Project 1 - Bug Algorithms and State Space Search

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. Bug Algorithms

A. Running Bug 0

Question: The path length in the Hook Grid example in Fig. 1 is very long and does not reach the goal. Briefly describe why this has occurred.

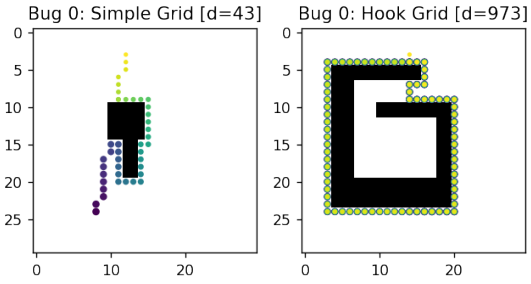

Fig. 1: Results of Bug 0

Answer: In the Hook Grid, the path is long and does not reach the goal. That is because in the $follow\_object()$ function, if there is a wall ahead, the robot will turn out all the time. For this type of obstacle and the starting point, the robots will go around the obstacle and cannot reach its goal.

B. Bug 1

In Bug 1 Algorithm, we asked the agent follows the entire boundary of the obstacles, and determine the point on the boundary which is closest to the goal. This algorithm does not provide the optimal paths, it can determine, however, path to the goals in the map: Simple Grid, Hook Grid, and Maze 0 (Fig. 2). In Maze 1, the algorithm does not terminate and runs in a infinity loop.

Part of Bug 1's algorithm is shown in Fig. 3: *looping_back* and *else* part. In the *looping_back* part, there is a *if* statements to terminate the follow obstacles if the robot gets to the closest point.

The *else* part moves the robot around the obstacle to find the closest point. As the agent reaches the starting point, this part will be terminated, and the algorithm goes to the *loop_back* part. The code in line $47-48$ used to terminate the algorithm (*early termination*) if there is not a path reaching the goal (the closest point is also the starting point).

C. Bug 2 Algorithm

Bug 2 algorithm combines the **m-line** concepts. It is different to Bug 1 is that it does not follow the whole obstacle boundary but until meeting the *m-line* again. At that point, the algorithm checks several conditions to decide whether it continues following the boundary or follow the *m-line*. The conditions are: The new intersection point must be different to the previous one, and it is also closer to the previous one.

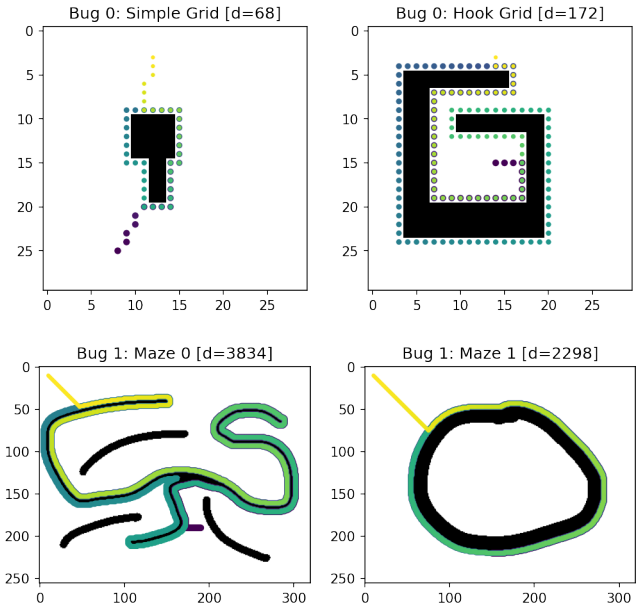

Fig. 2: Results of Bug 1

```
31 elif looping_back:
32     robot.follow_object(grid)
33     if (robot.position == obstacle_closest_position).all():
34         line = bresenham_points(robot.position, goal)
35         looping_back = False
36         follow_line = True
37 else:
38     robot.follow_object(grid)
39     shortDist = np.abs(obstacle_closest_position[0]-goal[0])+
40                 np.abs(obstacle_closest_position[1]-goal[1])
41     currDist = np.abs(goal[0] - robot.position[0])
42                + np.abs(goal[1] - robot.position[1])
43     if (currDist < shortDist):
44         obstacle_closest_position = robot.position.copy()
45     if (robot.position == obstacle_start_position).all():
46         looping_back = True
47         if (obstacle_start_position == obstacle_closest_position).all():
48             break
```

Fig. 3: The code in the looping_back part

Comparing to Bug 1, Bug 2 outperformed on Simple Grid, Hook Grid, and Maze 0. In my opinion, if obstacles are a set of long simple shapes, Bug 2 will win because it saves a lot of time not running around the whole obstacles boundary. If

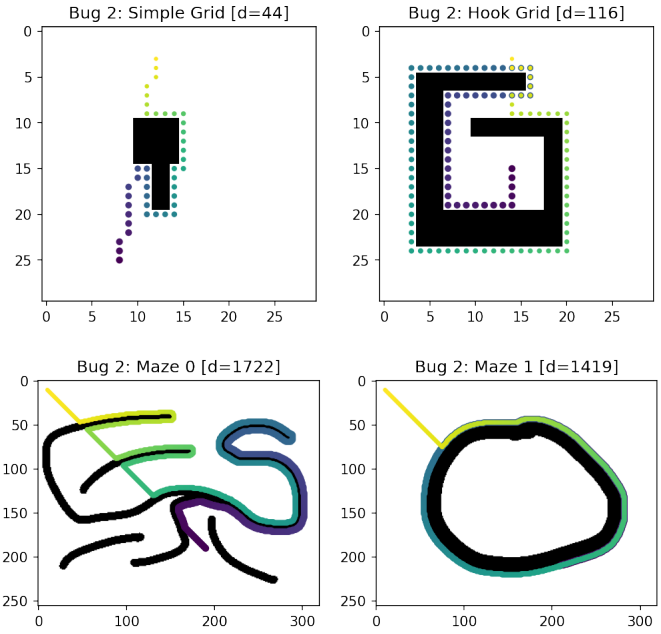

Fig. 4: Results of Bug 2

there is only 1 or 2 obstacles, and they are in complicated shape, Bug 1 will win because it need only 1 time to go around the obstacle, find the closest point, and head to the goal. To prove that, we generated two custom maps on the next section and let both algorithms run on them.

D. Custom maps

On a custom map B2, there are multiple obstacles with simple shape. The algorithm Bug 2 outperforms bug 1 because it does not run around all the obstacles on the way to the goal.

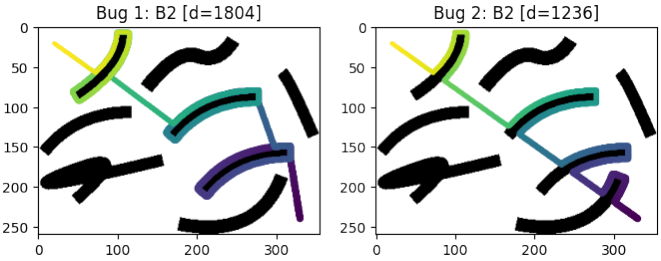

Fig. 5: On map B2: Bug 2 outperform Bug 1

In the case of only on obstacle which has a complex shape, and the goal is inside the map, Bug 1 algorithm has chance to outperform the Bug 2. If the goal is selected closer to one part of the obstacle than from the boundary, it makes Bug 2 moving around mostly all obstacle's boundary two times, and take much more steps for Bug 2 to find the target. (Fig. 6)

II. Stage Space Search - Sliding Puzzles

A. Implement BFS and DFS

We have implemented the two algorithms: **BFS** and **DFS**, and the code for **BFS** is shown in Fig. 7.

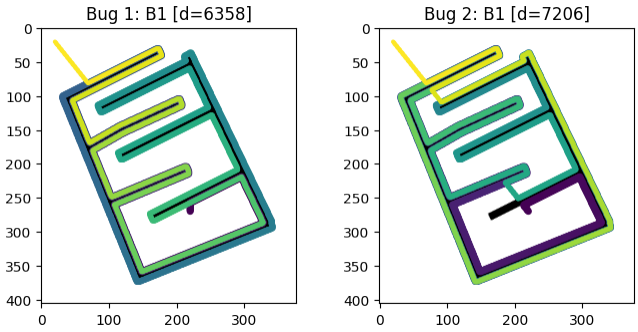

Fig. 6: On map B1: Bug 1 outperform Bug 2

```
5  def breadth_first_search(start, goal, max_iterations=100000):
6      Q = [[start]]
7      visited = set([start])
8      stime = time.time()
9      for ind in range(max_iterations):
10         N = Q.pop(0)
11         # Check if the goal has been reached
12         if N[-1] == goal:
13             return {
14                 'succeeded': True,
15                 'path': N,
16                 'num_iterations': ind,
17                 'path_len': len(N),
18                 'num_visited': len(visited),
19                 'time': time.time() - stime,
20             }
21         last_state = N[-1]
22         for ii, child_state in enumerate(last_state.get_children()):
23             if (child_state not in visited):
24                 visited.add(child_state)
25                 Q.append(N+[child_state])
26     return {'succeeded': False}
```

Fig. 7: Breath First Search's Code

The algorithm works as following. We initialized a list of discovering path Q, and a set of visited node *visited*. We will run a for loop with predefined maximum number of iteration. In the loop, we get the path at the first position in the list Q, and also remove that path from Q by command *pop*. If the last state in the path is not *goal*, we go through all its children state by function *get_children*. A second for loop runs over all the children nodes to check whether the child is in the visit set. If not, we will add that child to the path list and the *visited* set. Then, we add the modified path to the end of the path list Q, and repeat the outer for loop.

For **DFS** algorithm, the difference is only popping out the last path in Q by $pop(-1)$ instead of the first path (in line code 10 - Fig. 7).

Using **BFS** and **DFS** to run on 4x2 grid with 3 different seeds: 695, 710, and 111, the number of iteration and Path length are shown in Table I. The number of iteration and path's length for **DFS** is much longer than **BFS**. There could be 2 explanation for that:

1) the solutions are the neighbor nodes, so **BFS** finishes its work much faster, while **DFS** prioritize discovering the children nodes, which could lead to explore the whole search space.
2) Because the 4x2 grid has maximum 3 children, so the branching factor is less than or equal 3, so the BFS can take the advantage and exploring faster.

Question: If I were to ask you to find the optimal path to

	BFS			DFS		
seed	695	710	111	695	710	111
No. of Iter.	17	588	951	8	6667	3402
Path Length	5	13	15	9	6003	3139

TABLE I: BFS and DFS data

solve a 3x3 sliding puzzle, which algorithm should you use? Why?

Answer: The branching factor is 4, so DFS could do better than BFS.

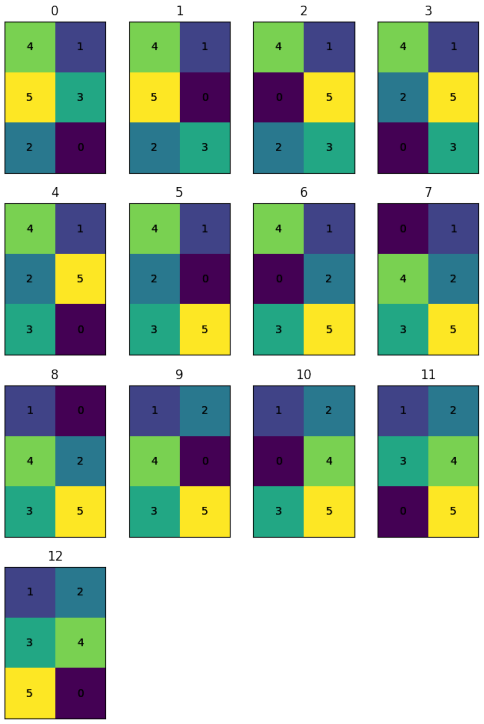

Fig. 8: Results of Breath First Search

The paths for the 3x2 grid are shown in Fig. 8 and 9. For the same initial state, BDS needs a path of 13 states to reach the goal, meanwhile DFS requires 71 state's path to reach the same goal. It is fit with the theoretical calculation in the lecture.

B. A^* Search

In the A^* search algorithm, each time discovering a children node, we need to check whether the node is in the $visited$ set. If no, we calculate the cost (distance), heuristic (line 31,32 - in Fig. 10), then add the node into the path. The path then is added back to the path list Q.

Question: The code block to check your A* star uses a heuristic function I have provided you with, $sliding\_puzzle\_zero\_heuristic$, that returns 0 no matter the inputs. Is this an admissible heuristic? Explain why or why not?

Answer: the heuristic function is admissible, because it always underestimates the cost to the goal from the current state (always $= 0$). But it is not a good one because it is not guiding the next state to be expanded.

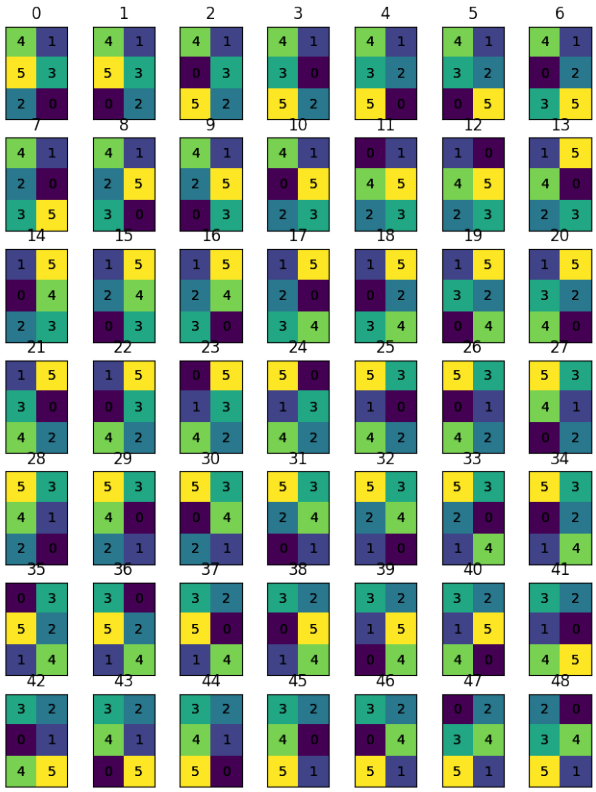

Fig. 9: Results of Depth First Search

```
14 for ind in range(max_iterations):
15     N = Q.pop(0)
16     distN, hN, statesN = N
17     # Check if the goal has been reached
18     if statesN[-1] == goal:
19         return {
20             'succeeded': True,
21             'path': statesN,
22             'num_iterations': ind,
23             'path_len': len(statesN),
24             'num_visited': len(visited),
25             'time': time.time() - stime,
26         }
27     last_state = statesN[-1]
28     for ii, child_state in enumerate(last_state.get_children()):
29         if (child_state not in visited):
30             visited.add(child_state)
31             newDist = distN+1
32             newhN = heuristic_fn(child_state, goal)
33             Q.add((newDist, newhN, statesN+[child_state]))
34 return {'succeeded': False}
```

Fig. 10: Parts of A^* Search algorithm

C. Heuristics for the Sliding Puzzle

The two heuristics $sliding\_puzzle\_heuristic\_num\_incorrect$ and $sliding\_puzzle\_heuristic\_manhattan$ functions are shown in Fig. 11.

In $sliding\_puzzle\_heuristic\_num\_incorrect$ function, we run through the state array, and if a number is positioned at a different position in the goal state, we increase the number of incorrect by 1.

In $sliding\_puzzle\_heuristic\_manhattan$ function, we use the module and % operators to get the positions of the numbers in the grid and sum all the difference to the positions of the corresponding numbers in the goal's state.

Question: Which of your two heuristics reaches the goal

```python
def sliding_puzzle_heuristic_num_incorrect(start, goal):
    incorrectNr = 0
    for ii, index in enumerate(start.state):
        if not (start.state[ii] == goal.state[ii]):
            incorrectNr +=1
    return incorrectNr


def sliding_puzzle_heuristic_manhattan(start, goal):
    dist = 0
    for ii, tile in enumerate(start.state):
        m = ii//start.ncols
        r = ii % start.ncols
        index2 = np.where(goal.state==tile)[0][0]
        m1 = index2//start.ncols
        r1 = index2 % start.ncols
        dist += np.abs(m-m1) + np.abs(r-r1)
    return dist
```

Fig. 11: Two heuristics functions

```
Starting puzzle:
[[7 1]
 [2 4]
 [6 5]
 [3 0]]

A* No Heuristic Time: 0.1798088550567627
A* No Heuristic Path Len: 21
A* No Heuristic NumIterations: 4594
A* Num Incorrect Heuristic Time: 0.04881644248962402
A* Num Incorrect Heuristic Path Len: 21
A* Num Incorrect Heuristic NumIterations: 1115
A* Manhattan Distance Heuristic Time: 0.023136138916015625
A* Manhattan Distance Heuristic Path Len: 21
A* Manhattan Distance Heuristic NumIterations: 255
A* Num Incorrect Squared Heuristic Time: 0.005084991455078125
A* Num Incorrect Squared Heuristic Path Len: 23
A* Num Incorrect Squared Heuristic NumIterations: 120
A* Manhattan Squared Heuristic Time: 0.009068727493286133
A* Manhattan Squared Heuristic Path Len: 31
A* Manhattan Squared Heuristic NumIterations: 101
```

Fig. 12: The results with 4 heuristics

in fewer iterations. Will this be true (or at least close enough) for any puzzle? Explain why

Answer: The Manhattan distance heuristic provides a better result with little iterations. The explanation is minimal cost at which the planner need to move all the tile to the goal state. With the distance, cost of a state can be varied in a wide range, so we can be easy to compare and select the best one. Meanwhile, the heuristic with number of incorrect is not "detail" enough, so multiple states can have the same heuristic value, and trying all the same value states could take more run. I think Manhattan distance can be true for all puzzles.

Question: Which of the four heuristics are admissible? What happens when you use a non-admissible heuristic to plan? How do the path lengths compare between the admissible and the non-admissible heuristics? How do the number of iterations compare between your heuristics and the "squared" versions I provided you with? Why does this happen?

Answer: In all four heuristics, Manhattan distance and number of incorrect ones are admissible because they will never overestimate the number of slide to reach the goal state, and the paths are optimal. Meanwhile, the "squared" heuristics are not admissible because they could overestimate the heuristic values. The non-admissible heuristics work as greedy approach, and their advantage is the number of iteration is smaller than admissible ones. However, the cons are they usually providing non-optimal solutions.

D. Statistical Analysis

We have tested the A^* Search with 5 heuristics with 100 seeds on the 2x2, 3x2, and 4x2 grids, and the result as in Table II.

Grid	2x2		3x2		4x2	
	Mean	Std	Mean	Std	Mean	Std
Zero heuristic						
Len.	3.90	2.14	11.90	4.80	16.06	5.63
Iter.	6.06	4.29	148.31	108.19	3119.77	3735.40
Incorrect Number heuristic						
Len.	3.90	2.14	11.96	4.87	16.12	5.68
Iter.	4.48	4.05	70.06	72.08	833.23	1234.94
Manhattan Dist. Heuristic						
Len.	3.90	2.14	12.10	4.99	16.30	5.81
Iter.	3.70	3.24	55.19	53.08	312.43	458.04
Incorrect Number Square heuristic						
Len.	3.90	2.14	13.22	6.16	20.80	9.96
Iter.	3.90	3.31	69.41	62.93	433.11	572.12
Manhattan Dist. Square Heuristic						
Len.	3.90	2.14	14.12	7.04	23.22	13.05
Iter.	3.04	2.24	34.75	27.60	119.55	144.07

TABLE II: The mean and std of path length and iteration numbers of A* with 4 heuristics

Overall, Manhattan distance heuristic and its square version provide the best results. The former one determines the optimal paths and the latter one works with least iterations.

III. Probabilistic Road Maps

The code to generate a probabilistic Road Map is shown in Fig. 13.

```python
for _ in range(num_sampled_points):
    # Randomly generate points and add them to points
    x = random.uniform(region_x[0], region_x[1])
    y = random.uniform(region_y[0], region_y[1])
    # check whether the point is in obstacles
    if not does_contain_point([x,y], obstacles):
        points.append([x,y])
# Initialize the "edge length graph"
edge_length_matrix = np.zeros((len(points), len(points)))
for ii, point_a in enumerate(points):
    for jj, point_b in enumerate(points[ii+1:]):
        distance = np.linalg.norm(np.array(point_a) - np.array(point_b))
        # reject (continue) if the points are farther than the max distance
        if distance > max_distance:
            continue
        # check the line intersect with obstacles
        if does_line_collide((point_a, point_b), obstacles):
            continue
        edge_length_matrix[ii, jj+ii+1] = distance
        edge_length_matrix[jj+ii+1, ii] = distance
```

Fig. 13: PRM code

We randomly select a point (x, y) in the $region\_x$ and $region\_y$, then check whether the point is in the obstacles (lines 94 to 98, Fig. 13). If the point is in any obstacle, it will be ignored. Next, the edges will be generated by connecting the valid points. There are two conditions for the edges: their length is not over a $max\_distance$ and the edge should not collide with any obstacle (line 105 -111 in Fig. 13).

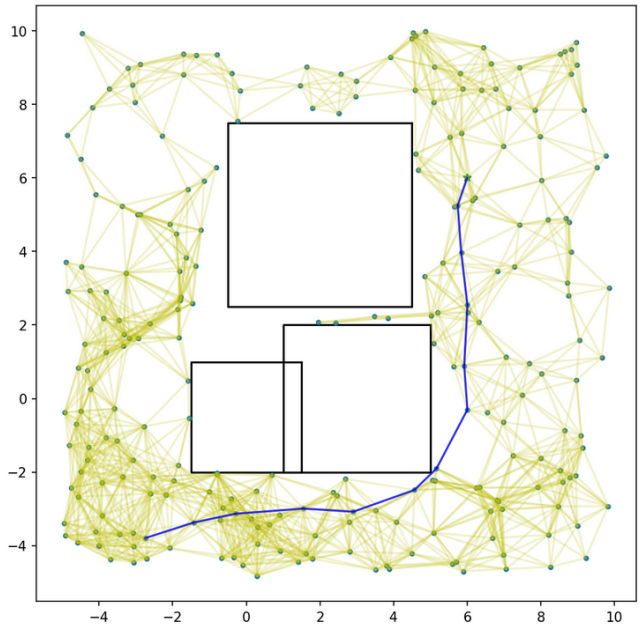

Fig. 14: PRM graph and the found path

After the probabilistic road maps is done, we can add the start and goal into the environments, and use the function $shortest\_path$ to find the shortest path between the start and goal (Fig. 14).

A. Understanding of PRM trends

As number of sample point varies from 100 to 2500, the path length is reduced gradually as shown in Fig. 15 and Table IV

Nr. of Sample Points	path length
100	Fail
200	19.761
300	17.677
400	17.228
500	17.864
600	15.265
700	14.778
800	14.841
900	15.157
1000	14.681
1100	13.997
1200	14.739
1300	14.520
1400	14.463
1500	13.952
1600	14.604
1700	15.525
1800	14.871
1900	15.505
2000	14.168
2100	15.047
2200	14.868
2300	15.244
2400	14.222
2500	14.138

TABLE III: The number of sample points and path lengths

There are some drops in the path lengths, that is because of the random feature and a narrow gap between the obstacles. In the environment, there is a narrow gap between the obstacles, and sometimes the algorithm can not place a random point in the gap. If the shortest path connecting the start and the goal goes through the gap, the connecting path's length will change significantly. As the number of sample points increases, there is more likely a random point will be placed in the gap, so the function can find the near shortest path. However, prm will never ensure the shortest path.

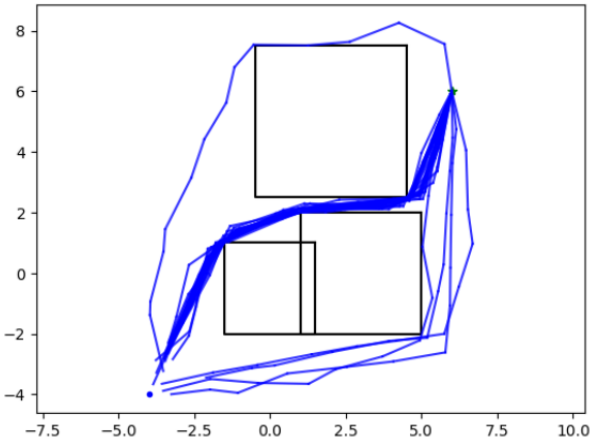

Fig. 15: PRM graph and the found path

B. Custom PRM

We created a map with narrow passages, which make the planner challenging to find a path connecting a start and a goal. To do that, a L-shape obstacle is made, then there are three other obstacles around the L-shape obstacle (Fig. 16). There are at least three narrow gaps between the obstacles, and the shortest path must go through those narrow gaps.

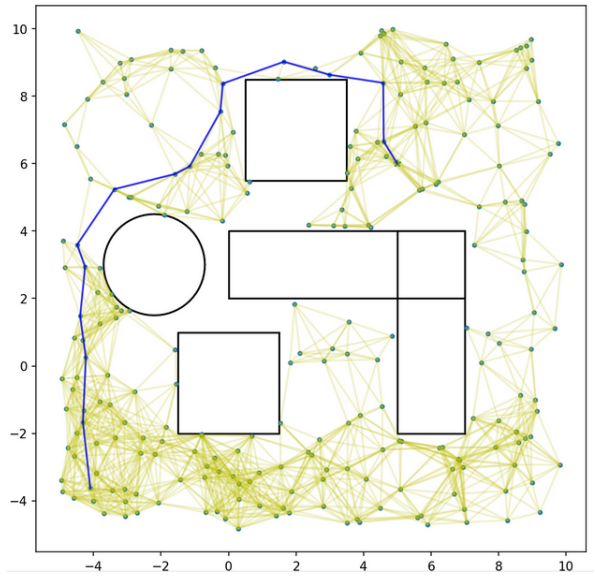

Fig. 16: PRM graph with custom map and path with 300 sample points

We run 10 tests with increasing number of sample points from 300 to 1200. The path lengths are shown in Table IV. Similar to the result in Section III-A, as increasing the

Nr. of Sample Points	path length
300	21.756
400	15.651
500	14.965
600	15.823
700	13.391
800	13.020
900	13.253
1000	12.727
1100	14.241
1200	13.12

TABLE IV: The number of sample points and path lengths

number of sample points, the path length is shorter. With 300 sample points, the prm graph is sparse, and the path length is quite large (21.756) (Fig. 16). As the number of sample points is up to 1200, the prm graph is dense (Fig. 17), and consequently, the length of the path is much shorter (13.120).

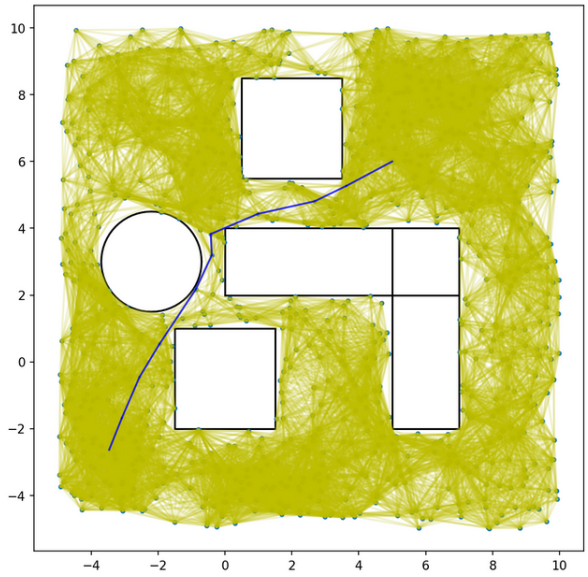

Fig. 17: PRM graph with custom map and path with 1200 sample points

IV. Docker, CGAL, and Voronoi Decompositions

We installed successfully the Docker and CGAL in our local computer and run the provided code of CGAL and Decompositions. The algorithm outputted a diagram as shown in Fig. 18.

The object *CS695* in the diagram is partitioned into complex shapes, which will bring advantage to generate a path in this map for a motion planner.

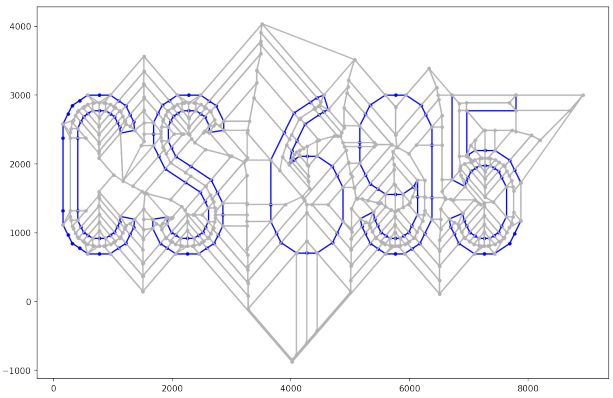

Fig. 18: CGAL - Voronoi diagram