Lab 4 Report: Graph Search – A* algorithm

Mohammad Rami Koujan

1 Introduction

The aim of this lab assignment is to program the A* algorithm which is a graph search algorithm so as to solve a visibility graph. Solving the visibility graph means that finding the shortest path in this graph that connects a start node to a goal node. In fact, A* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance travelled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node.

2 A* Algorithm

This section presents in details the implementation of the A* algorithm in Matlab program. In order to test this algorithm, some visibility graphs were generated from a topological path planning algorithm (Rotation Plane Sweep-RPS). Therefore, two main arrays are used to represent the visibility graph: the "vertices", the set of nodes to be explored, which contains information about the coordinates of each vertex along with the object number that it belongs to, and the "edges" which represents the edges of this graph by storing the indices of the vertices that it connects together. Basically, the implemented algorithm works as follows:

- It defines and initializes a number of parameters at the beginning:
  1. "closedset" to store the explored nodes. This variable is empty at the beginning. By explored, it is meant that those nodes are known as the ones that have the minimum cost to reach the goal along their discovered path.
  2. "openset" to store the nodes that are the neighbours of the already explored nodes. This variable is initialized to the start node at the beginning of the program.
  3. "came_from" which is used as pointer to point to the explored nodes. Hence, this variable is an array of the same size as the "closedset" and it points to each of these nodes by storing the number of the previously explored node along the same path.
  4. "g_score" stores the cost from start along best known path.
  5. "f_score" stores the estimated total cost from start to goal through each node in the "openset".
- Then, a loop is defined to repeat the following steps:
  1. Initializing the "current" variable to the node that has the lowest f_score value.
  2. checking if the "current" is equal to the goal node in order to reconstruct the best path, returned, and stop executing the program. Otherwise, "current" node is removed from the openset and added to the closedset.
  3. For every neighbour of the "current" node, the program calculates a tentative score for this neighbour which is indeed the cost of reaching this neighbour along the currently explored path. This tentative score is calculated by simply adding the g_score of the "current" node to the euclidean distance between those two nodes. The euclidean distance is considered here as the cost of the edge connecting the "current" node and its neighbour.
  4. This neighbour is checked weather it is already in the closedset and has g_score that is less than the tentative cost or not. If yes, the program will move to another neighbour since this one is already discovered with a lower cost previously. Thus, it should not be explored again from a higher cost path. Otherwise, if the neighbour is not in the openset or it is in the openset but with g_score that is higher than the tentative score for this node, the following parameters are modified:

(a) "came_from": it is updated with the "current" node in the position dedicated for the neighbour as a pointer to the it.

(b) "g_score: it is updated with the tentative score in the position dedicated for the neighbour as it is new g_score.

(c) "f_score: it is updated with the g_score of the neighbour node plus the heuristic cost estimate from the neighbour to the goal. The heuristic cost estimate is considered as the euclidean distance in the implemented function.

(d) "openset": updated by adding the neighbour to it, if the neighbour is not already added.

The current running loop keeps iterating until the "openset" becomes empty. However, a solution must be obtained before exiting this loop.

In order to reconstruct the final path, another function is used to generate this path iteratively based on the the pointer ("came_from) array and the goal node. The idea is to recursively moves from one node to another based on the value of the mentioned pointer which guides the tracking process along the optimal path.

3 Testing and Results

This section presents and discusses the results of testing the implemented A* algorithm with some visibility graphs generated from the RPS algorithm, which is implemented in the second lab assignment.

In the first test, the following "vertices" are used to generate the visibility graph by using the RPS function which returns the "edges" parameter. Then, both of these parameters are used as an input to the A* algorithm.

$$Vertices = \begin{bmatrix} 0.7807 & 9.0497 & 0 \\ 3.0322 & 8.9912 & 1 \\ 1.3655 & 6.7105 & 1 \\ 4.1140 & 4.0497 & 1 \\ 6.2778 & 8.2310 & 1 \\ 8.2953 & 5.8333 & 2 \\ 5.6345 & 2.6170 & 2 \\ 9.1433 & 1.9152 & 2 \\ 11.4825 & 6.9444 & 2 \\ 10.2544 & 0.5702 & 3 \end{bmatrix}$$

The following figure shows the topological map (blue), visibility graph (red), and the best returned path by the A* algorithm (green). In this example, nodes number 1 and 10 are the start and goal nodes respectively.

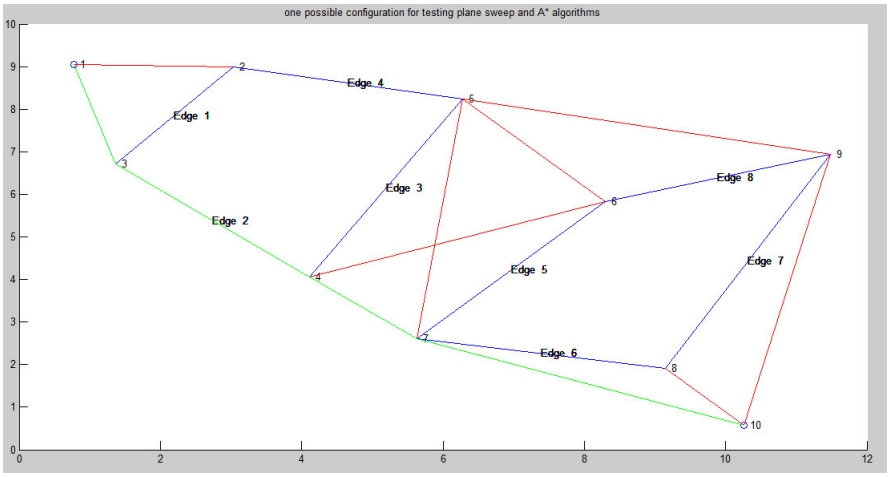

Figure 1: Testing A* algorithm with a visibility graph (1 is the start node and 10 is the goal) where the green path shows the optimal path

The returned path and minimum cost by the A* algorithm for the shown map are:

path =[1 3 4 7 10]
minCost=13.3788

Since the cost of the nodes in the implemented A* function is the euclidean distance between them, the returned minCost shows that A* function has returned the minimum possible distance that has to be passed in order to reach the goal from the start node.

Another visibility graph was generated to test the implemented A* algorithm with different vertices and the following results have been obtained:

$$Vertices = \begin{bmatrix} 0.6053 & 7.9971 & 0 \\ 1.0439 & 6.8567 & 1.0000 \\ 2.9737 & 8.2602 & 1.0000 \\ 3.9386 & 6.3304 & 1.0000 \\ 1.9795 & 5.3655 & 1.0000 \\ 6.4532 & 8.3187 & 2.0000 \\ 5.1959 & 6.6228 & 2.0000 \\ 6.3070 & 4.6637 & 2.0000 \\ 8.7339 & 6.2719 & 2.0000 \\ 8.4708 & 7.8801 & 2.0000 \\ 3.2368 & 4.8684 & 3.0000 \\ 0.8684 & 3.9620 & 3.0000 \\ 1.2485 & 2.5585 & 3.0000 \\ 3.3538 & 2.4123 & 3.0000 \\ 4.8450 & 4.0497 & 3.0000 \\ 6.5994 & 3.9327 & 4.0000 \\ 6.5409 & 2.0906 & 4.0000 \\ 8.5877 & 2.2076 & 4.0000 \\ 8.6170 & 4.6053 & 4.0000 \\ 9.4357 & 7.2368 & 5.0000 \\ 11.1608 & 4.0789 & 5.0000 \\ 10.3129 & 7.9094 & 5.0000 \\ 10.3713 & 1.5351 & 6.0000 \end{bmatrix}$$

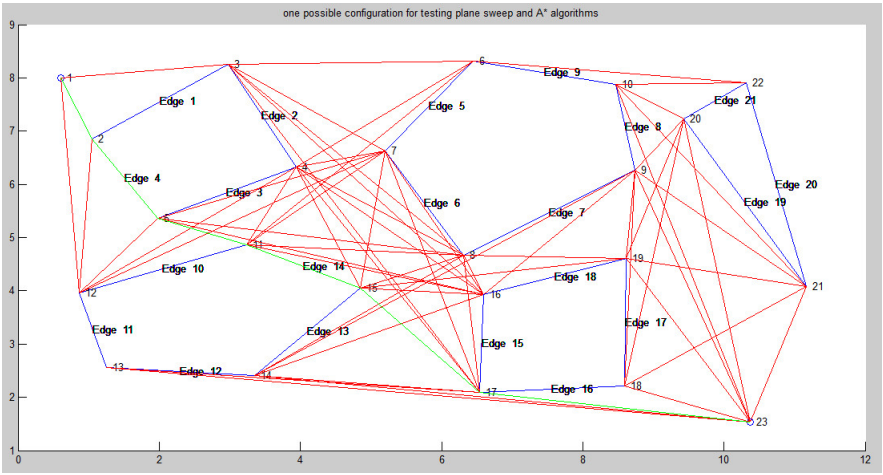

Figure 2: Testing A* algorithm with a visibility graph (1 is the start node and 23 is the goal) where the green path shows the optimal path

path =[1 2 5 11 15 17 23]
minCost=12.6005

The minCost here also gives the minimal euclidean distance that has to be travelled form the start node so as to reach the gaol.

A third example with interactively generated topological map is used to test the algorithm. Below are the obtained results.

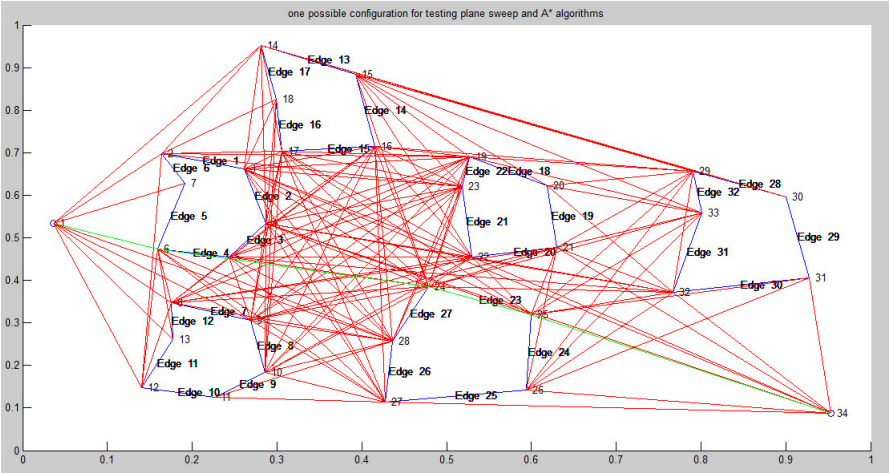

Figure 3: Testing A* algorithm with a topological map (1 is the start node and 34 is the goal) where the green path shows the optimal path

path =[1 6 24 25 34]
minCost=1.0288

The following table demonstrates the time spent by the A* algorithm in order to get the final result for the three preceding examples.

Measured parameters	Elapsed time (sec)	Number of nodes
test 1	0.018862	10
test 2	0.020938	23
test 3	0.02402	34

Table 1: Elapsed time for getting the results of the three previous examples

Table 1 shows clearly that higher number of nodes in the generated visibility graph takes more time to be precessed by the A* and eventually to obtain the final results. This is an expected consequence because larger number of nodes means more iterations have to be executed and more operations have to be carried out by the function.

4 Conclusion

In this lab assignment, a graph search algorithm (A*) is designed, implemented, and tested on different visibility graphs generated by the Rotational Plane Sweep algorithm. The obtained results demonstrate the optimality of this algorithm, its dependency on the heuristic cost which should be optimistic, and the higher time that is required to generate the results for larger maps.