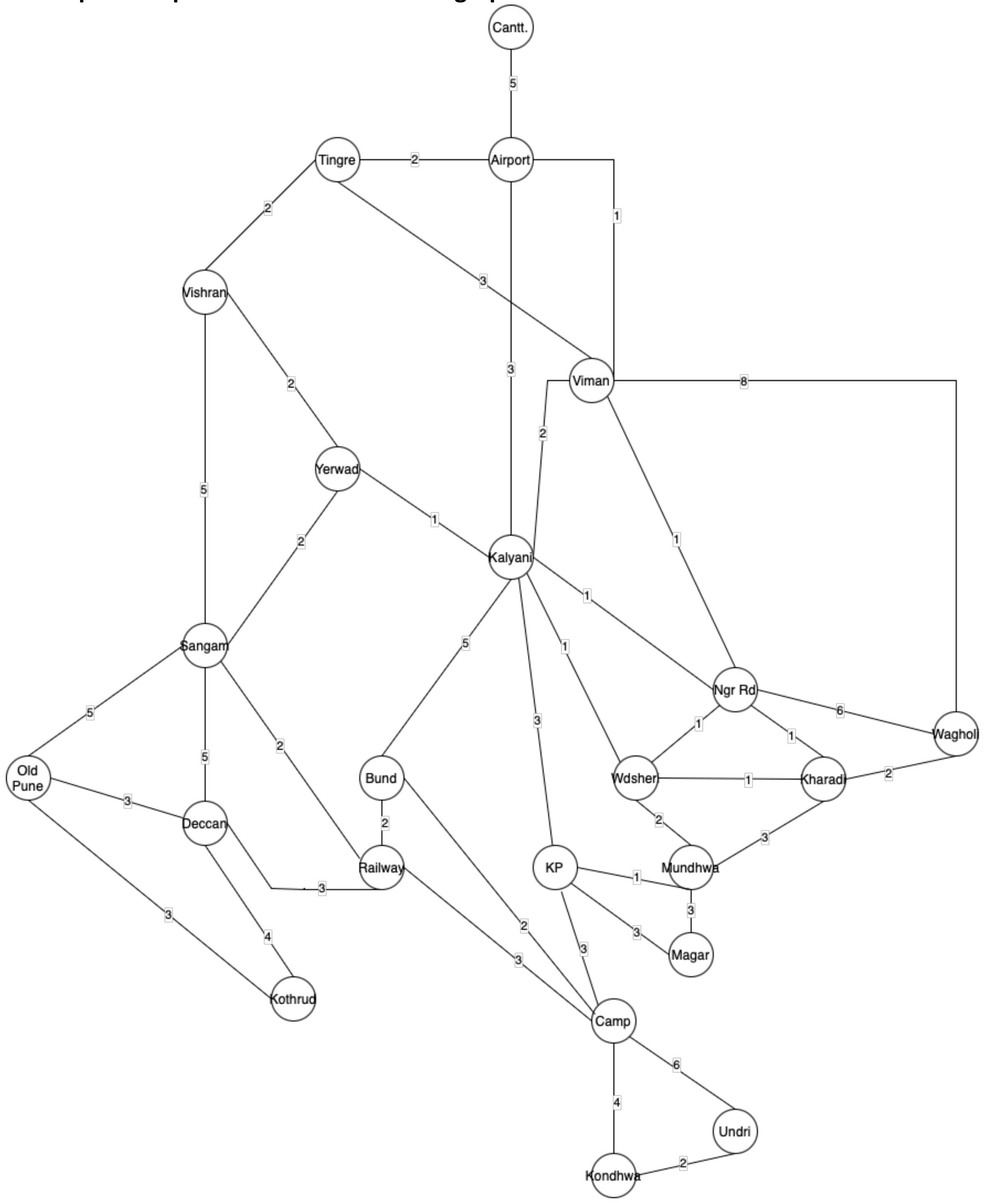
A left click of the mouse sets the current position of the boid start position and a right click sets the mouse position(where the click was made) as the end point and calls the A* algorithm.

The pathfinding function won't be called until a right click is performed.

Finally, screenshots of the implementation have been attached to the appendix (IV).

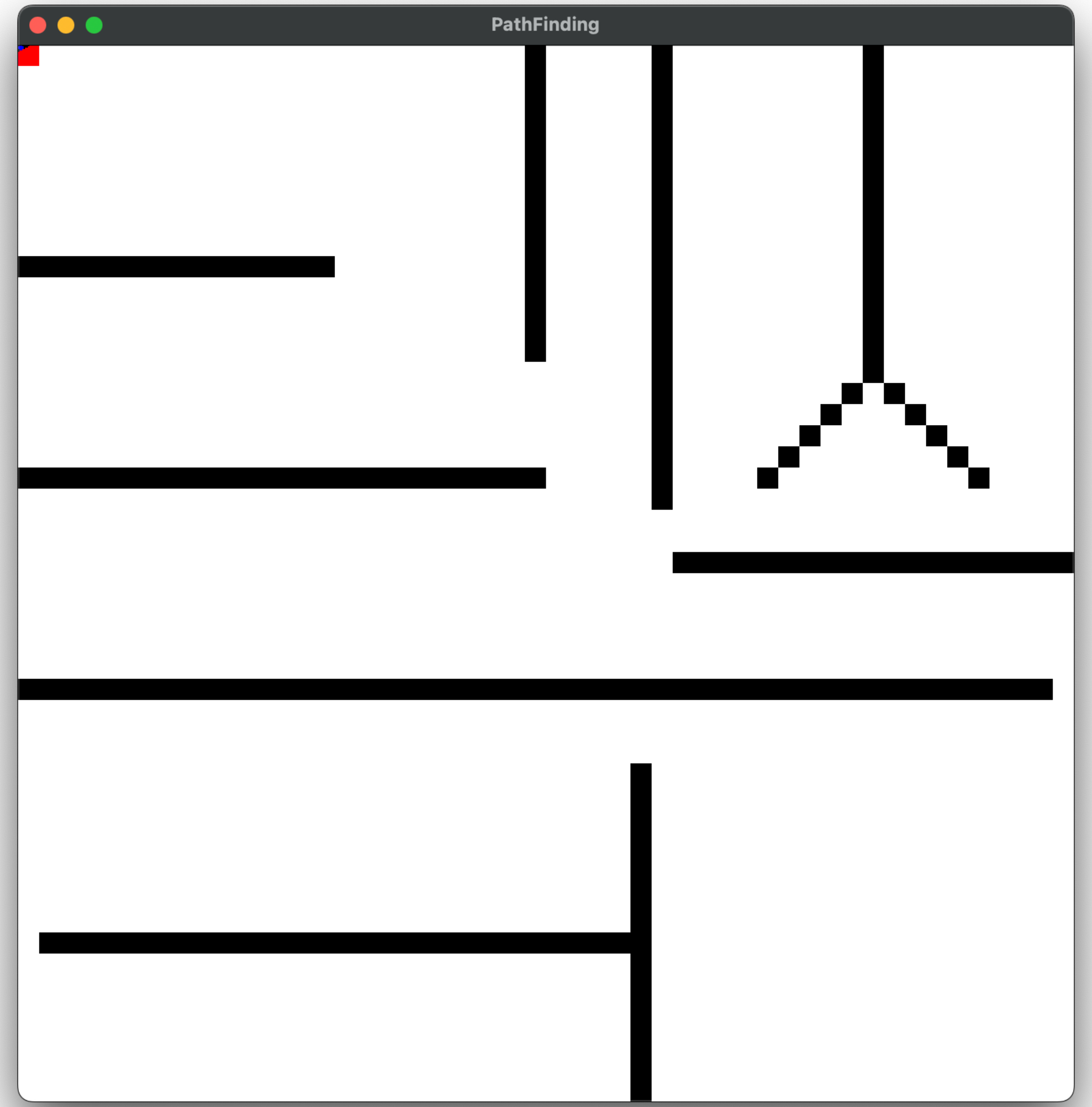
APPENDIX

I.i. Graphical representation of the small graph

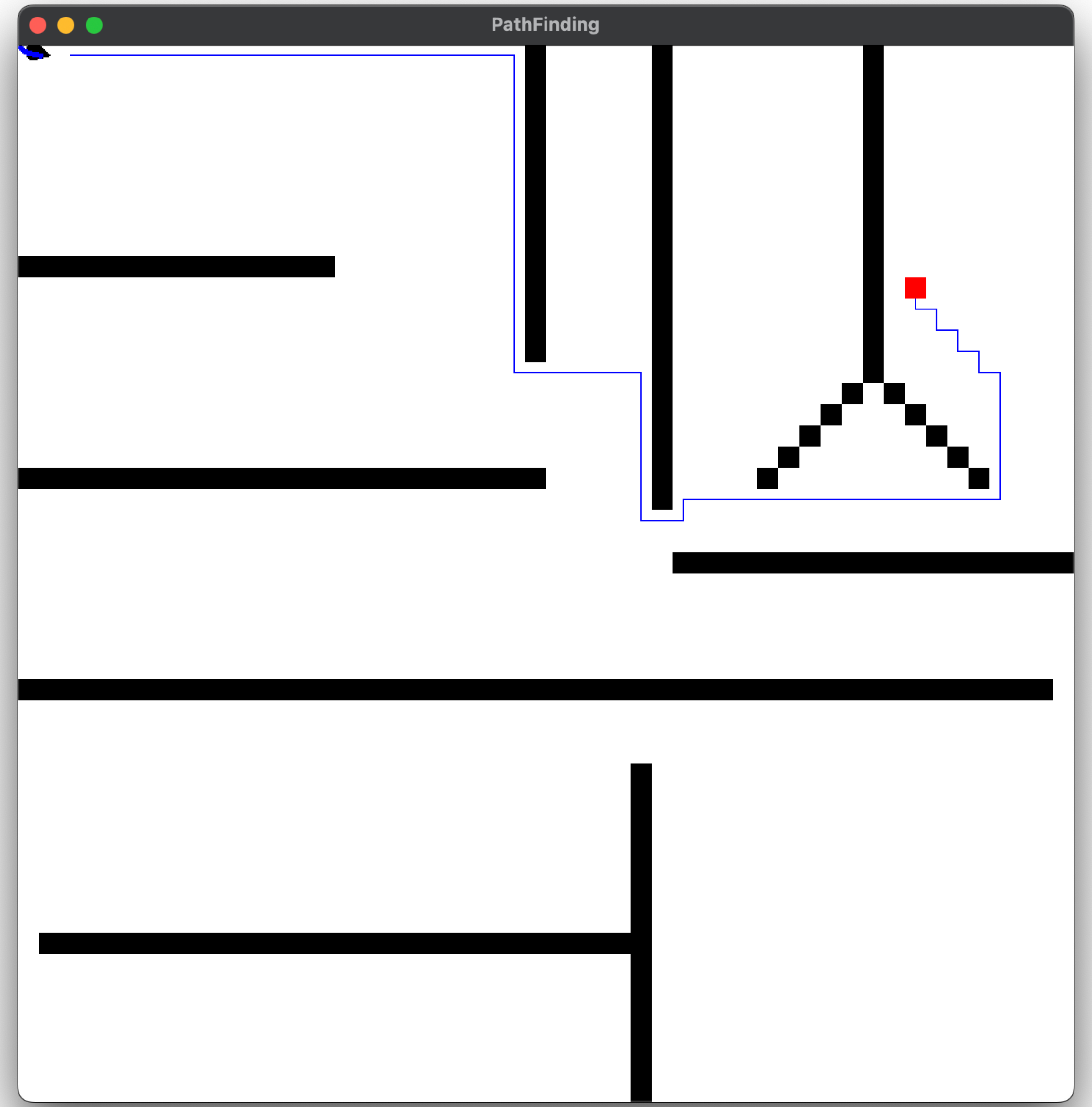


IV.

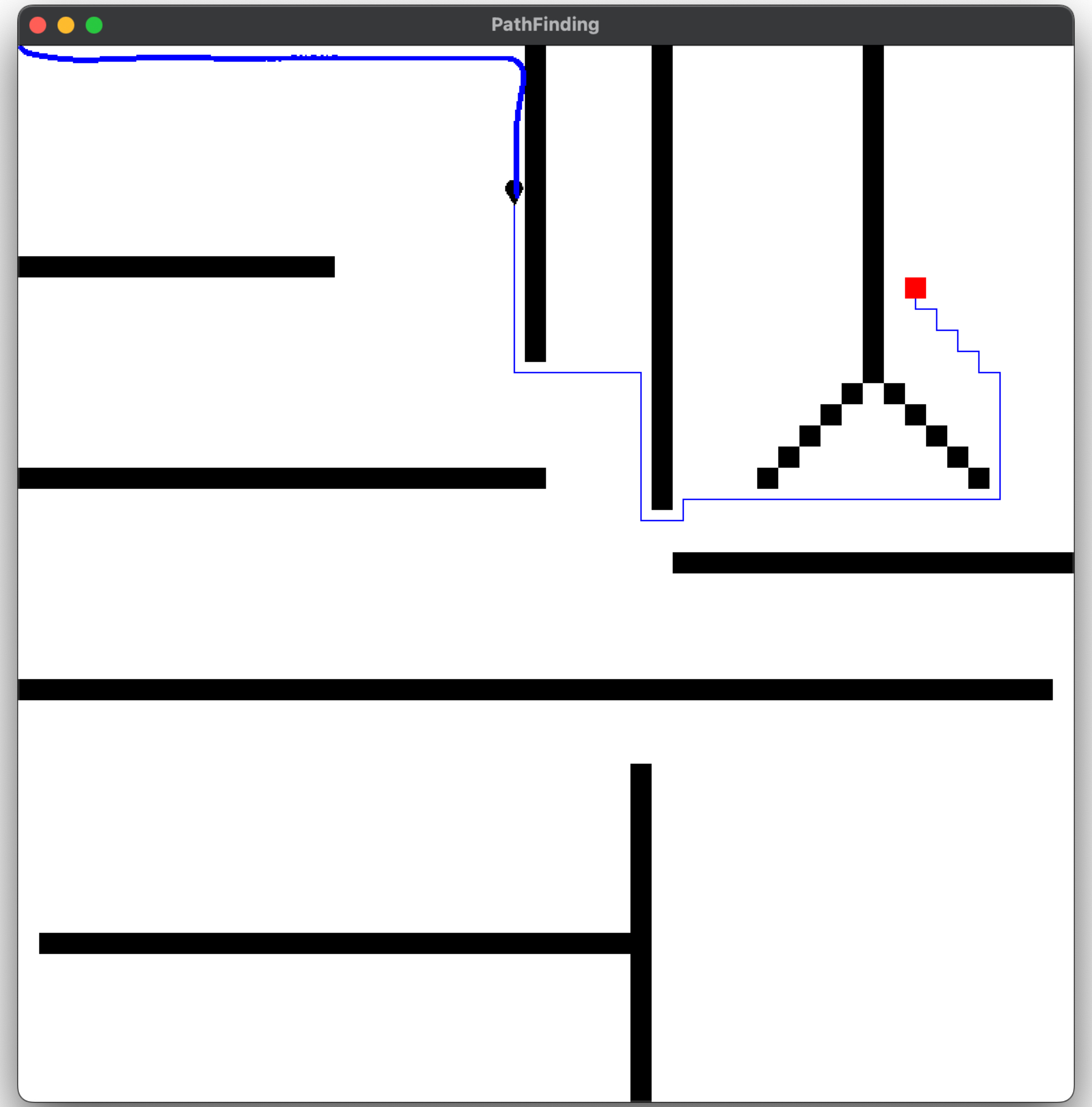
i. Map of the environment

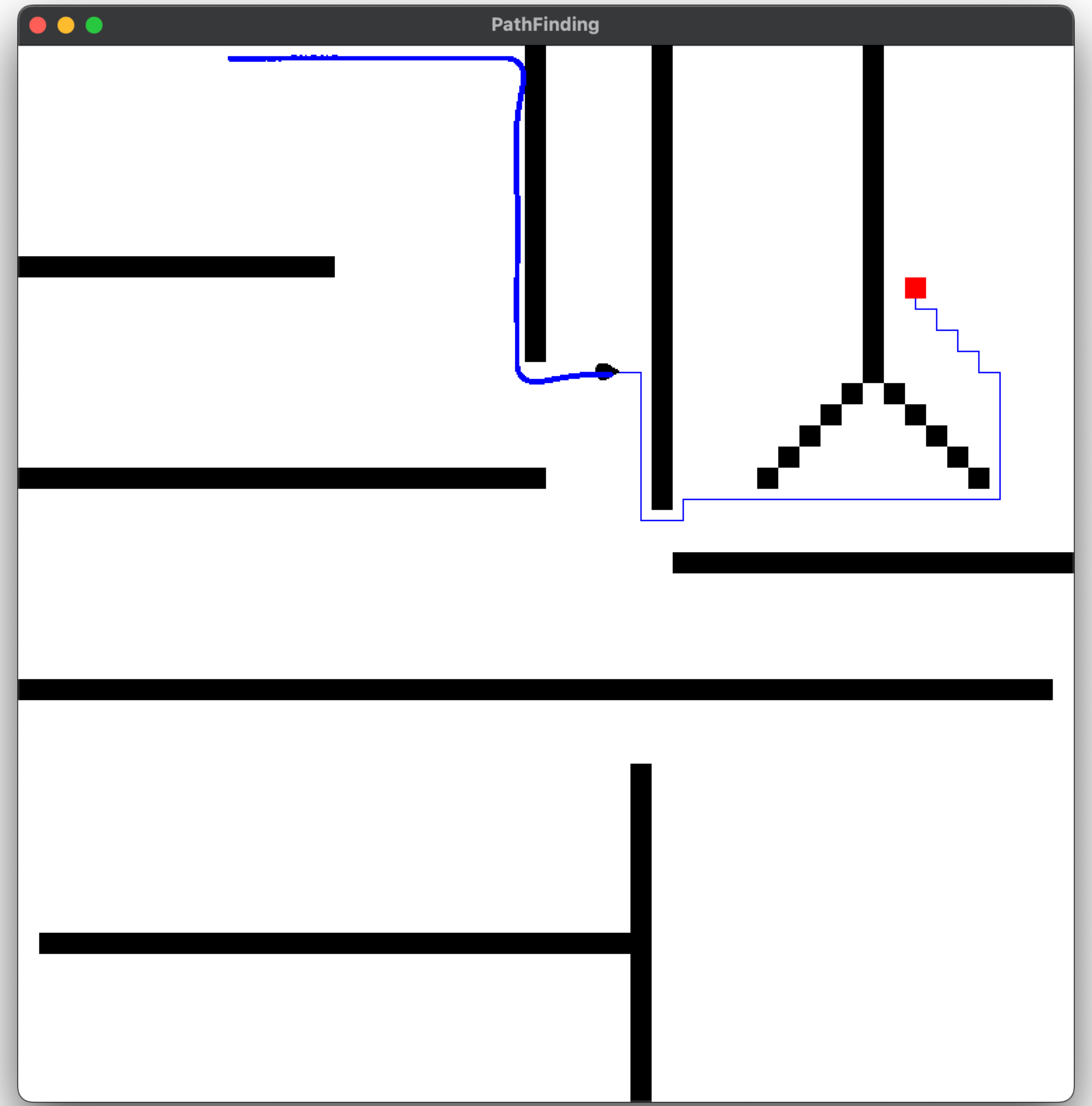


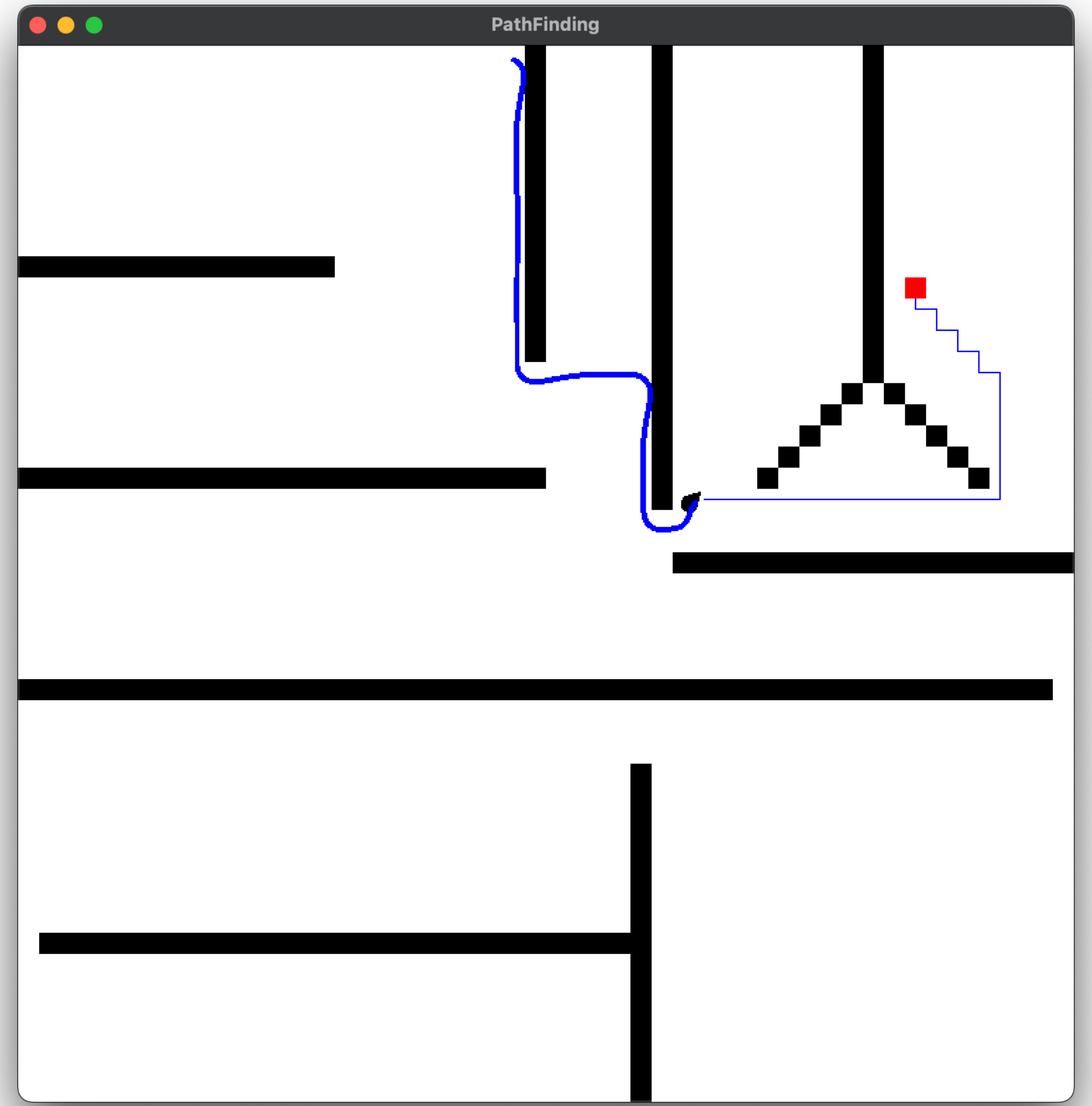
ii. Path generated

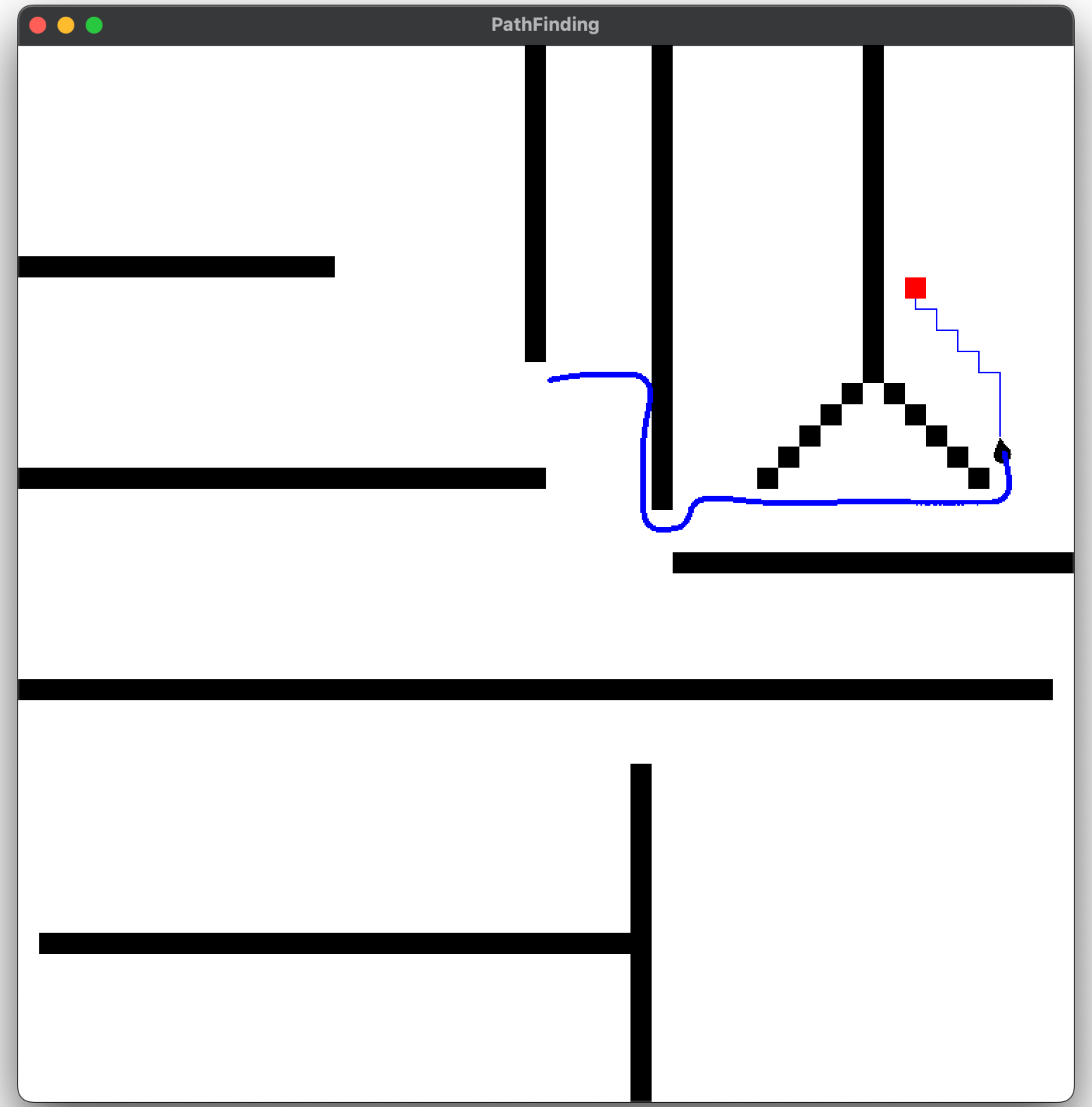


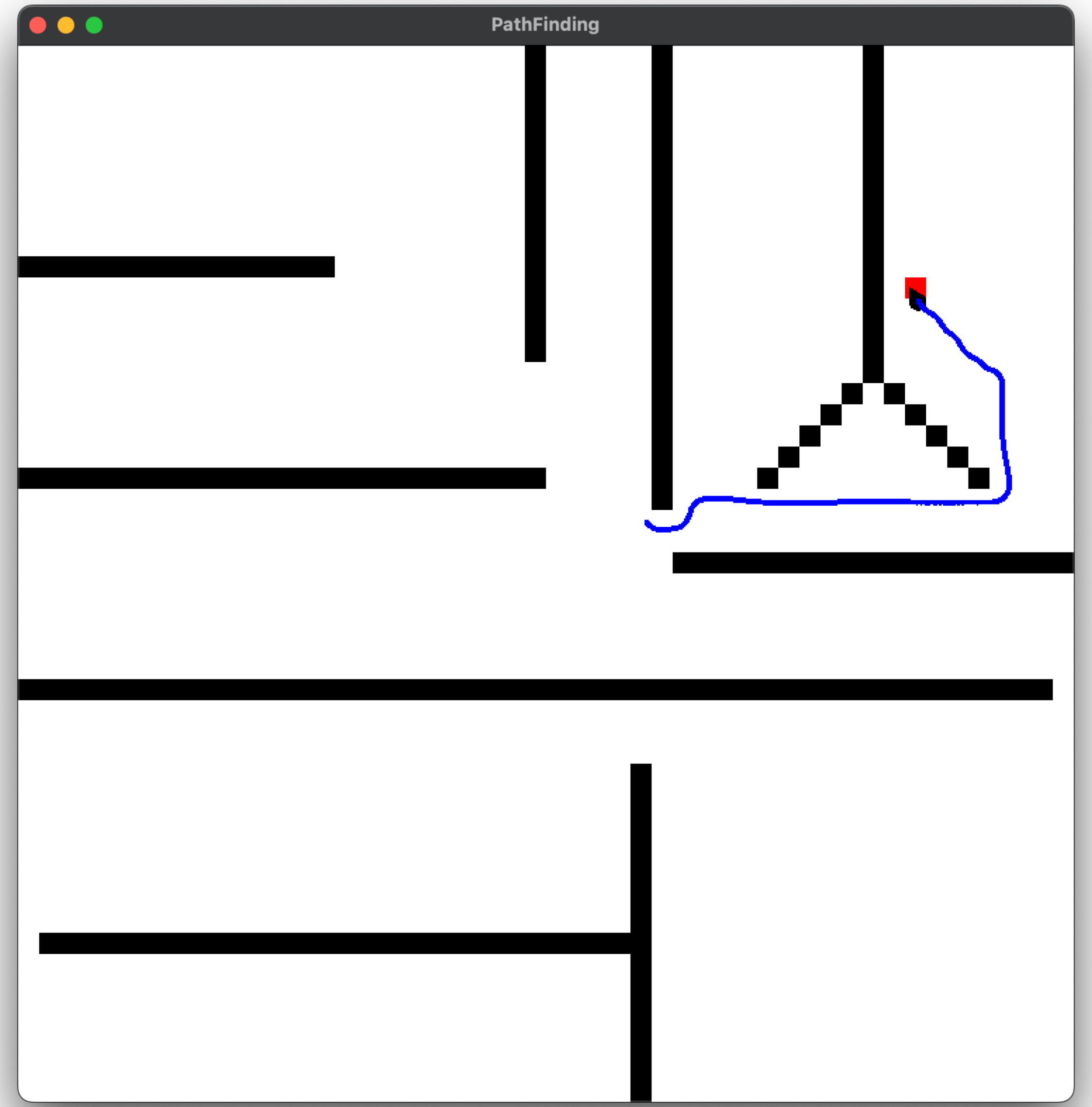
iii. Movement




PathFinding


PathFinding


PathFinding


PathFinding