Autonomous Vehicles
Assignment IV: Algorithms for grid-based motion planning

Tommaso Bocchietti 10740309

A.Y. 2024/25

Contents

List of Figures

1 Introduction

The aim of this work is to gain insight into the implementation of some of the most basic grid-based planning algorithms, specifically the `Dijkstra` and `A*` algorithms.
The following sections will provide a detailed description of the requests associated with this assignment, the approach taken to fulfill them, and the discussion of the results obtained.

Tools As for the tools used, `MATLAB` is employed as the main platform for implementing the algorithms and performing the analysis of the data collected during the simulation.

2 Grid-based planning

In the field of robotics, a variety of algorithms are available for path planning, each with its own strengths and weaknesses. One of the most ancient and well known class of algorithms is the so called *Grid-based*, which is based on the discretization of the environment into a grid of cells. Once the environment is represented as a grid, the problem of path planning is reduced to a graph search problem, where the cells of the grid are the nodes of the graph and the edges are the connections between adjacent cells.

2.1 Requests

Among the graph search algorithms, two of the most widely known are the `Dijkstra` and its extension `A*` algorithm. The goal of this assignment can be summarized as follows:

- Implement both the `Dijkstra` and `A*` algorithms;
- Allow the possibility of orthogonal and diagonal movements;
- Test the algorithms on a set of predefined maps and compare the performances.

2.2 Graph search algorithms

In general, given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a graph search algorithm is used to find a path from a starting vertex $s \in V$ to a goal vertex $g \in V$. The main idea is that the algorithm explores the graph by expanding nodes and evaluating their neighbors until it finds the goal node or exhausts all possibilities.

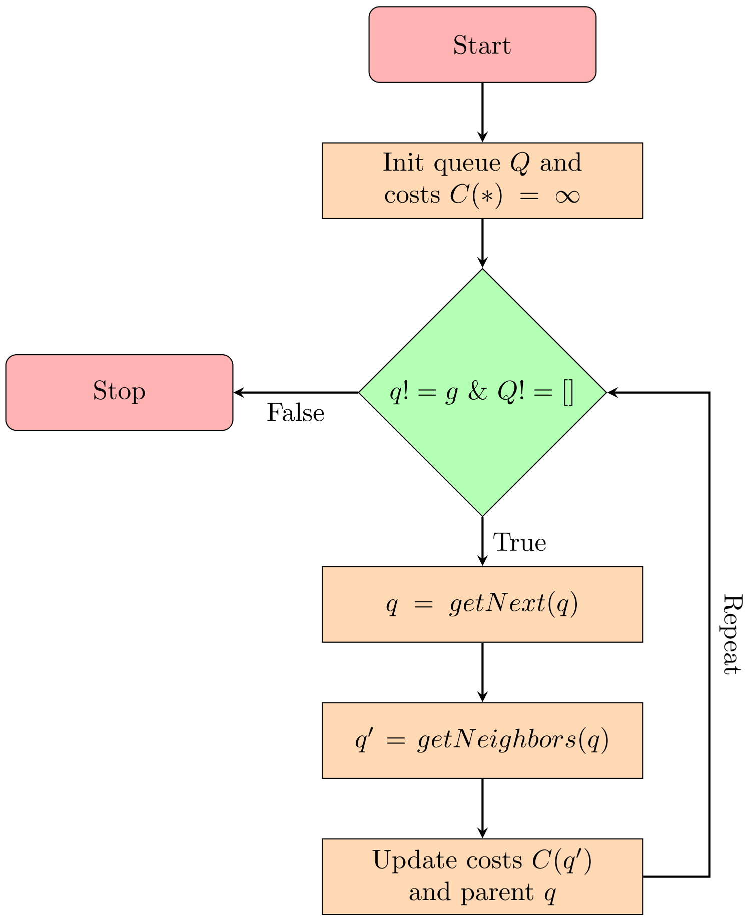


Figure 1: Generic graph search algorithm flowchart

The flowchart in Figure 1 summarizes the main steps of a generic graph search algorithm. The algorithm starts by initializing the open list of nodes and the costs of all nodes to infinity. Then, it enters a loop where based on the previous node visited, its neighbors are evaluated and added to the open list. For each neighbor, the algorithm updates the cost if lower than what already stored and in case updates also the parent node. The loop continues until the open list is empty, or the goal node is reached.
Grid-based algorithms have been proven to be complete and optimal, meaning that they will always find the shortest path if one exists. However, they can be computationally expensive, especially in large environments. There exist many variations built on top of the core algorithm presented in Figure 1. But, what really differentiates them, is the criteria used to select the next node to analyze from the open list. Some of the most common approaches are:

- **Depth-first search (DFS)**: explores as far as possible along each branch before backtracking. It uses a stack to keep track of the nodes to be explored. This approach can easily get stuck in loops.
- **Breadth-first search (BFS)**: explores all the neighbors of a node before moving on to the next level. It uses a queue to keep track of the nodes to be explored.
- **Best-first (a.k.a. Dijkstra)**: selects the node with the lowest cost from the start. This approach is optimal and complete, but can be slow in large environments.
- **A***: an extension of Dijkstra's algorithm that uses a heuristic to estimate the cost from the current node to the goal node. This approach is also optimal and complete, and is often faster than Dijkstra's algorithm in practice, requiring a lower number of nodes to be explored.

In the following sections, we will focus on the Dijkstra and A* algorithms, explaining their mechanisms in choosing the next node to be explored and how they can be implemented to solve the path planning problem in a grid-based environment.

2.2.1 Dijkstra's algorithm

As already mentioned, Dijkstra's algorithm leverages the core structure of the graph search algorithm presented in Figure 1, but with a specific strategy for selecting the next node to explore.

In particular, at each iteration, the algorithm selects the node among the nodes in the open list that has the lowest cost from the start node. The idea is so to select $q' \in Q$ such that:

$$q' = minarg_{q \in Q} C(q) \tag{1}$$

Where $C(q)$ is the cost of reaching node q from the start node s. The cost is calculated as the sum of the cost of the previous node q and the cost of moving from q to q'.
From an implementation point of view, the algorithm can be easily implemented using a priority queue to store the nodes in the open list. The priority queue allows for efficient retrieval of the node with the lowest cost, which is crucial for the performance of the algorithm. Listing 1 shows a possible implementation of the Dijkstra's algorithm. The algorithm takes as input the graph G, the start node s, and the goal node g.

```matlab
function [exist, distances, parents] = dijkstra(G, node_initial, node_final)
% DIJKSTRA Finds the shortest path between two nodes in a graph
% using Dijkstra's algorithm.

N = length(G.nodes);
distances = inf(N, 1);
parents   = NaN(N, 1);
visited   = false(N, 1);

% Initialization
ID_current = node_initial.ID;
distances(ID_current) = 0;

while (ID_current ~= node_final.ID && ~all(visited))

    unvisited = find(~visited);
    [~, ID_current] = min(distances(unvisited));
    ID_current = unvisited(ID_current);

    visited(ID_current) = true;

    % For each neighbor not yet visited
    IDs_neighbors = G.adjacents{ID_current};
    IDs_neighbors = IDs_neighbors(~visited(IDs_neighbors));
    for ID_neighbor = IDs_neighbors'

        distance = ...
            distances(ID_current) + ...
            G.getEdgeByConnection(ID_current, ID_neighbor).weight;

        if distance < distances(ID_neighbor)
            distances(ID_neighbor) = distance;
            parents(ID_neighbor) = ID_current;
        end

    end

end

exist = ~isinf(distances(node_final.ID));

end
```

Listing 1: Code for the implementation of Dijkstra's algorithm.

The algorithm starts by initializing the distances of all nodes to infinity, except for the start node, which is set to zero. Then, it enters a loop where it selects the node with the lowest cost from the open list and explores its neighbors. For each neighbor, it updates the cost if the new cost is lower than the previously stored cost and updates the parent node accordingly. The loop continues until the goal node is reached, or all nodes have been visited. Finally, the algorithm returns the distances and parents of each node, which can be used to reconstruct the shortest path from the start node to the goal node.

2.2.2 A* algorithm

A* algorithm is an extension of Dijkstra's algorithm that uses a heuristic to estimate the cost from the current node to the goal node. The heuristic is a function that provides an estimate of the cost to reach the goal from a given node, and it is used to guide the search towards the goal more efficiently.
In particular, the A* algorithm selects the next node to explore based on the sum of the cost from the start node to the next node and the estimated cost from the next node to the goal node. The selection is done as follows:

$$q' = min\, \arg_{q \in Q}(C(q) + h(q)) \tag{2}$$

Where $h(q)$ is the heuristic function that estimates the cost from node q to the goal node g. The heuristic function can be any function that provides an estimate of the cost, but it is important that it is admissible, meaning that it never overestimates the true cost to reach the goal. For the sake of simplicity, and in particular leveraging the meaningful geometrical interpretation of the grid-based environment, we will use the Euclidean distance as the heuristic function. The A* algorithm can be implemented similarly to Dijkstra's algorithm, with the addition of the heuristic function. Listing 2 shows a possible implementation of the A* algorithm. The algorithm takes in the same input as the Dijkstra's one.

```
function [exist, distances, parents] = astar(G, node_initial, node_final)
% ASTAR Finds the shortest path between two nodes in a graph
% using the A* algorithm.

N = length(G.nodes);
distances = inf(N, 1);
costs     = inf(N, 1);
parents   = NaN(N, 1);
visited   = false(N, 1);

% Initialization
ID_current = node_initial.ID;
distances(ID_current) = 0;
costs(ID_current) = Node.euclideanDistance(node_initial.state, node_final.state);

while (ID_current ~= node_final.ID && ~all(visited))

    unvisited = find(~visited);
    [~, ID_current] = min(costs(unvisited));
    ID_current = unvisited(ID_current);

    visited(ID_current) = true;

    % For each neighbor not yet visited
    IDs_neighbors = G.adjacents{ID_current};
    IDs_neighbors = IDs_neighbors(~visited(IDs_neighbors));
    for ID_neighbor = IDs_neighbors'

        distance = ...
            distances(ID_current) + ...
            G.getEdgeByConnection(ID_current, ID_neighbor).weight;

        if distance < distances(ID_neighbor)

            distances(ID_neighbor) = distance;
            parents(ID_neighbor) = ID_current;

            node_neighbor = G.getNodeByID(ID_neighbor);
            costs(ID_neighbor) = distance + Node.euclideanDistance(node_neighbor.state,
    node_final.state);

        end

    end

end

exist = ~isinf(distances(node_final.ID));

end
```

Listing 2: Code for the implementation of A* algorithm.

One can clearly notice that the structure of the A* algorithm is basically the same as the Dijkstra's algorithm, except for the addition of the heuristic function in the cost calculation. Notice also that the heuristic function is only used to update the cost of the nodes in the open list, and it is not used to update the distances of the nodes from the start node.
Given that now the algorithms is somehow guided, one can expect that the A* algorithm will be faster than the Dijkstra one, as fewer nodes will be explored.

3 Algorithm Testing

Now that the algorithm has been implemented, we can proceed presenting at first the test environment that have been used to test them and then the results of the tests.
Figures 2 show the four image maps used to test the algorithm. Even if the images here below are of size 300x300, we preferred to scale down to 30x30 given that each pixel is then converted into a node of the graph and so to avoid huge graphs and long computation times. Notice also that while the white areas are free space, the one marked in black or gray are to be considered as obstacles. Finally, the green dot is the starting point of the vehicle, while the red one is the goal.

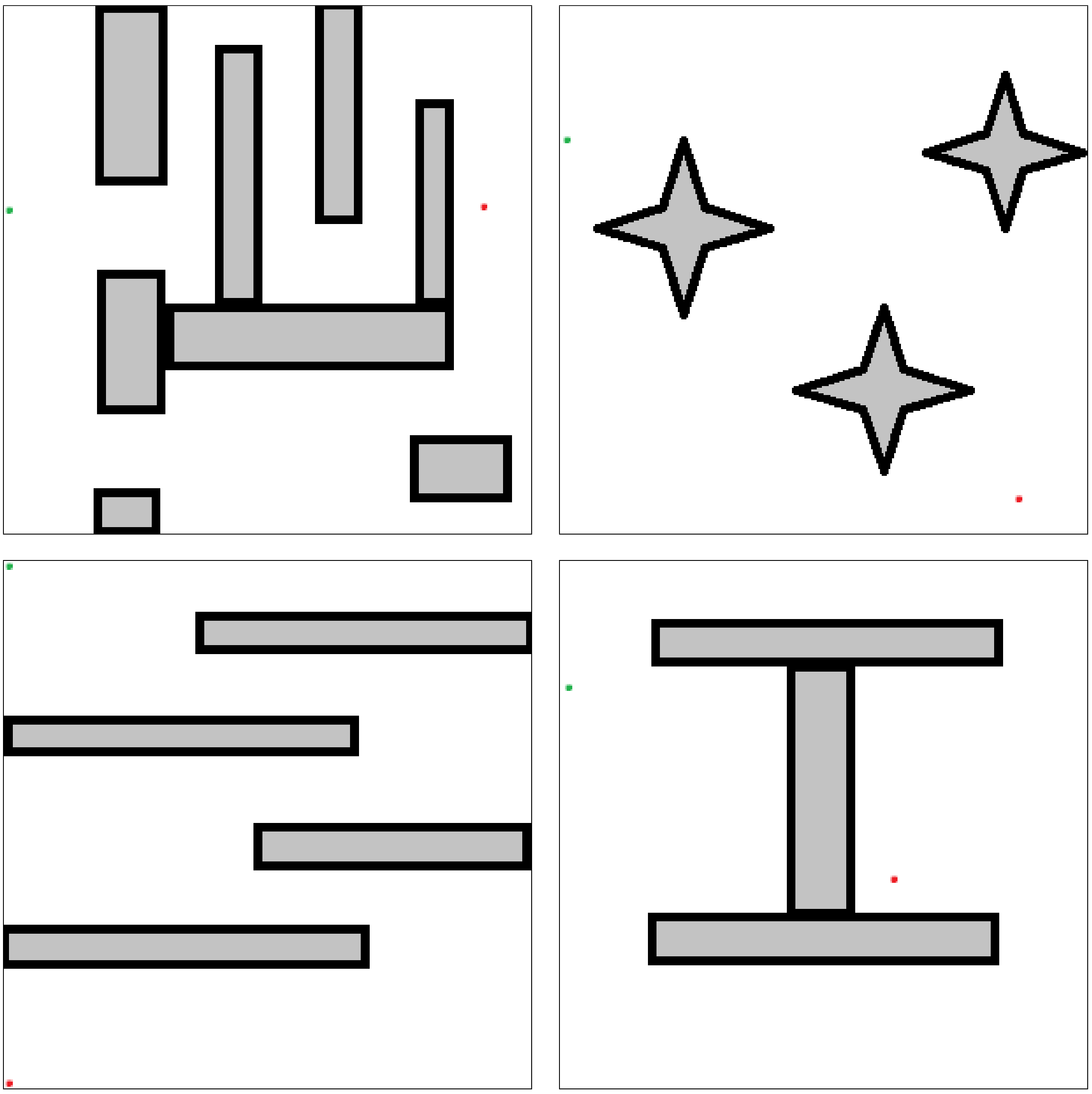


Figure 2: Test maps used for the algorithm testing.

In the following, results obtained running both the A* and the Dijkstra algorithms on each of the four maps are presented.
The plotting of both the (geometrical) connected graph and the best path found is also shown. Notice that each test case has been run allowing or disabling the diagonal movement of the vehicle, which can be achieved during the graph generation phase modifying the connection of the nodes.
Tables presenting the main metrics of the tests are also shown. The metrics are the following:

- **Elapsed time**: here only the time needed to run the searching algorithm is considered (i.e. we are excluding both the generation of the graph and the plotting of the results);
- **Nodes analyzed**: number of nodes analyzed (visited) before reaching the goal;
- **Path length**: length of the path found by the algorithm. This also correspond to the path cost, given that the cost of each edge is equal to its geometrical length.

3.1 Map 1

Results obtained running the A* and Dijkstra algorithms on Map 1 are shown in Figures 3 and Table 1. About the figures, the upper row shows the graph generated running the Dijkstra algorithm while the lower one shows the graph generated running the A* algorithm. The left column shows the graph generated allowing only orthogonal movements, while the right one shows the graph generated allowing also diagonal movements. The path found by the algorithm is shown in red, while the graph is shown in black.

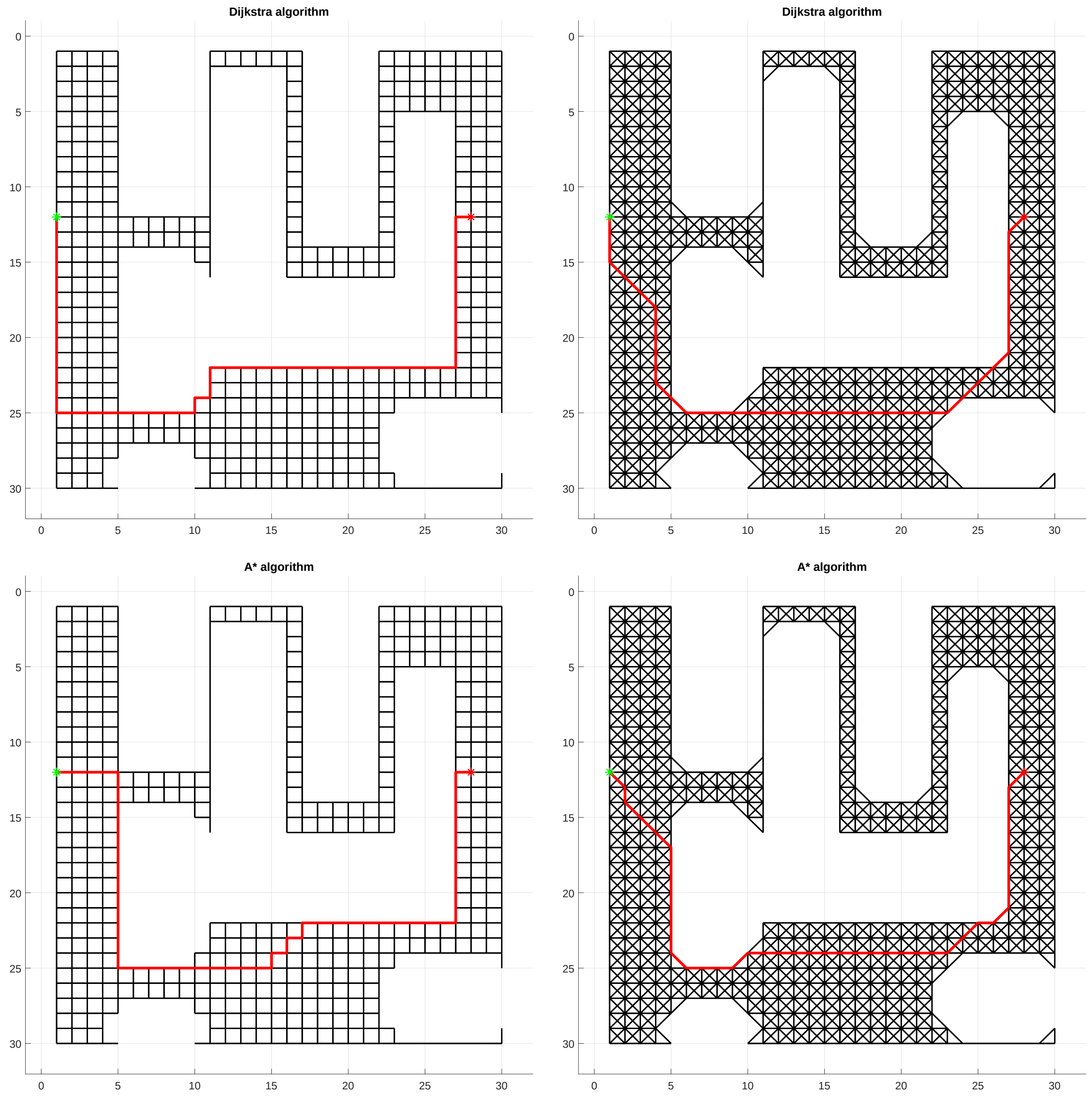


Figure 3: Map 1: Graph and path found by the algorithms.

Algorithm	Diagonal	Elapsed time (ms)	Nodes analyzed	Path length (m)
Dijkstra	No	624	456	53
	Yes	1592	453	47
A*	No	339	378	53
	Yes	1227	339	47

Table 1: Map 1: Results of the tests.

From the results presented in Table 1 and in Figure 3, we can already notice that while the cost of the path found by the two algorithms is equivalent, the number of nodes considered by the A* algorithm is significantly lower than the one considered by the Dijkstra algorithm. This behavior is expected due to the use of the heuristic function to guide the search of the A* algorithm. The advantage in terms of computational time is also evident, with the A* algorithm outperforming the Dijkstra algorithm.

3.2 Map 2

Results obtained running the A* and Dijkstra algorithms on Map 2 are shown in Figures 4 and Table 2. The same layout of the previous map is used for the figures and the same metrics are used for the table.

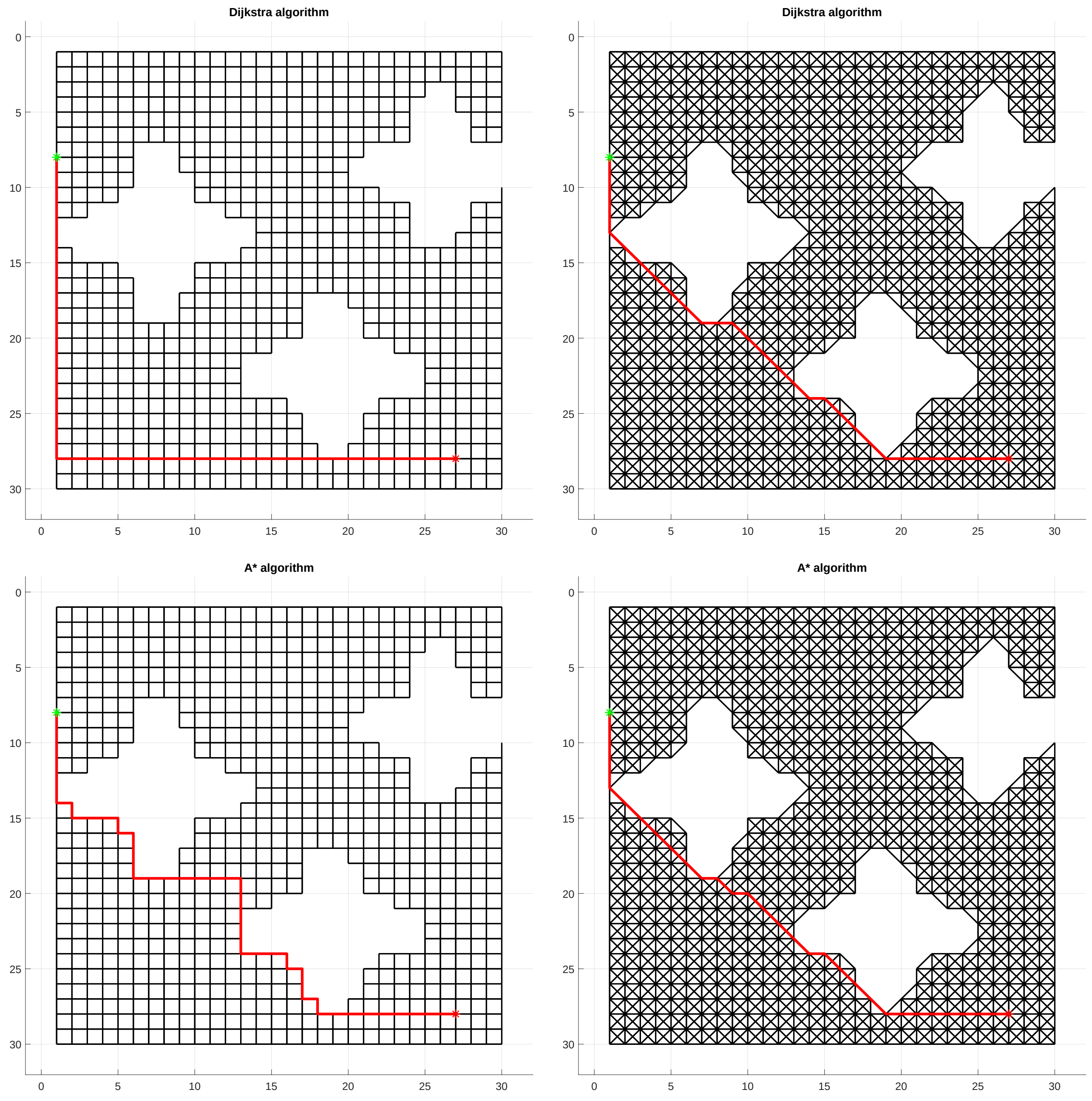


Figure 4: Map 2: Graph and path found by the algorithms.

Algorithm	Diagonal	Elapsed time (ms)	Nodes analyzed	Path length (m)
Dijkstra	No	1009	731	46
	Yes	3963	742	37
A*	No	732	490	46
	Yes	1434	233	37

Table 2: Map 2: Results of the tests.

Similarly to the previous map, the results presented in Table 2 and in Figure 4 show that while the cost of the path found by the two algorithms is equivalent, the number of nodes considered by the A* algorithm is significantly lower than the one considered by the Dijkstra algorithm. Similar conclusions can be drawn in terms of computational time and performance of the two algorithms.

As an additional note, in this case we can also appreciate that the number of analyzed nodes by A* is significantly lower than the one analyzed by Dijkstra, particularly in case of diagonal movements. This is, however, expected. Looking at the final path, we observe that is almost equivalent to the diagonal connecting ideally the start and goal points. This means that the heuristic function used by A* is even more effective in this case, given that very few obstacles are present along the optimal path designed by the heuristic function itself and so fewer death ends are encountered. In other words, if the optimal path turns out to be a straight line connecting the start and goal points, the A* algorithm would analyze a number of nodes equal to the number of nodes along the straight line, while the Dijkstra algorithm would still analyze a much larger number of nodes given that no information is used to guide the search.

3.3 Map 3

Results obtained running the A* and Dijkstra algorithms on Map 3 are shown in Figures 5 and table. The same layout of the previous map is used for the figures and the same metrics are used for the table.

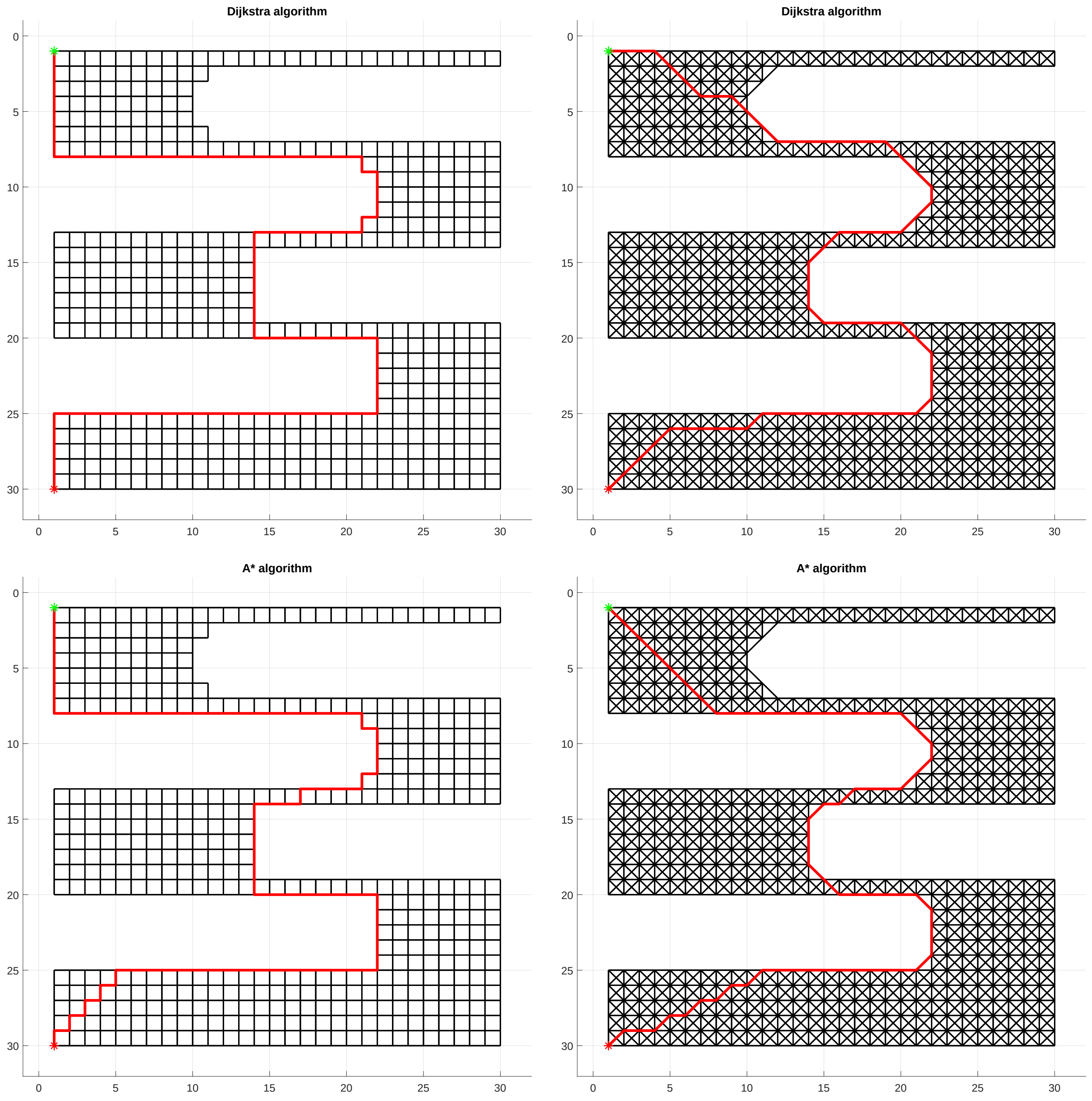


Figure 5: Map 3: Graph and path found by the algorithms.

Algorithm	Diagonal	Elapsed time (ms)	Nodes analyzed	Path length (m)
Dijkstra	No	567	592	87
	Yes	2461	592	74
A*	No	496	497	87
	Yes	1780	452	74

Table 3: Map 3: Results of the tests.

Again, similar considerations as before can be drawn from the results presented in Table 3 and in Figure 5.

3.4 Map 4

Results obtained running the A* and Dijkstra algorithms on Map 4 are shown in Figures 6 and table. The same layout of the previous map is used for the figures and the same metrics are used for the table.

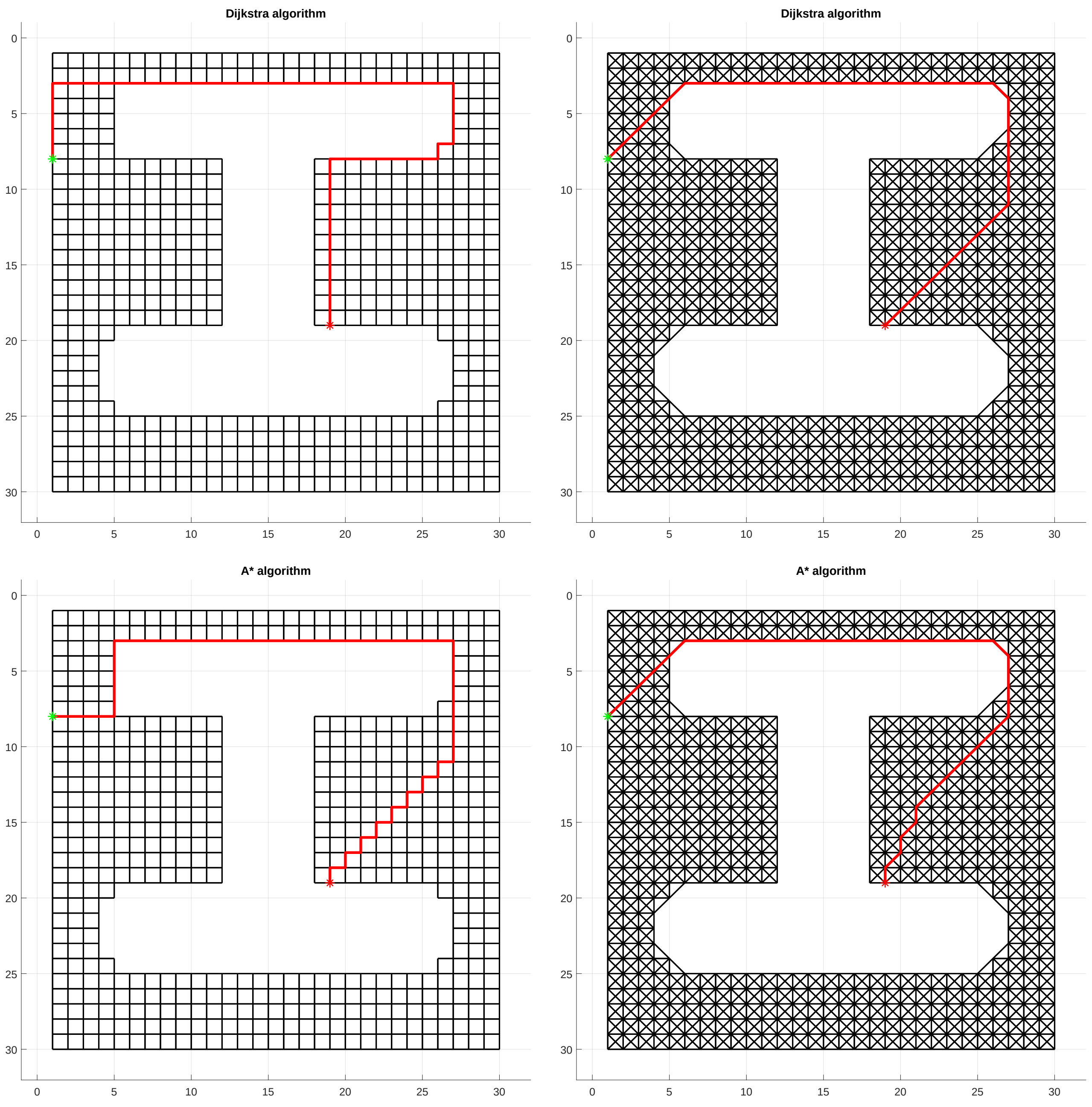


Figure 6: Map 4: Graph and path found by the algorithms.

Algorithm	Diagonal	Elapsed time (ms)	Nodes analyzed	Path length (m)
Dijkstra	No	725	650	55
	Yes	2799	650	47
A*	No	640	562	55
	Yes	2060	450	47

Table 4: Map 4: Results of the tests.

Once again, even from the tests on Map 4, similar conclusions as before can be drawn from the results presented in Table 4 and in Figure 6.

4 Conclusions

In this short report, we have presented the results of the fourth assignment of the course on Autonomous Vehicles. We have briefly explained the grid-based algorithm for path planning, and deepened into the implementation of both the Dijkstra and A* algorithms.
These algorithms have been tested on four different scenarios, with different obstacles and connectivity configurations (i.e., allowing or not diagonal movements). The results have shown that the A* algorithm is generally faster than the Dijkstra algorithm as he uses heuristics to guide the search towards the goal allowing for a more efficient exploration of the search space. This is also reflected in the number of nodes explored, which is significantly lower for the A* algorithm compared to the Dijkstra algorithm.
As also proved by the theroetical analysis performed for this grid-based algorithm, both algorithms are complete and optimal, meaning that they will always find the shortest path if one exists. Obtained results have reflected this theoretical analysis, as both algorithms have been able to find the optimal path in all tested scenarios.
One of the main limitations of the grid-based algorithm is that they can be computationally expensive, especially in large or hiperdimensional spaces. A quick solution to this problem is to use a coarser grid, which can reduce the number of nodes and edges in the graph, but at the cost of accuracy and possibly missing the optimal path, or even failing to find a path at all.
Because of the above limitation, the grid-based algorithm is not suitable for all applications, especially in dynamic environments where the obstacles can change over time. In these cases, more advanced algorithms based on combinatorial techniques or sampling-based methods, such as Rapidly-exploring Random Trees (RRT) or Probabilistic Roadmaps (PRM), may be more appropriate.

A MATLAB Graph implementation

During the implementation of the Dijkstra and A* algorithms, a needs for a solid and efficient graph representation arose. In order to achieve this, a set of MATLAB classes was created to represent the graph, its nodes and its edges.
In the following, the complete code used for this purpose is shown. One can refer back to the graph search algorithm implementation to understand how these classes are used.

```matlab
classdef Graph < handle
    % GRAPH Class for representing a graph structure with nodes and weighted edges.
    % Provides methods for adding and retrieving nodes, edges, and adjacency data.

    properties
        nodes Node
        edges Edge
        adjacents cell
    end

    methods
        function obj = Graph()
            obj.nodes = Node.empty;
            obj.edges = Edge.empty;
            obj.adjacents = {};
        end

        function ID = addNode(obj, state)
            ID = length(obj.nodes) + 1;
            obj.nodes(ID, 1) = Node(ID, state);
            obj.adjacents{ID, 1} = [];
        end

        function ID = addEdge(obj, ID_A, ID_B, weight)
            ID = length(obj.edges) + 1;
            obj.edges(ID, 1) = Edge(ID, ID_A, ID_B, weight);
            obj.adjacents{ID_A, 1}(end+1, 1) = ID_B;
        end

        function node = getNodeByID(obj, ID)
            node = obj.nodes(ID);
        end

        function edge = getEdgeByID(obj, ID)
            edge = obj.edges(ID);
        end


        function nodes = getNodesByState(obj, state)
            nodes = Node.empty;
            for node = obj.nodes'
                if Node.compareStates(node.state, state)
                    nodes(end+1) = node;
                end
            end
        end


        function edge = getEdgeByConnection(obj, ID_A, ID_B)
            for edge = obj.edges([obj.edges.ID_A] == ID_A)'
                if (edge.ID_B == ID_B)
                    return
                end
            end
        end


        function adjacents = getAdjacents(obj, node_ID)
            adjacents = obj.adjacents{node_ID};
        end
    end

end
```

Listing 3: Graph class used to represent a weighted and directed/undirected graph.

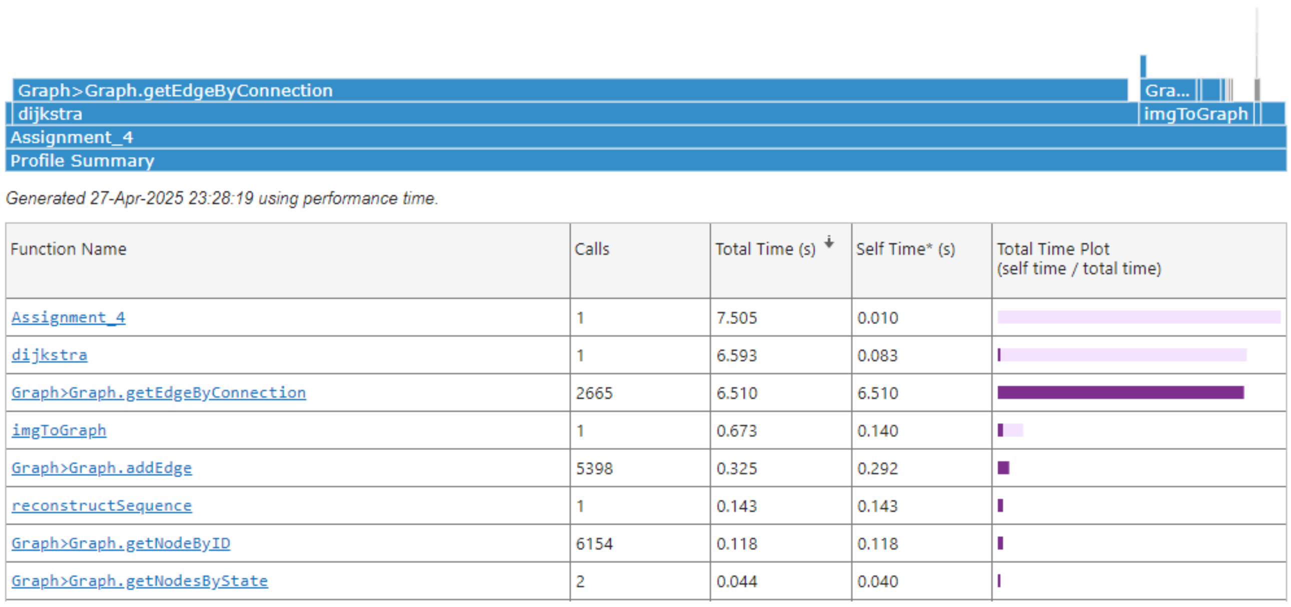
```
classdef Edge
    % EDGE Represents a weighted connection between two nodes in a graph.
    % Stores IDs of connected nodes and the edge weight.

    properties
        ID double
        ID_A double
        ID_B double
        weight double
    end

    methods
        function obj = Edge(ID, ID_A, ID_B, weight)
            obj.ID = ID;
            obj.ID_A = ID_A;
            obj.ID_B = ID_B;
            obj.weight = weight;
        end

        function isConnection = isConnectionTo(obj, ID_A, ID_B)
            isConnection = obj.ID_A == ID_A & obj.ID_B == ID_B;
        end
    end

end
```

Listing 4: Edge class used to represent a weighted and directed edge in the graph.

```
classdef Node
    % NODE Represents a graph node with a unique ID and a state vector.
    % Includes utility methods for comparison and distance calculation.

    properties
        ID double
        state double
    end

    methods
        function obj = Node(ID, state)
            obj.ID = ID;
            obj.state = state;
        end

        function isEqual = eq(node_A, node_B)
            isEqual = Node.compareStates(node_A.state, node_B.state);
        end
    end

    methods (Static)
        function dist = euclideanDistance(state_A, state_B)
            dist = norm(state_A - state_B);
        end

        function equal = compareStates(state_A, state_B)
            equal = isequal(state_A, state_B);
        end
    end

end
```

Listing 5: Node class used to represent a node in the graph.

As a final note, a code profiler was used to test the performance of the graph implementation. The main bottleneck was found in the `getEdgeByConnection` method of the `Graph` class, as shown in Figure 7.



Generated 27-Apr-2025 23:28:19 using performance time.

Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (self time / total time)
Assignment_4	1	7.505	0.010	
dijkstra	1	6.593	0.083	
Graph>Graph.getEdgeByConnection	2665	6.510	6.510	
imgToGraph	1	0.673	0.140	
Graph>Graph.addEdge	5398	0.325	0.292	
reconstructSequence	1	0.143	0.143	
Graph>Graph.getNodeByID	6154	0.118	0.118	
Graph>Graph.getNodesByState	2	0.044	0.040	

Figure 7: Profiler output for the graph implementation.

The author of this report is aware that the graph implementation can be improved in terms of performance by using perhaps a caching mechanism or a more efficient data structure. However, this was not the main focus of the assignment and further investigation is left as future work.