Report for Assignment 3

April 9, 2020

1 Implementation

Based on the code given, we implement the path planning simulation domain (2D map navigation) in **createDomain.py**. We treat the domain as an object which consists of its dimension, obstacles, goal and start. When creating a domain instance, we specify the number of obstacles and the maximum size of the obstacles, which helps facilitate our experiment in part 4. We also write some codes to help visualize our result by using matplotlib.pyplot. This function, whose name is **drawDomain**, receives a path (a list of tuples) and draws the path within the domain. Optionally, it can take a list of the nodes generated by the state space partitioning.

1.1 Quadtree and FBSP Decomposition

We first created a Quadrant object in **quad.py**. This object represents a quadrant. It is capable of returning if it is entirely an obstacle, entirely empty space or mixed. It is also capable of spliting itself into four smaller Quadrant object.

Then we simply implement Quadtree Decomposition by setting the first quadrant to as the entire domain, and recursivly splitting it down into smaller objects until we are left with no mixed quadrants. This is done in **decompose.py**.

Then we implement a State object, in **decompose.py**, that takes a list of full Quadrants and a list of empty Quadrants. It generates a set of nodes for our searching algorithm to search through. Instead of having a node at the center of each quadrant, we have a node at the center of each wall shared by two quadrants. This eliminates the algorithms tendency to find paths that go straight over corners.

Finally, a basic A* search is implmented in **a_star.py**.

For FBSP Decomposition, we use scikit-learn's decision tree classifier with entropy as a loss function. We train a tree on each of the coordinates as the feature and whether its empty or not as the label. We then convert the decision boundry at each leaf of the tree into our Quadrant object. This is done in **decompose.py**. After this we can procceed exacty as we did for QTD.

1.2 Rapidly Exploring Random Tree (RRT)

We first implement some additional data structures (Node and Tree) in **Structure.py**. Node is a subclass of obstacle but it has one more instance: parent. Each tree consists of a list of nodes and its root. Each tree has a function called **findNearestNode**, it receives a node in the domain and outputs the nearest node in the tree.

We then implement the algorithm in **RRTsolver.py**. A solver receives a problem instance (a domain) and outputs the solution and stats. At each iteration, **solve** (the algorithm) generates a random node with some probability of choosing the goal(**randomNode**). Then it attempts to find a new node on the line connecting the random node and the nearest node, determined by the step size. Finally, it connect this new node to the nearest node in the tree (**extend**). When the new node is extremely close to the goal, the algorithm connects the new node and the goal directly and the problem is solved and **traceBack** is triggered. It takes advantage of the data structure and outputs a whole path from the goal to the start by calling a node's parent repeatedly.

2 Results

2.1 Quadtree vs FBSP Decomposition

Below are figures of the solution found for different domain types. The first row is the domain, the second is the QTD's solution and the third is the FBSP's solution.

10 obstacles with length,width$\sim$ uniform$(10, 20)$

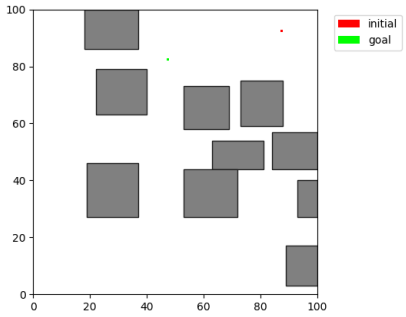

Figure 1: Domain

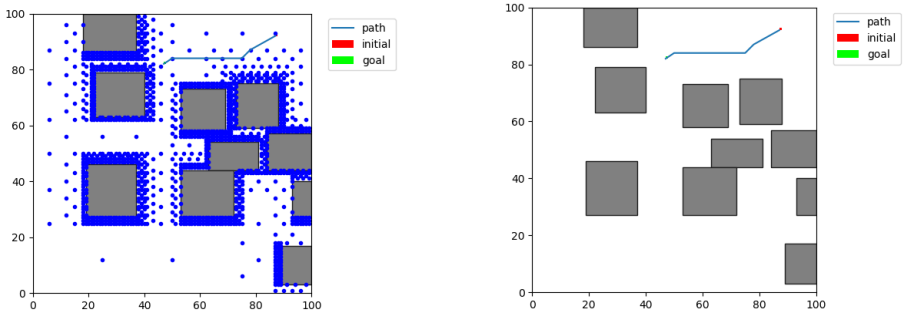

Figure 2: QTD

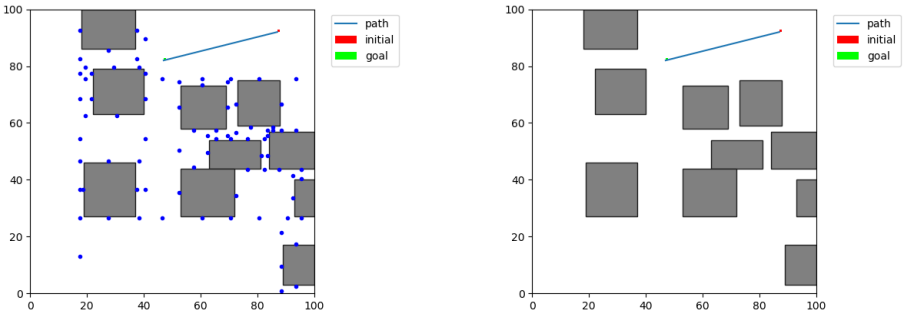

Figure 3: FBSP

10 obstacles with length,width$\sim$ uniform$(10, 50)$

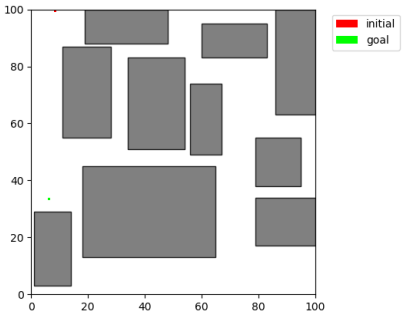

Figure 4: Domain

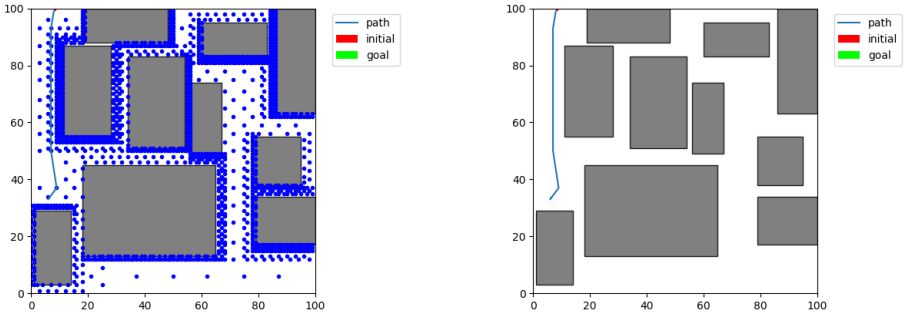

Figure 5: QTD

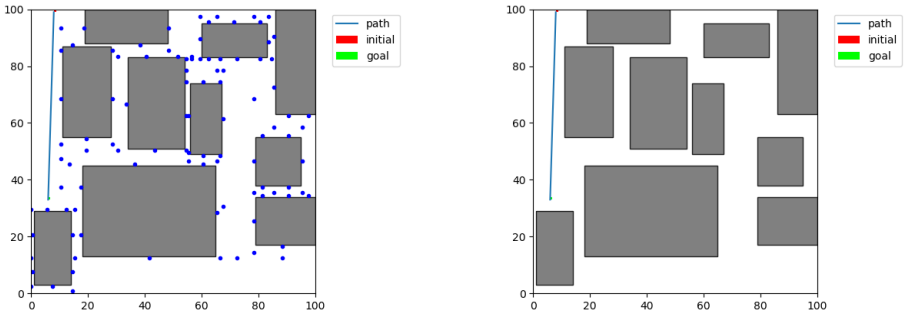

Figure 6: FBSP

20 obstacles with length,width$\sim$ uniform$(10, 20)$

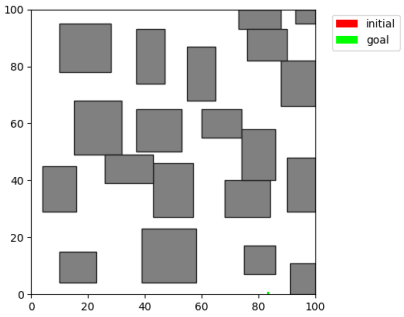

Figure 7: Domain

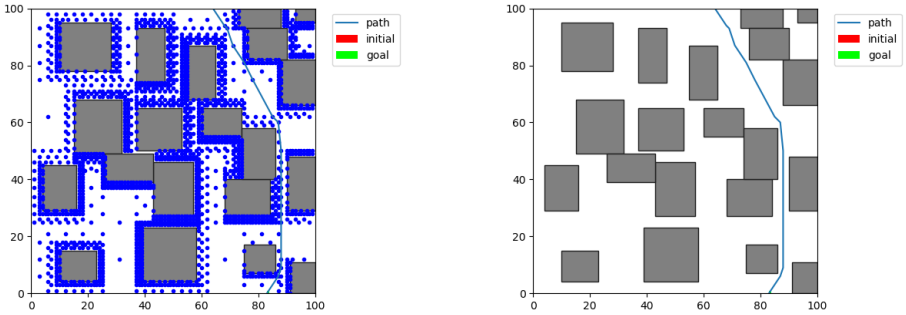

Figure 8: QTD

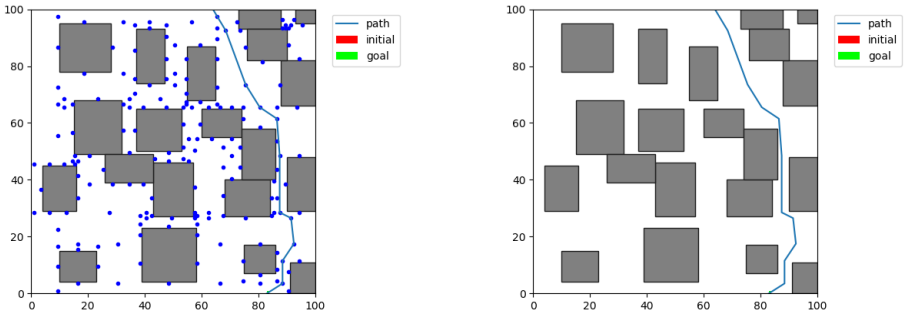

Figure 9: FBSP

20 obstacles with length,width$\sim$ uniform$(10, 50)$

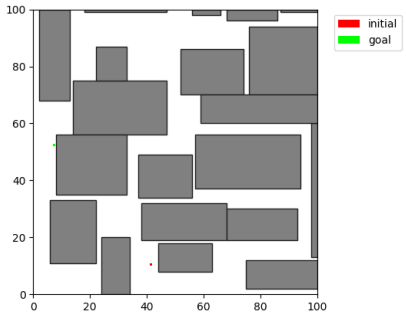

Figure 10: Domain

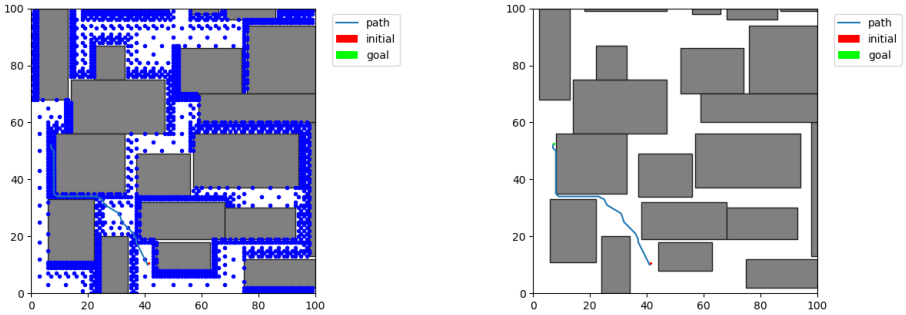

Figure 11: QTD

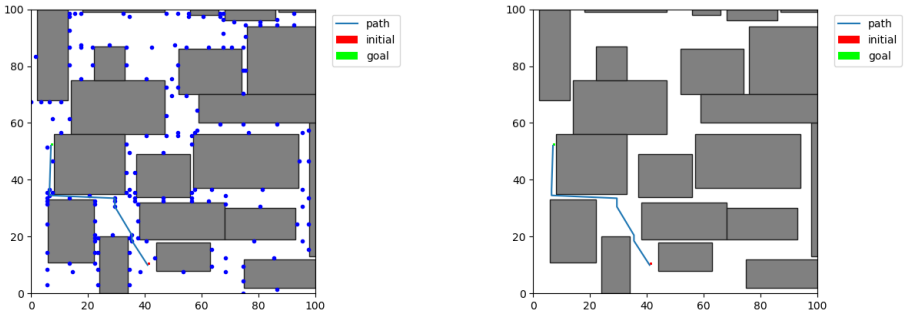

Figure 12: FBSP

30 obstacles with length,width$\sim$ uniform$(10, 20)$

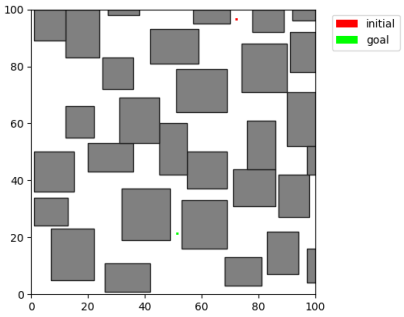

Figure 13: Domain

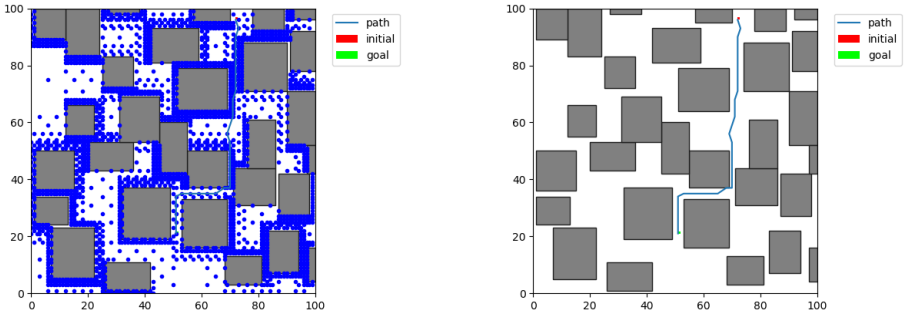

Figure 14: QTD

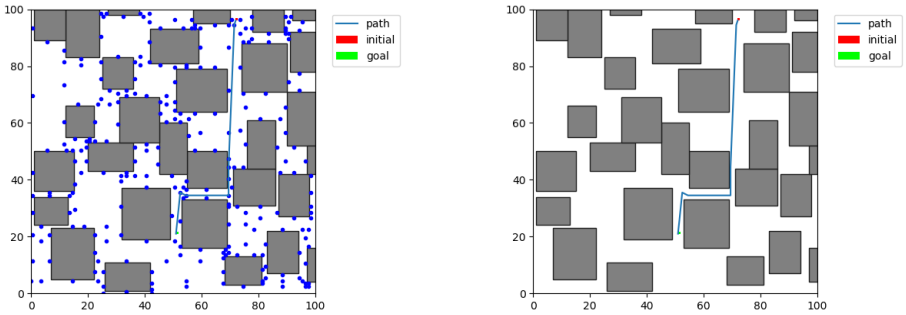

Figure 15: FBSP

40 obstacles with length,width$\sim$ uniform$(10, 20)$

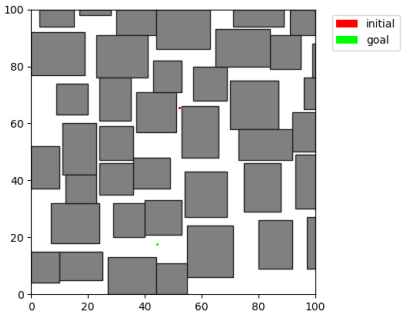

Figure 16: Domain

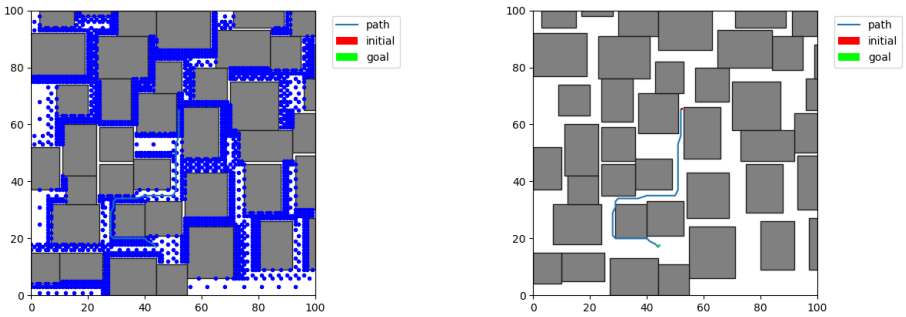

Figure 17: QTD

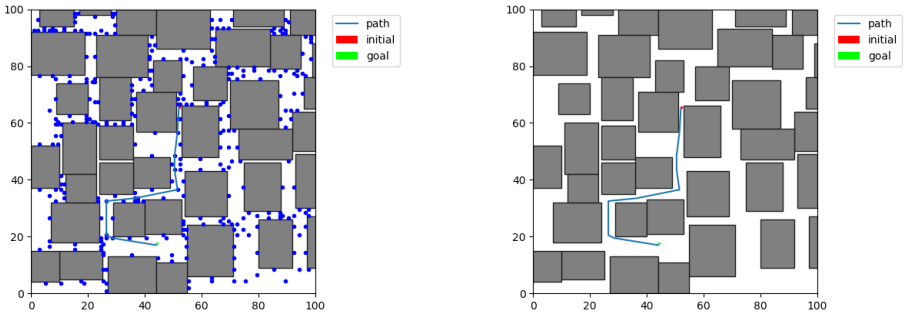

Figure 18: FBSP

We ran the algorithm ten times on each domain type. The average results are in the table below:

QTD					
number of obstacles/maximum size	total time	partition time	search time	total num nodes	num nodes in path
10/20	2.23	2.22	0.01	1134.10	17.20
10/50	3.05	2.97	0.08	1303.90	26.60
20/20	5.61	5.57	0.04	1809.50	27.70
20/50	5.42	5.22	0.20	1655.70	40.10
30/20	10.20	9.98	0.23	2252.60	36.20
40/20	9.27	8.99	0.28	2140.90	53.50

FBSP					
number of obstacles/maximum size	total time	partition time	search time	total num nodes	num nodes in path
10/20	0.04	0.04	0.00	98.10	5.30
10/50	0.04	0.04	0.00	83.50	5.40
20/20	0.11	0.11	0.00	193.40	8.50
20/50	0.09	0.09	0.00	155.30	8.00
30/20	0.26	0.25	0.00	278.50	10.30
40/20	0.29	0.29	0.01	303.10	13.20

From these tables, several interpretations can be made:

- The size of the obstacles does not have much effect on either algorithm
- The number of obstacles has a large negative impact on the time for both algorithms
- For both algorims, the time it takes to find a solution is dominated by the time it takes to partition the state space
- Both the partition speed and the search speed is significantly faster for FBSP
- As expected, FBSP generates far fewer nodes

2.2 Rapidly Exploring Random Tree (RRT)

Below are figures of the solution found for different domain types.

When the number of obstacles is 10:

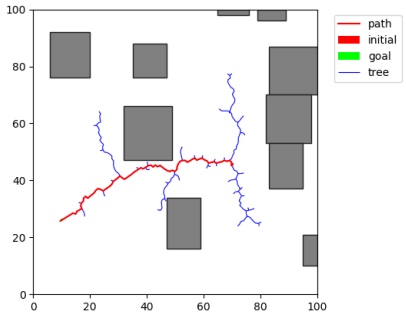

Figure 19: 10 obstacles, length,width$\sim$ uniform$(10, 20)$

When the number of obstacles is 10:

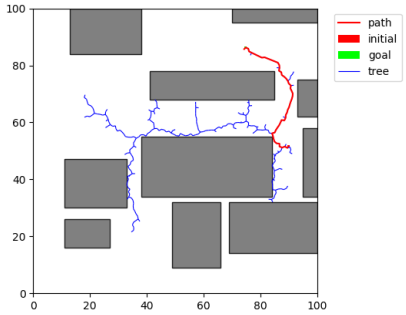

Figure 20: 10 obstacles, length,width$\sim$ uniform$(10, 50)$

When the number of obstacles is 20:

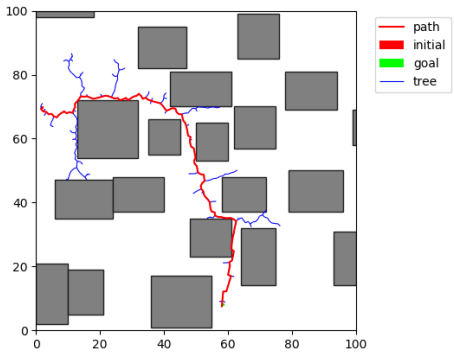

Figure 21: 20 obstacles, length,width$\sim$ uniform$(10, 20)$

When the number of obstacles is 20:

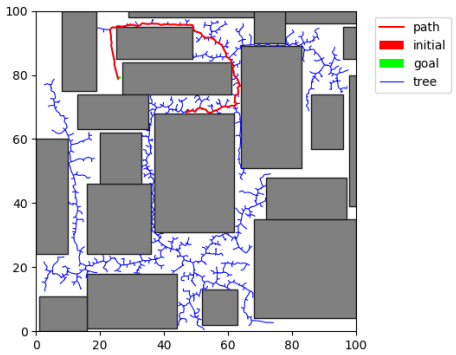

Figure 22: 20 obstacles, length,width$\sim$ uniform$(10, 50)$

When the number of obstacles is 30:

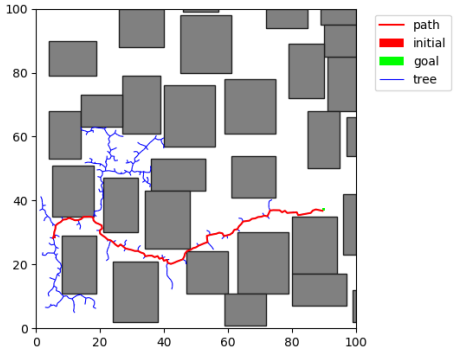

Figure 23: 30 obstacles

When the number of obstacles is 40:

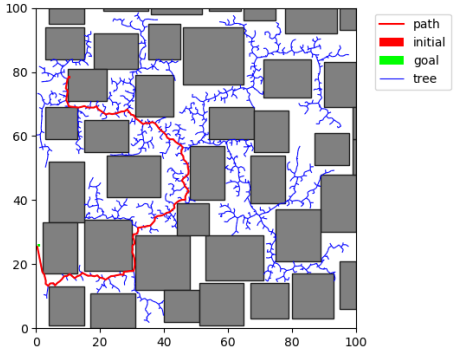

Figure 24: 40 obstacles

When the number of obstacles is 45:

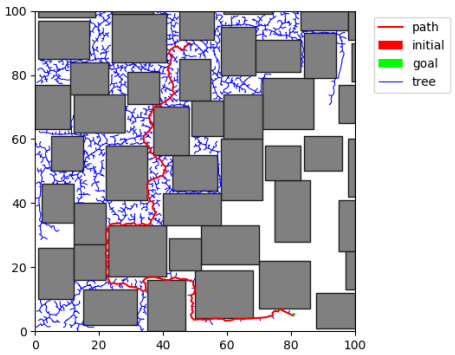

Figure 25: 45 obstacles

When the number of obstacles is 45:

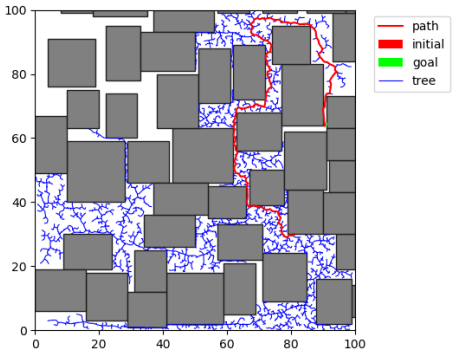

Figure 26: 45 obstacles

We ran the algorithm ten times on each domain type. The average results are in the table below:

Experiment			
number of obstacles/maximum size	run time(avg)	num of nodes in the path(avg)	num of nodes in the tree(avg)
10/20	0.17	45.25	169.75
10/50	0.55425	87.25	252.25
20/20	0.8525	91.5	333
20/50	5.979	98.25	795
30/20	1.789	115.75	414
40/20	20.715	136.75	1459
45/20	57.59	167.67	2511.33

In summary, when the maximum size of obstacle is fixed, the run time and the number of nodes in the tree increase exponentially as the number of obstacles

increases. The number of nodes in the path increases as the number of obstacles increases, but it is impacted less than the run time. On the other hand, when the number of obstacles is fixed, if we change the maximum size of obstacles, the rum time and the number of nodes in the tree increase in general. Furthermore, if the number of obstacles is already large, then changing the maximum size will bring less impact to the behavior of the algorithm.