RRT

INTRODUCTION

In this exercise, the goal is to implement a MATLAB code for RRT (Rapidly exploring Random Tree) Algorithm. In particular, given a predefined map containing an obstacle (in this case a polygon that can be described as the union of an horizontal and vertical rectangle), the algorithm selects random nodes within the map and based on them it adds new nodes to a tree (checking the no collision condition). This tree, which will incrementally increase with the iterations, grows until the Goal node is reached. At the end, the tree densely covers the map and RRT Algorithm returns the Shortest Path Tree from the initial Starting Node position to the final Goal Node position, of course avoiding the defined obstacle.

RRT is an algorithm that yields to good results, providing that parameters like number of iterations, stepsize, etc are chosen keeping under consideration the application. Also, noticed that with an increase of the number of nodes, the required computational effort for the algorithm increases a lot. To reduce this heavy drawback, many modification to original algorithm can be done. However, since this is a didactic assignment, a simple version of RRT has been implemented following the example explained during lecture (in particular, RRT ver. 2 has been implemented).

CODE DESCRIPTION

The first part of the code, provided by the Teacher, is the part where the obstacle is defined. In particular, it is defined with the BinaryOccupancyGrid function, which creates a grid where obstacle is defined with a logical value 1 (probability of having an obstacle = 1) and free space with a logical value 0.

Then, stepSize (also called EPSILON), startState and goalState are defined.

The following section is the implementation of the RRT Algorithm. Firstly, parameters need to be defined.

In particular, it can be defined: EPS (equal to provided stepSize in this case), K, max number of iterations and threshold (fundamental to determine found path condition).

Then, in this code the tree has been implemented as a set of nodes. Each node is a struct type data with three field: coord, cost and parent. Of course, coord is the field where x and y coordinates of the node (in the map) are stored. Cost, is a value respresenting the overall distance of the node from the start_node. In particular, it is defined in an inductive way: it is the sum of the distance between the node and its parent (near_node) and the cost of the parent itself.

Cost actually is not fundamental for the algorithm but it is necessary, as I will explain later, for an additional improvement showing the absolute shortest path.

Parent is a value representing the index within the nodes set (the tree) of the nearest node to the current node.

1. G.add.vertex(q_{init})

So in the initialization part, the set nodes is initialize with the start_node.

2. For k = 1 to K

This is the beginning of the core of the algorithm, the for loop: the algorithm iterates for a number of iteration at most equal to the defined K value. If the algorithm can find a Path within K iterations → it returns the Path. Otherwise, if maximum number of iterations k is reached, the algorithm displays:

```
'No Path found. Increase number of iterations and retry.'
```

Here, the predefined value of K is 5000. This is to be sure that path is found substantially every time (for a stepSize of 1). We will see however that this value sometimes needs to be adjust if the stepSize is further reduced.

2.1 q_{rand} ← Random Selected Vertex

q_rand is selected as a point with integer coordinate between [0,25] through rand(1)*25 function. The integer coordinates are not fundamental but without

this condition there is the risk that the code iterates many times due to the high number of possible directions that expanding tree can have.

2.2 $q_{near} \leftarrow$ Nearest_Vertex(G, q_{rand})

A for loop is used, for as many iterations as actual length of the nodes array, to determine which is the closest node to q_rand node in the nodes set. Distance between q_rand and all the other points is computed using the dist() function, that simply implements the formula for the distance between two points.
q_near is identified by idx variable representing the position of the node within the nodes set. Then, q_near is simply = nodes(idx).

2.3 $q_{add} \leftarrow q_{near} + \gamma(q_{rand} - q_{near})/ |q_{rand} - q_{near}|$

This step characterized the version 2 of RRT algorithm: thanks to this updating, algorithm is way faster and reliable because without this step, probability of having collision between q_near and q_rand is larger. The drawback is that this step affects the algorithm from a computational point of view. In the code, this has been implemented through a “steer function” which has been defined at the end of the script. This function, simply applies the formula $q_{near} + \gamma(q_{rand} - q_{near})/ |q_{rand} - q_{near}|$ if the distance (val) between q_rand and q_near is bigger than the maximum allowed step size. Since the direction is a vector with x and y component, x and y value of q_near are computely separately.
Once the q_add.coord have been determined, the q_add.cost is equal to the distance between q_add and q_near + the already defined cost for q_near; q_add.parent is equal to the idx of the parent node in the nodes set.

2.3.1 If detect_collision(q_{add} , q_{near})

Through detect_collision function, it is checked if there is collision between the line connecting q_add, q_near and the object. This is done defining the obstacle as a polygon: firstly verteces of the two unified rectangles are defined independently, then poly = union(poly1,poly2) is used to create the overall polygon. Finally, intersect function is used to return the value of intersecting points between this polygon and the line connecting the above defined nodes. If the sum of in is equal to 0, this means that no collision has occurred and exit_flag is set to 0; else, exit_flag = 1.
If detect_collision is true, the continue command is used to skip to following lines in the for loop and skip to the next iteration.

2.3.2 Else

2.3.2.1 G.add.vertex(q_{add})
2.3.2.2 G.add.edge(q_{add} , q_{near})

In the code, as you may notice, the Graph declaration is slightly different and these two steps are done at the same time simply concatenating nodes array

with the new q_add. Vertex corresponds to q_add.coord and Edge corresponds to q_add.cost. In fact, immediately after the updating of the tree with the new node, a line (the edge) between q_add and q_near is drawn in blue on the grid.

2.4 $q^* \leftarrow$ Nearest_Vertex(G, q_goal)

In this step, exactly in the same way as done in point 2.2, the node in the tree closer to the q_goal is computed and the distance between them is evaluated. q_near_star is equal to nodes(idx), where idx is as before the index returned by min function corresponding to the index of the closest node in the nodes set.

2.5 If Distance(q^*, q_{goal}) < threshold

This is evaluated with the dist() function

2.5.1 FoundPass ← shortestPathTree(G, q_{init} , q^*)

This is the final part of the algorithm: if the above condition is verified, it means that algorithm has added to the tree a node which is closer enough to our goal. This means that algorithm has found a path! It is printed on the video that a Path have been founded and then two path are displayed on the grid: one (in red) is the shortest path required by the algorithm: it is the path connecting the q_goal to q_start that follows the nodes. In the code, this is done defining the q_goal.parent value as the idx value of q_near_star. Then, with a while loop that ends only when the q_end.parent is equal to zero (that means, when current node in the loop is the starting node), the path is plotted from the end q_goal to the initial q_start.

An additional path is also plotted: in green, it is represented the “absolute shortest path”. This path, considering all the nodes in the tree, represent the path with minimum distance within the tree, so it is different from the previous one because instead of considering only the nodes belonging to the connecting path, it considers all the nodes added since the time the path is discovered. This is done in a similar way as the previous path, but in this case in every iteration of the while loop (since the final node is the q_start), the parent starting from the q_near star is updated with the “assign_parent function”.

The assign_parent function firstly computes a set called “q_nearest” where all the nodes where no collision happens between them and the current node are included. Then, distances between the current node and all the points in this set are computed and the node where the cost between it and the current node is minimum is computed. Finally, the index value of this node in the nodes dataset is returned in order to enable the updating of the parent. The line between this new parent that minimizes the cost and the current node is drawn in green.

Finally, with the break command, the algorithm stops.

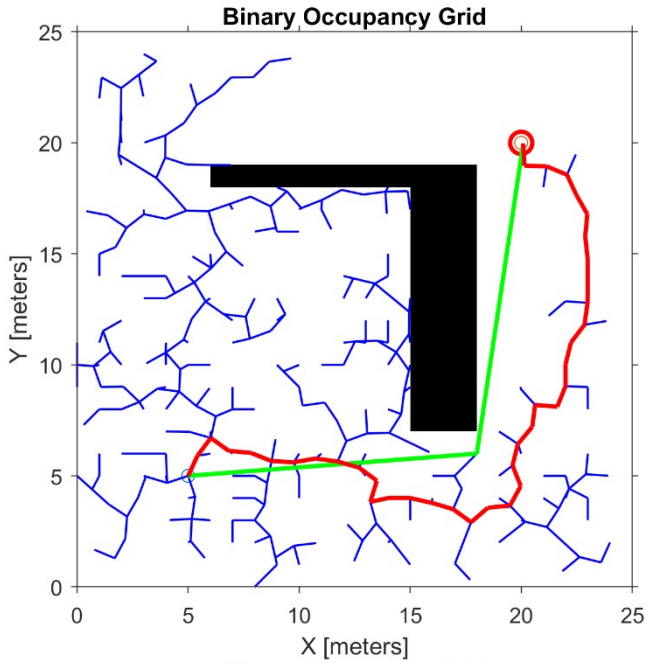
However, at the end of the for loop, if the iterations counter is equal to K, it returns the info that maybe the number of iterations is not sufficiently big and so it must be augmented.

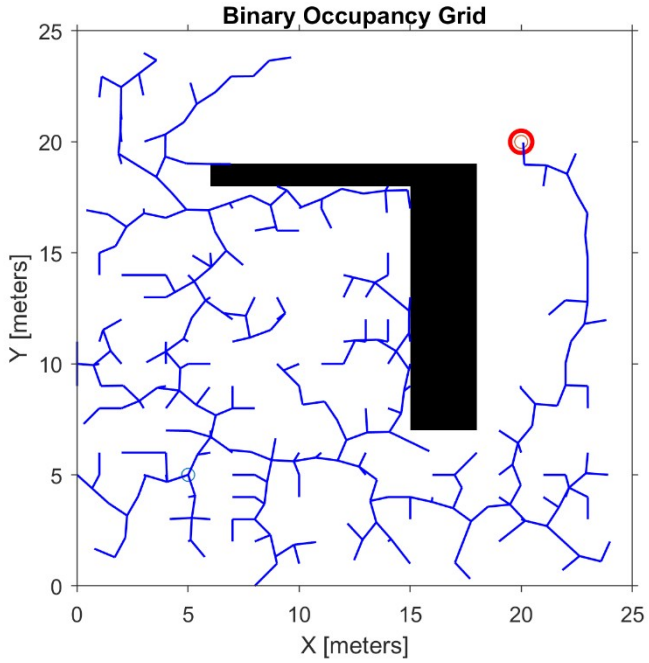
FINAL CONSIDERATIONS AND ANALYSIS OF RESULTS

First of all, the implemented code works fine because at almost every iterations it is able to return a path between q_start and q_goal. However, as already anticipated in the Introduction section, this implementation is a very basic example of RRT. Further improvement can be done: for example, it is possible not to break the loop when a path is discovered, but continue for the whole number of iterations and at the end return the path with minimum cost, or also we can considered additional mechanism to make the node research not completely random but directed towards the goal etc.

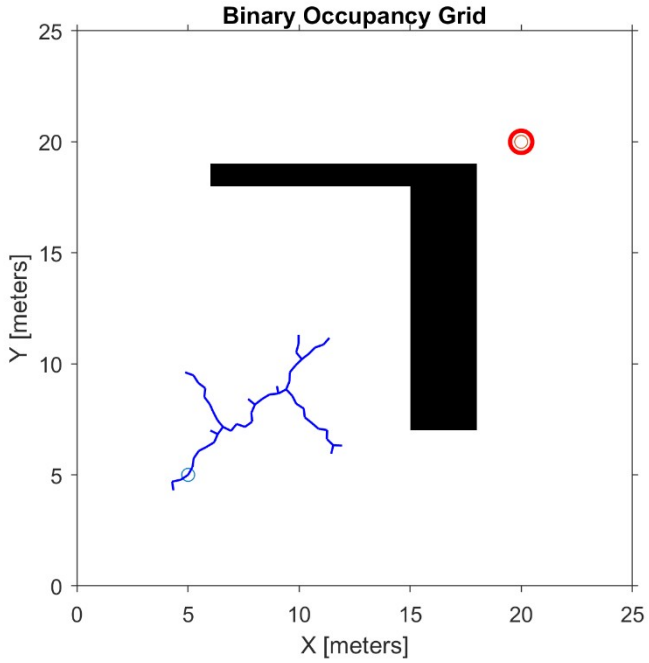
Here are two examples of the code results for a StepSize of 1 and 0.4:

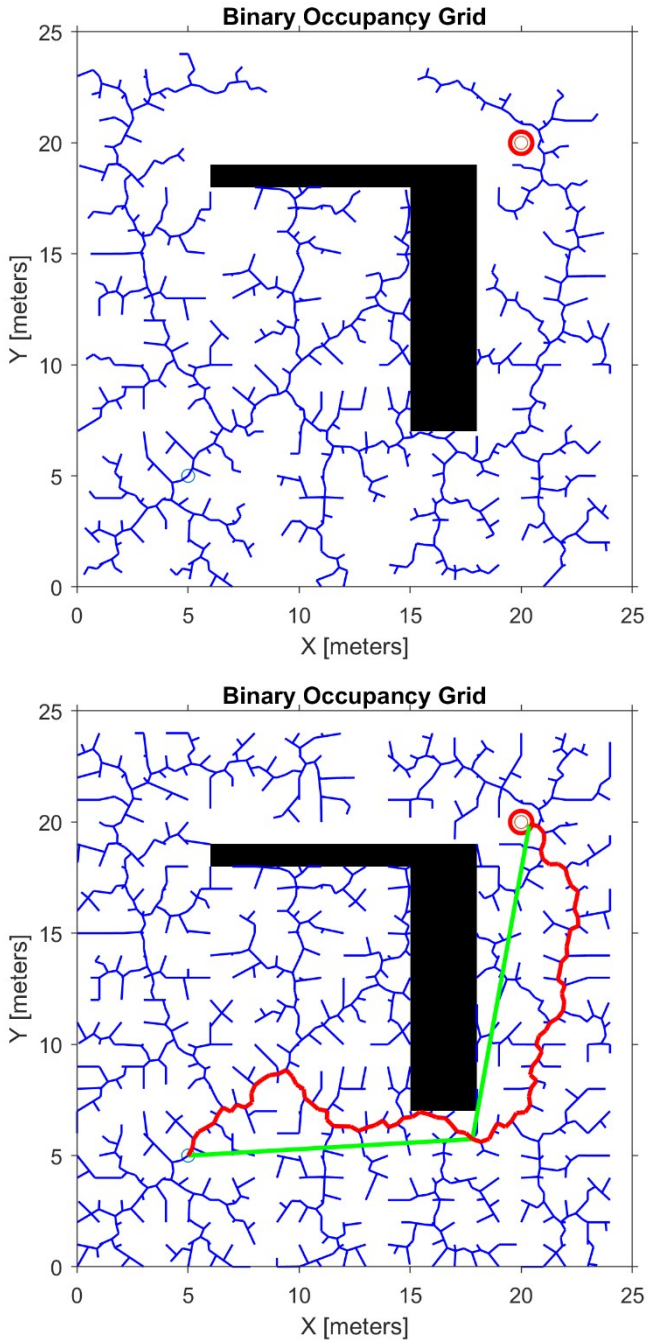
STEP SIZE: 1





STEP SIZE: 0.4





You can immediately noticed that reducing the step size the density of the tree with respect to the same area increases a lot. This means additional computational effort because increasing the number of nodes, all the operations within the code that require the computation of all the distances exponentially increases. However, we can say that with the same threshold, the probability to find a point within this threshold increases reducing the Step Size.

Even with this basic code, results are very positive, but some considerations need to be done:

1) Since the algorithm is random-based, the time required to find the path strongly varies. It can be vary low or quite high, but code has been run man times and for both Step Size of 1 and 0.4, the path is found within the 3000 iterations.
2) Step Size affects the computational effort and a too low value can lead to fail of the algorithm
3) Threshold value need to be chosen wisely: a too big value leads to poor results because path is not very close to the goal. Too low value increments the number of iterations required to find the path and also decreases the probability to find a node within the threshold. Here, threshold of 0.5 is quite good with a Step Size of 1 and 0.4. In general, a good compromise is to have threshold of the same order of the step size.

To conclude, here is an example of the command window results:

Step Size = 1

```
Found Path!
Iteration:930
```

Step Size = 0.4

```
Found Path!
Iteration:2991
```

These results show that iterations required for the StepSize of 0.4 is way bigger than the number of iterations required for the 1 StepSize. This confirms what explained above.