



Universitat de Girona



Autonomous Systems

A star algorithm

Delivered by:

Muhammad Faran Akram
Raúl López Musito

Supervisor:

El Masri El Chaarani, Alaaeddine

Date of Submission:

21/11/2024

Table of Contents

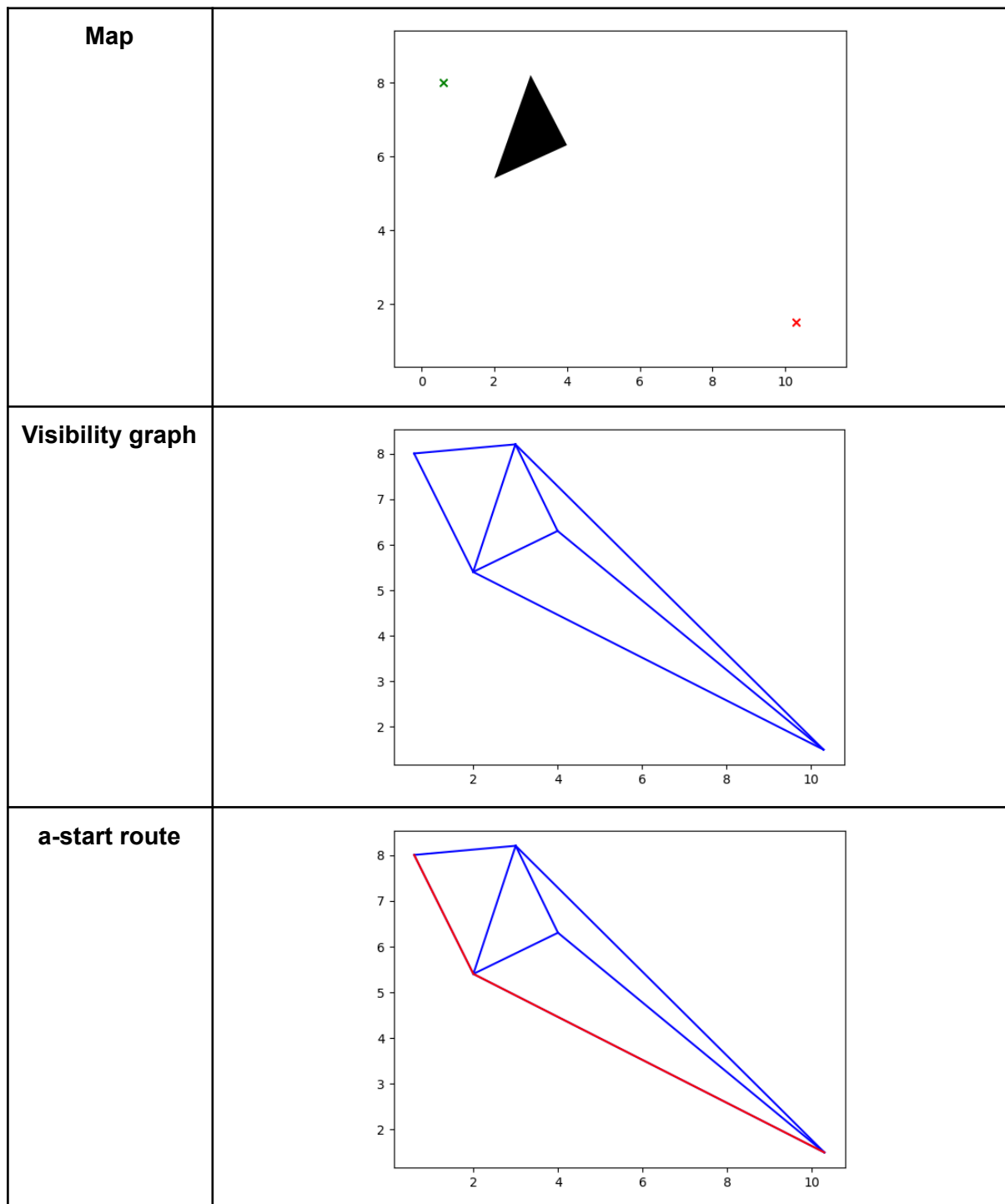
1. Final Outcomes:	3
1.1. Results for the Graph continues environments:	3
1.2. Grid environment:	8
2. Implementations:	11
2.1. Node Class:	11
2.2. Graph environment:	12
2.3. Grid Environment:	14
3. Issues and areas of improvement:	15
3.1. Same Cost but Different Path:	15
3.2. Not optimal path in given environments:	15
4. Conclusion:	16
5. How to execute the code	16

1. Final Outcomes:

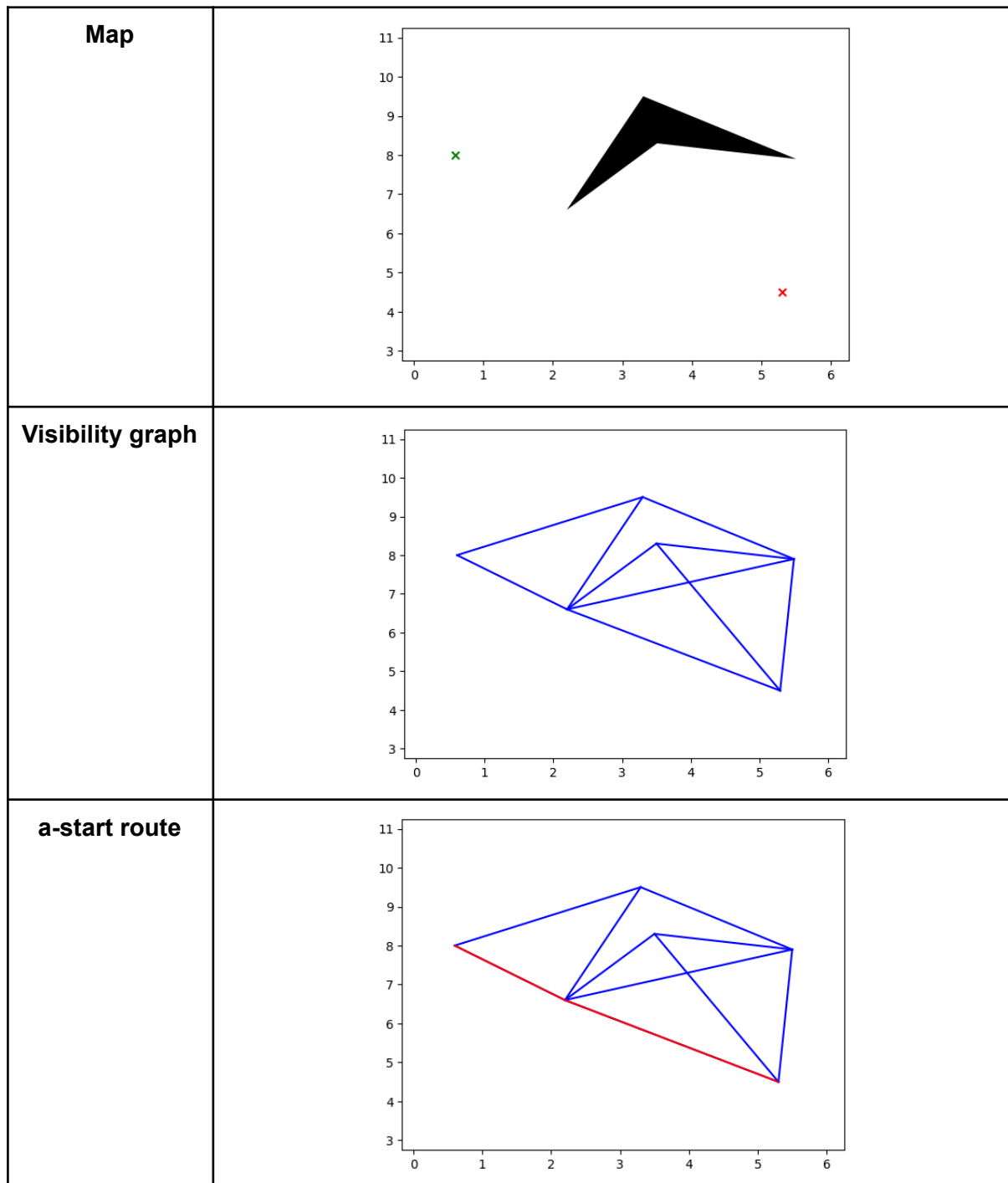
1.1. Results for the Graph continues environments:

For each map provided the computed, then the visibility graph was created and finally with this graph the a star algorithm was used to find the fastest path from the start node to the goal.

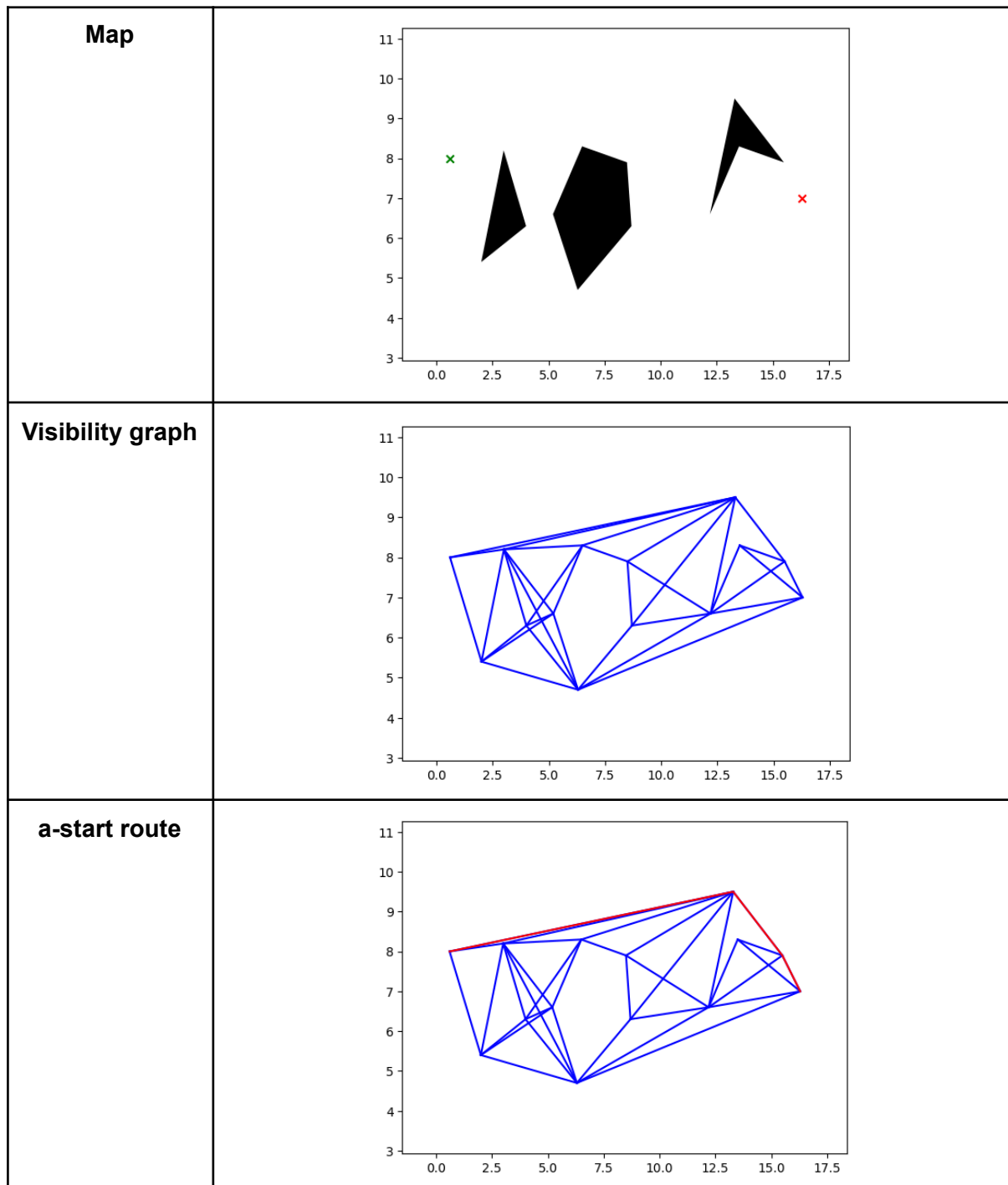
1.1.1. Map 0



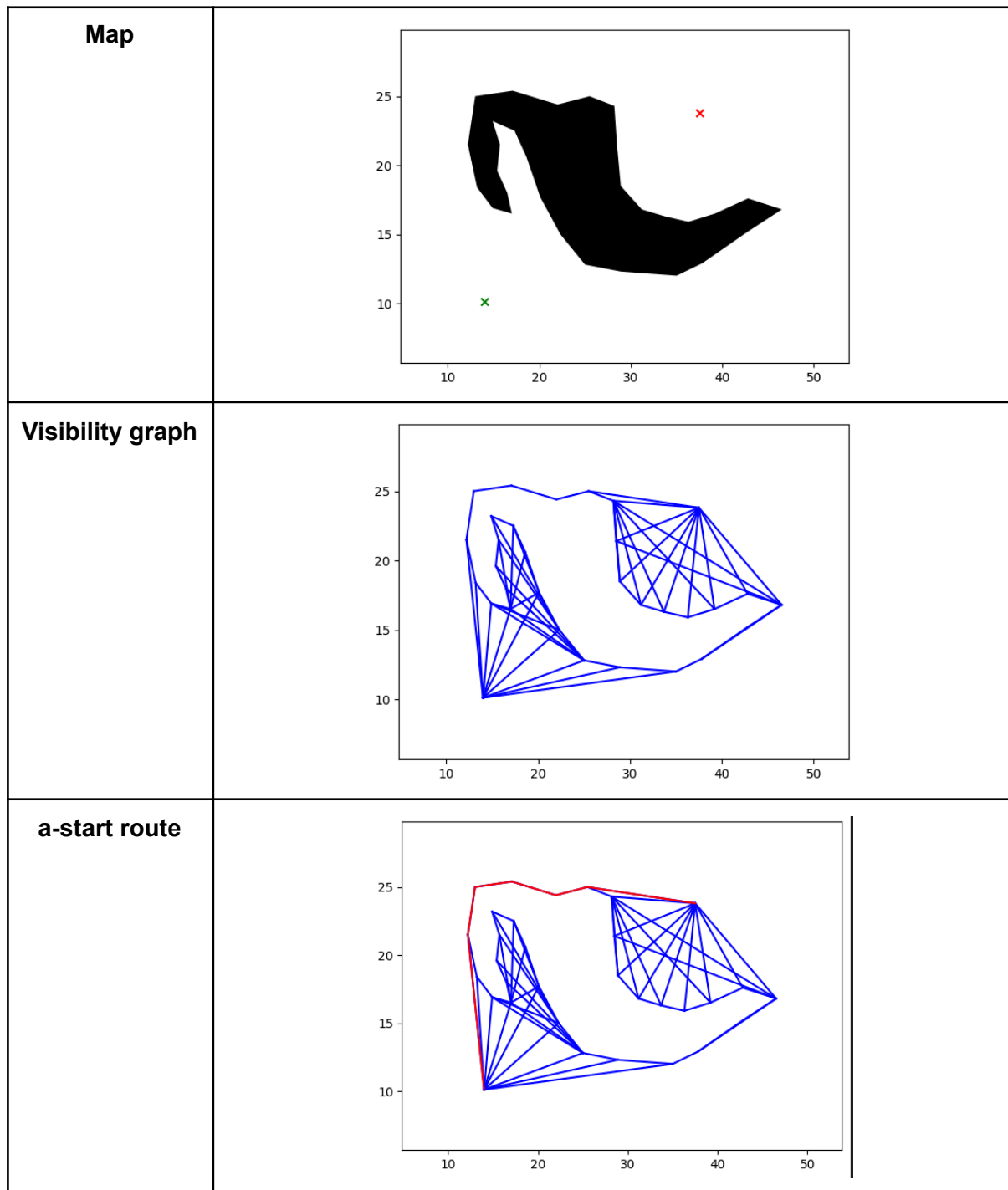
1.1.2. Map 1



1.1.3. Map 2

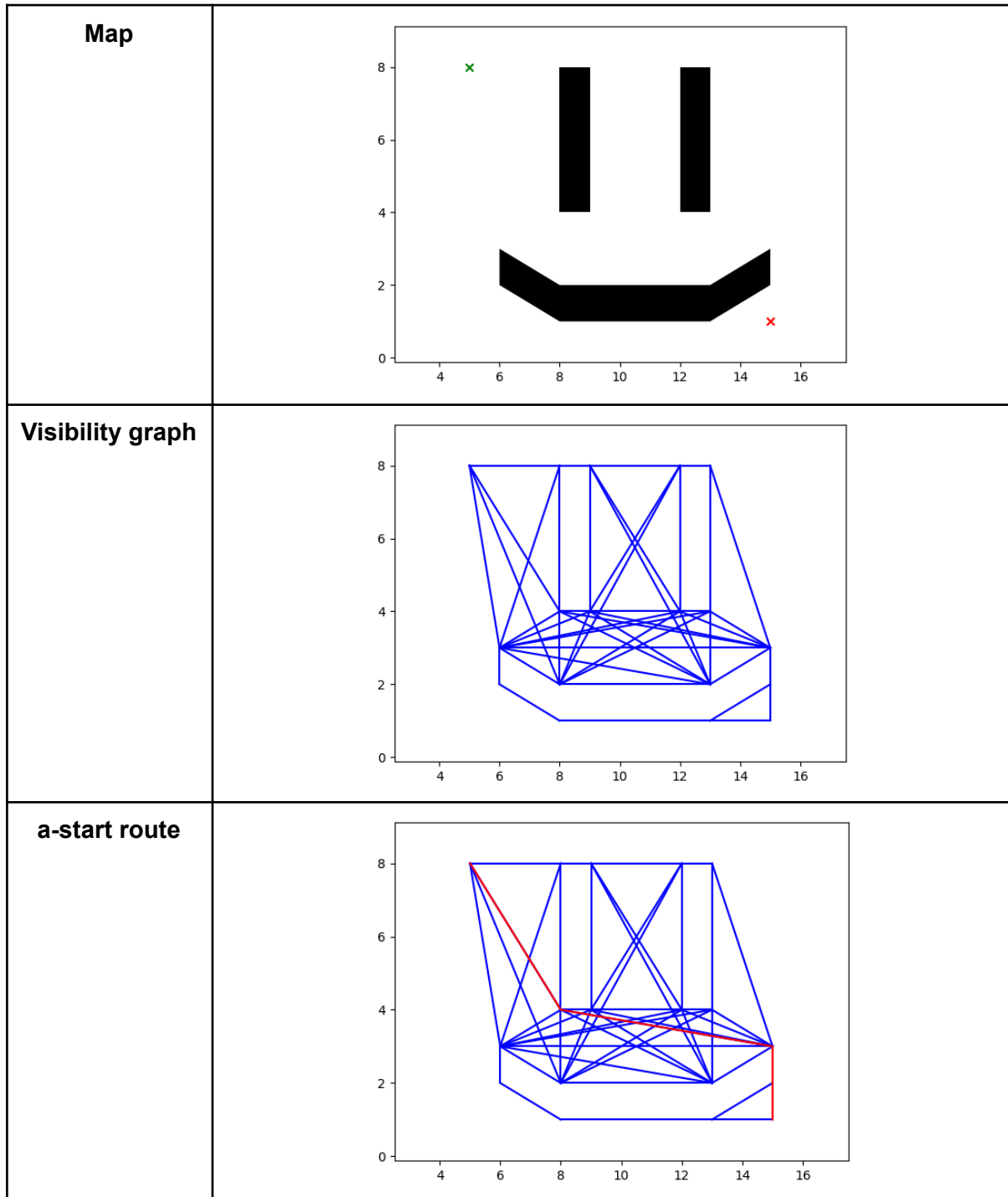


1.1.4. Map mx



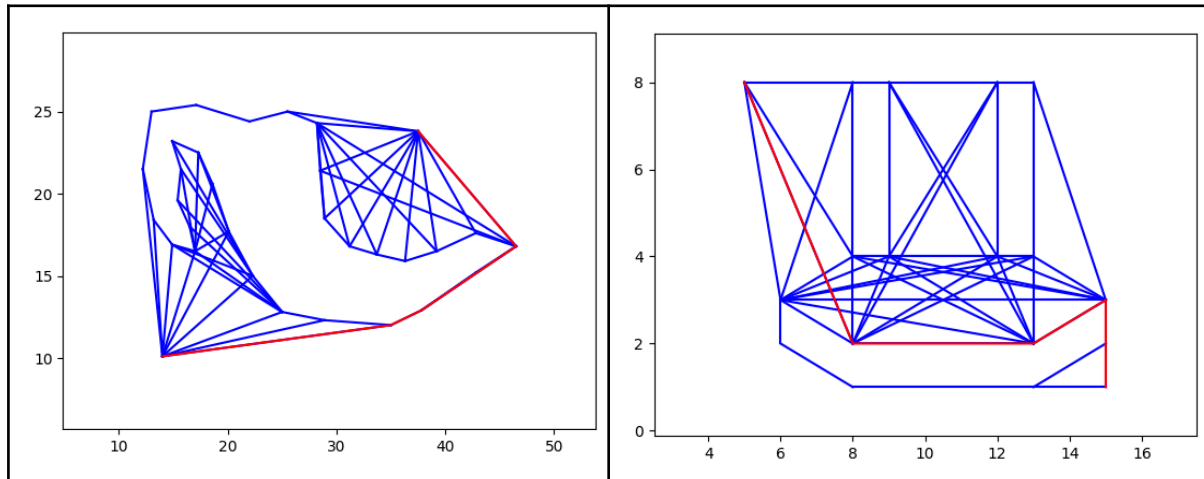
1.1.5. Map :)

An extra environment was implemented just for fun



The optimal path was quickly found in all the tested environments, which were binarized images with one or more polygons, along with a start and goal point. The cost of moving from one node to the next was determined using the Euclidean distance between nodes. Even with different levels of complexity in the environments, the algorithm worked well and consistently found the shortest path from the start to the goal.

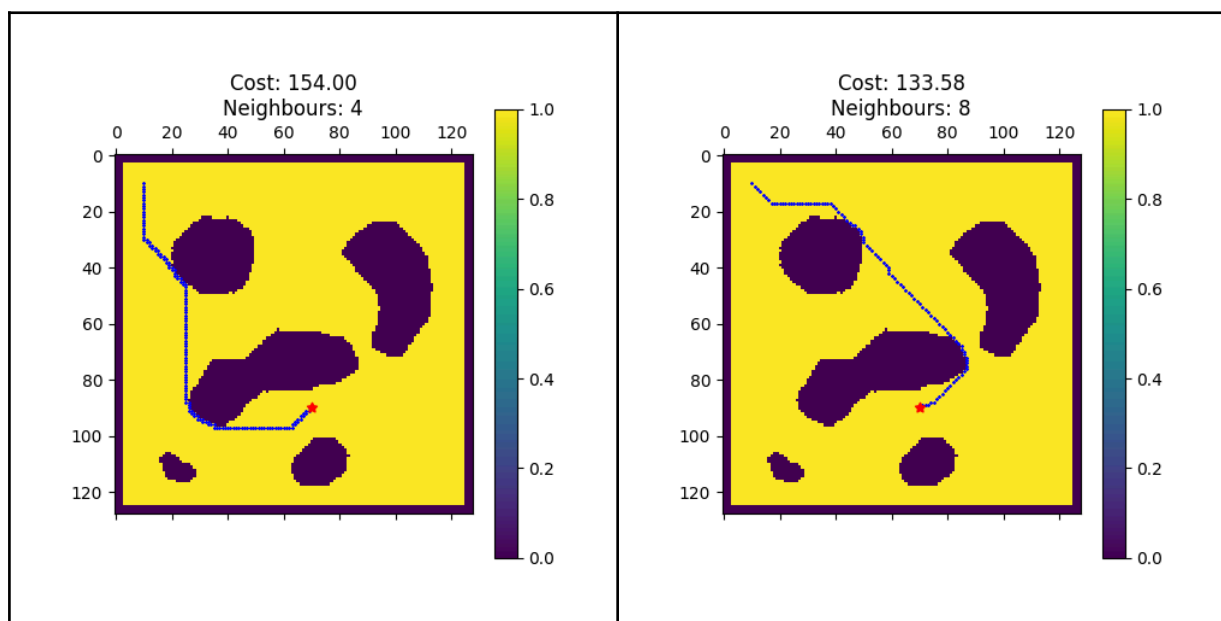
Additionally, to test the behavior of the algorithm with different g costs (the cost of reaching a node from the start), an alternative cost model was implemented. In this model, the cost of moving from one node to another was set to a constant value of 1, ignoring the actual distance between nodes. For the first three maps, the results were identical to the original cost model, so they are not included here. However, in the “mx” and “:;)” maps, slight variations were observed, leading to paths that were no longer optimal, as shown in the following table.

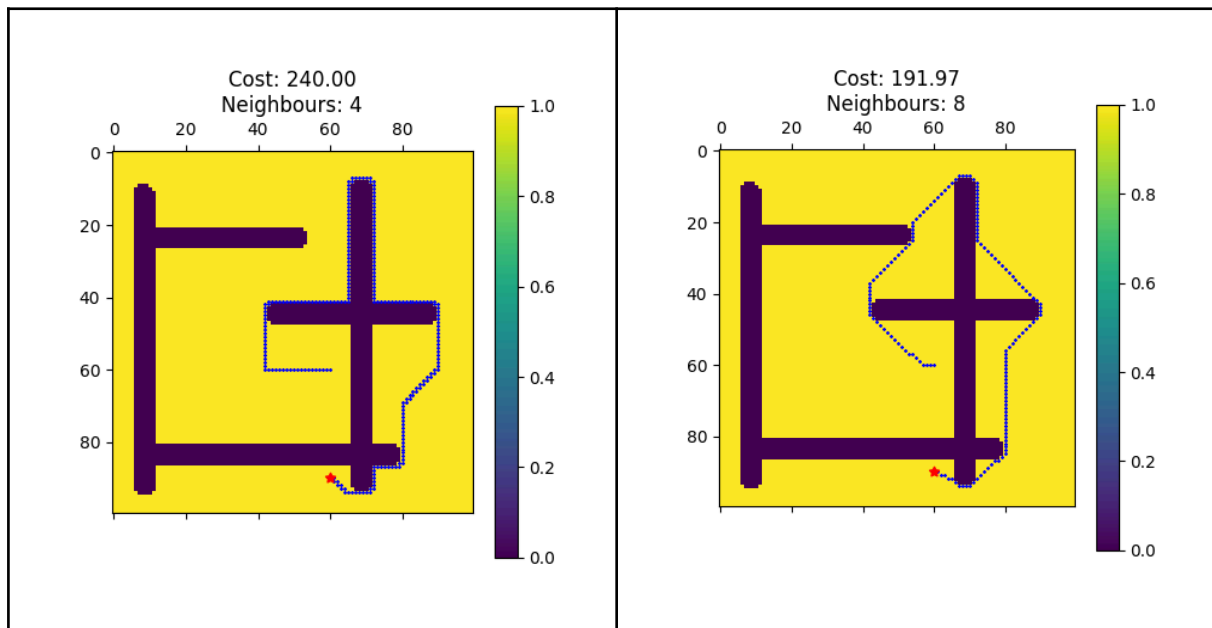
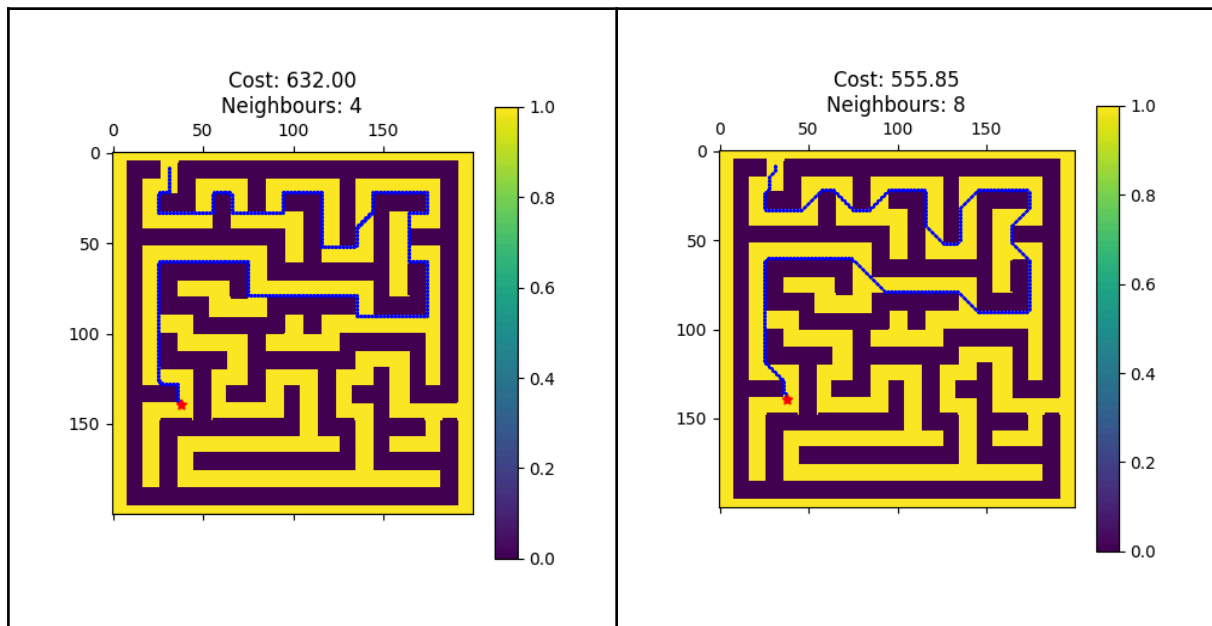


1.2. Grid environment

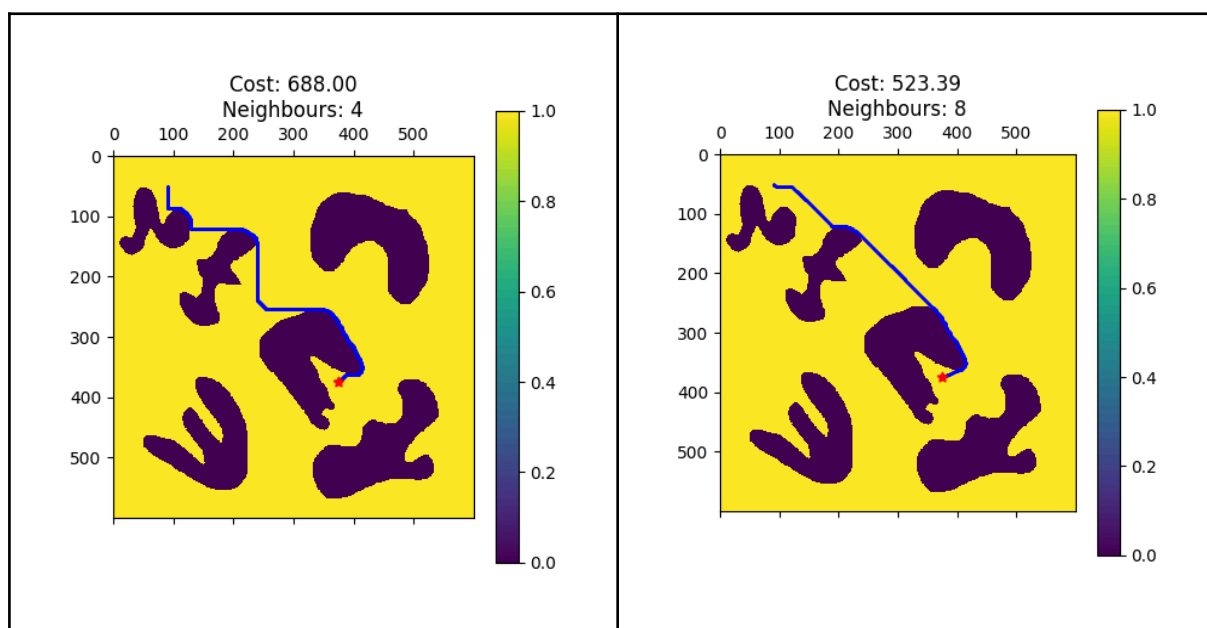
The results of discrete implementation of A* algorithm are given below:

1.2.1. Map 0:



1.2.2. Map 1:**1.2.3. Map 2:**

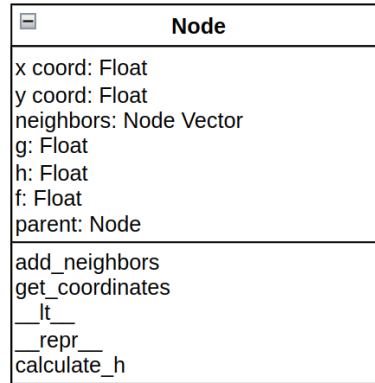
1.2.4. Map3



2. Implementations:

2.1. Node Class

A **Node class** was implemented to have a more efficient and clean code. Node class diagram is given below:



As shown in the node class diagram, each instance of this class contains a set of instance variables: **coordinates** (x , y), represented by two floats, includes a vector of **neighbors**, which stores all the connections that the node has with other nodes. In the case of a grid environment, the neighbors are determined by the chosen connectivity (4 or 8 neighbors), while in a continuous environment (graphs), the neighbors are directly defined by the graph's topology. Additionally, each node has three costs:

g: the cost to reach the node from the start,

h: the heuristic estimate of the cost to reach the goal from the node, in this case calculated as the Euclidean distance between the node and the goal,

f: the sum of g and h, representing the total estimated cost of the path through the node.

Finally, each node also contains a parent node, which is used to trace the path back to the start once the goal has been reached.

The class has 5 variables:

add_neighbors : receives a node and appends it to the vector of neighbors

get_coordinates : returns the x and y coordinates of the node, this can be easily substituted by calling the instance variables x and y.

__lt__ : Overrides the operator "<" action. In this case compares if the f value of the current node is smaller than the f value of a second node

__repr__ : Overrides the print variable. In this case is set to print "**Node(self.x,self.y)**" adding the actual values of x and y

calculate_h : Receives the goal and computes the heuristic cost (euclidean distance) of reaching it from the current node. Also updates the f value of the node performing the addition of the g and h.

2.2. Graph environment

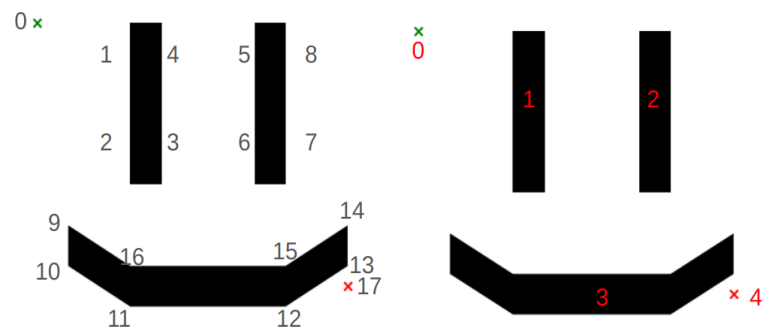
To solve continuous environments (graphs), the problem was divided into two main parts: first, the formation and visualization of the graph, and second the implementation of the A* algorithm.

2.2.1. Graph formation and visualization:

The first step to build the graph was to acquire all the nodes (edges) of the map which was previously binarised so contains only 1 and 0 representing obstacles and free space. The nodes were given in a csv file with the format:

(polygon, coordinate x, coordinate y)

A polygon refers to the number assigned to the shape (obstacle) in the map each node corresponds to. For the start node, this value is 0, and for the goal node, it corresponds to the last polygon. Additionally each node was assigned a number:



By iterating over all the nodes and creating polygons with instances of the Node class, with each node's corresponding coordinates.

The connections were determine in a second file with the shape:

(first node, second node)

This indicated the connection between one node and another. The identifier for this association is the number previously assigned. Iterating over the list of connections the neighbors of each node were added with the class add neighbor.

Having the coordinates of the nodes and its connections was possible to plot the graph and start testing the A* algorithm.

To standardize the plotting size the following border limits were imposed:

The left limit was the more left node - * 1/8 of the distance between the more left and more right nodes.

The right limit was the more right node + * 1/8 of the distance between the more left and more right nodes.

And the same logic was followed for the top and bottom limits.

2.2.2. Continuous A*

The implementation of A* algorithm for the continuous environment corresponds to the following pseudo code:

```

a_star_conti (nodes, cost="euclidean"):
    initialize open_list, closed_list
    start = first node
    goal = last node
    assign f to the first node
    add start to open_list

    while open_list is not empty:
        current_node = node in open_list with the lowest f value
        remove current_node from open_list and add to closed_list

        if current_node is goal:
            path = []
            while current_node has a parent:
                add current_node to path
                current_node = current_node.parent
            return reversed(path)

        for each neighbor in current_node.neighbors:
            if neighbor is in closed_list:
                continue
            if neighbor is not in open_list:
                add neighbor to open_list
                set neighbor.parent = current_node
                if cost == "euclidean":
                    neighbor.g = current_node.g + euclidean_distance(current_node, neighbor)
                else:
                    neighbor.g = current_node.g + 1
                    neighbor.f = neighbor.g + neighbor.h
            else:
                if cost == "euclidean" and neighbor.g > current_node.g +
euclidean_distance(current_node, neighbor):
                    update neighbor.g and f
                    set neighbor.parent = current_node
                else if neighbor.g > current_node.g + 1:
                    update neighbor.g and f
                    set neighbor.parent = current_node
    return None

```

Before implementing the A* on the graph the heuristics of each node were calculated with the calculate_h class function.

2.3. Grid environment

2.3.1. Discrete Implementation of A*:

The implementation of the A* algorithm is divided into a main function (`a_star`) and several helper functions to ensure modularity and clarity. The **`find_neighbours()`** function determines the valid neighboring cells for a given position based on 4-connectivity or 8-connectivity, ensuring neighbors are within the map boundaries and not blocked by obstacles.

`find_neighbours(position, map, neighbours_num)`

Unpack the current position into y (row) and x (column).

Initialize an empty list `neighbours` to store valid neighbors.

Define relative positions for:

4-connectivity: Up, down, left, right.

8-connectivity: Includes diagonal movements.

Select the directions to use based on `neighbours_num` (4 or 8).

For each direction (dy, dx):

Compute the neighbor position as $(ny, nx) = (y + dy, x + dx)$.

Check if (ny, nx) is within grid boundaries and is free (map value = 0).

If valid, add (ny, nx) to the `neighbours` list.

Return the list of valid neighbors.

The **`calculate_h()`** function computes the heuristic cost, using the Euclidean distance between the current position and the goal. The **`reconstruct_path()`** function traces back the optimal path from the goal to the start using a dictionary of visited nodes (`cameFrom`).

`reconstruct_path(cameFrom, current)`

Initialize a list `total_path` with the current position.

While the current position exists in the `cameFrom` dictionary:

Update current to its parent (from `cameFrom`).

Append the updated current to `total_path`.

Reverse `total_path` to get the path from start to goal.

Return the reversed path.

In the **`a_star()`** function, these helpers are combined to handle the core logic of the algorithm: initializing cost maps (`g_map` and `f_map`), managing the priority queue (`open_set`), and expanding nodes by iterating through valid neighbors.

The pseudo code of discrete implementation of A* is below:

`a_star(map, start, goal, num_neigh)`

`g_map` and `f_map` as grids filled with infinity.

Set `g_map[start] = 0` and `f_map[start] = calculate_h(start, goal)`.

Add start to open_set with its f_map value.
 Create an empty cameFrom dictionary to track parents.
 While open_set is not empty:
 Pop the node with the lowest f value as current.
 If current is the goal, **Return** the reconstructed path and $g_map[goal]$.
 Get valid neighbors of current node using **find_neighbours**.
 For each neighbor:
 Compute tentative cost g_score using **calculate_h**.
 If g_score is better than $g_map[neighbor]$:
 Update cameFrom, g_map , and f_map for the neighbor.
 Add neighbor to open_set if not already present.
Return If no path found, return None and infinite cost.

The discrete A* algorithm gave the expected path and cost as shown in section 1.2.

3. Issues and areas of improvement:

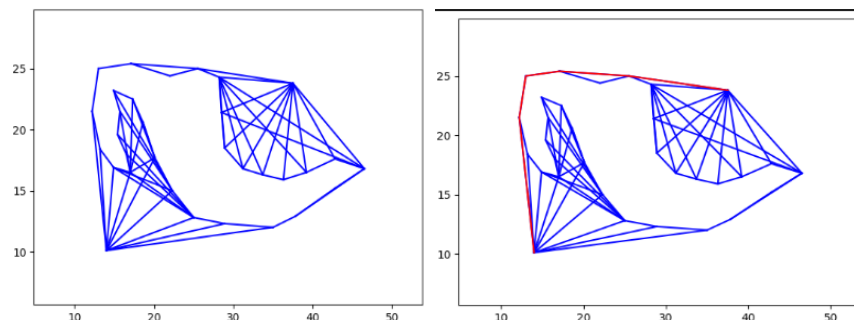
3.1. Same Cost but Different Path:

We implemented A* using our **Node class** and simple helper functions. The output of our implementation, which utilized the Node class, produced slightly different results. While the total cost matched the cost reported in the lab manual, the path trace was different.

Upon analysis, we discovered that the issue arose from how we initialized the nodes in our search loop. Specifically, the g_cost and h_cost values were being reset to infinity during the process. The algorithm assumes that these costs should only be initialized to infinity at the start of the search, not during the pathfinding process. Furthermore, reinitializing g_cost and f_cost can overwrite previously calculated finite costs, leading to slight deviations in the path while still maintaining almost correct overall cost.

3.2. Not optimal path in given environments:

The given graph connectivities of the environments mx and 3, do not represent the entire visibility graph. For instance, in the mx map, the top connection is missing, which results in suboptimal paths. Leading to non optimal paths as they could be obtained using the entire visibility graph.



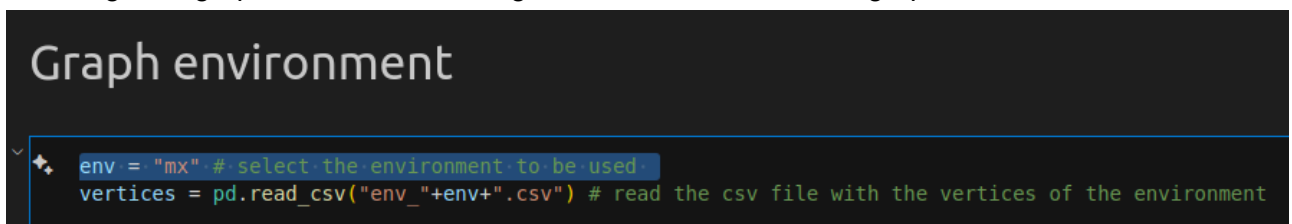
4. Conclusion:

In conclusion, the A* algorithm worked well in both grid-based and continuous environments. It successfully found the shortest path in all tested maps, even as the environments became more complex. The Euclidean distance model, which calculates the cost between nodes based on their distance, was effective in guiding the algorithm to the goal. We also tested a simpler cost model where the cost between nodes was always 1, which gave the same results in simpler maps but led to worse paths in more complex maps like "mx" and ":"). This shows that the algorithm's performance can change depending on the cost model used. When working with continuous environments, the algorithm was able to adapt to different graph structures and connectivity.

Overall, the A* algorithm performed well in solving pathfinding problems, but there are still areas for improvement, especially in ensuring complete graph connectivity and testing different cost models. More testing in more complex environments could help improve its performance even further.

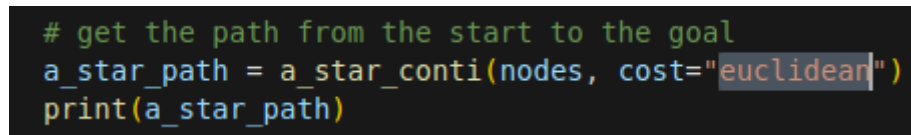
5. How to execute the code:

To change the graph environment change the "env" variable at the graph environment:



```
Graph environment
env = "mx" # select the environment to be used
vertices = pd.read_csv("env_"+env+".csv") # read the csv file with the vertices of the environment
```

To modify the g cost used modify the cost parameter when calling the a_star_conti:



```
# get the path from the start to the goal
a_star_path = a_star_conti(nodes, cost="euclidean")
print(a_star_path)
```

Github repo: <https://github.com/Raulmusito/autonomous-labs.git>