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.

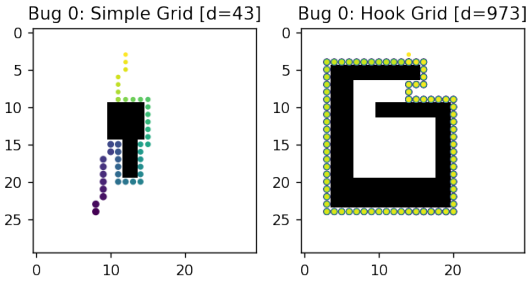

Fig. 1: Results of Bug 0

Answer: In the Hook Grid, the path is long and does not reach the goal. There are two reasons for it.

- It does not remember the visited position
- As the robot meets the line again, it will turn left head to the goal. However, it faces the wall, then it turn around to the opposite direction and follow the line. This will make the robot going back to the visited position.

B. Bug 1

In Bug 1 Algorithm, we asked the agent follow 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.

The codes of Bug 1 are shown in Fig. 3 and 4. We do not show the entire code of the algorithm, but the adjusted code, which consist of the *back_loop* and the *else* part. In the *loop_back* part, there are two *if* statements. One is to terminate the infinity loop if the algorithm find out the starting point is also the closest point. It means the agent runs around the whole obstacle, and comes back the initial point and it is also the closest point. We conclude that we cannot reach to the goal. The second *if* statement controls the robot move back the closest point along the obstacle's boundary. As reaching the closest point, the *loop_back* statement will terminate.

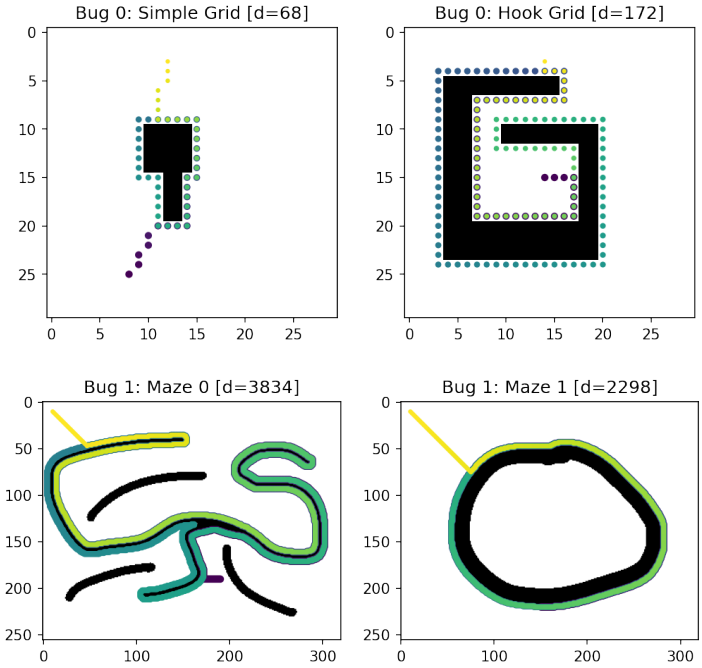

Fig. 2: Results of Bug 1

```
elif looping_back:
    # Follow the object until we reach the closest point
    # and recompute the line.
    if ((obstacle_start_position[0] == obstacle_closest_position[0])
            and (obstacle_start_position[1] == obstacle_closest_position[1])):
        print("The goal is not reachable")
        break
    robot.follow_object(grid)
    if ((robot.position[0] == obstacle_closest_position[0])
            and (robot.position[1] == obstacle_closest_position[1])):
        # When should you stop looping back around the object and follow the line
        line = bresenham_points(robot.position, goal)
        looping_back = False
        follow_line = True
```

Fig. 3: The code in the loop_back part

The *else* part control the robot move around the entire obstacle, and determine the closest point. As the agent reaches the starting point, this part will be terminated, and the algorithm goes to the *loop_back* part.

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

```
else:
    # Follow the object until we loop back to our start position
    # Then set 'looping_back' to True
    robot.follow_object(grid)
    manhattan_dist1 = (np.abs(goal[0] - robot.position[0]) +
                       np.abs(goal[1] - robot.position[1]))
    manhattan_dist2 = (np.abs(goal[0] - obstacle_closest_position[0]) +
                       np.abs(goal[1] - obstacle_closest_position[1]))
    if manhattan_dist1 < manhattan_dist2:
        obstacle_closest_position = robot.position.copy()

    if ((obstacle_start_position[0] == robot.position[0])
            and (obstacle_start_position[1] == robot.position[1])):
        looping_back = True
```

Fig. 4: The code in the *else* part of the for loop

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.

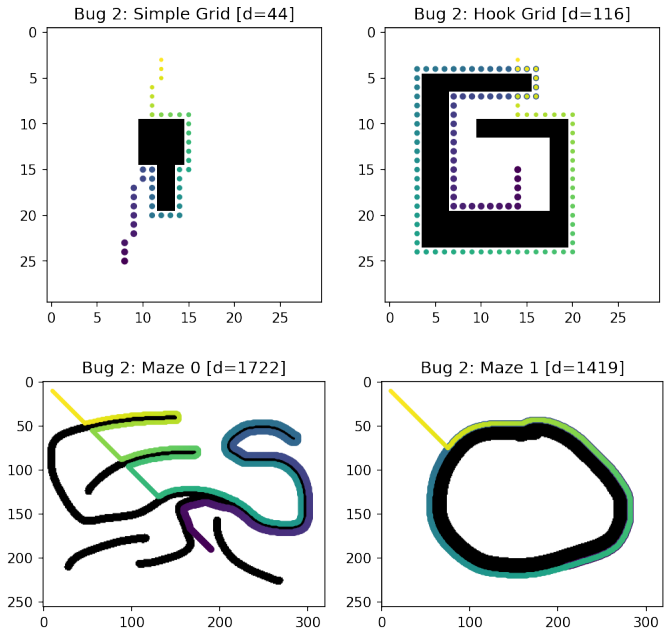

Fig. 5: Results of Bug 2

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 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.

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

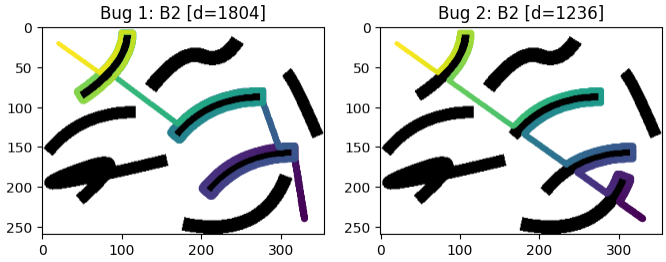

Fig. 6: On map B2: Bug 2 outperform Bug 1

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. 7)

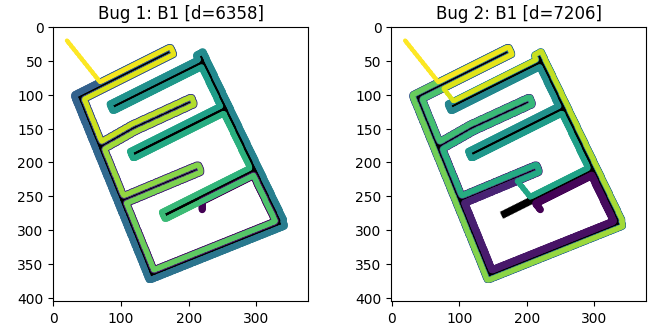

Fig. 7: On map B1: Bug 1 outperform Bug 2

II. Stage Space Search - Sliding Puzzles

A. Expanding States

Question: You should see in both plots that the number of states levels off and asymptotically approaches a constant value and actually reaches that value for the 2x2 grid (and perhaps for the 3x3 grid as well). Why does this occur? What does it mean when the expansion ratio (the right plot) reaches 1?

Answer: The number of states and expansion ratio asymptotically approaches a constant values because:

- The number of state in each grid are limited. After several first expansions, we find a lot of new states and added them into the visited set. As the size of visited set grows large enough, the children states appear in the visited set more frequently. This will reduce the expansion ratio.
- The expansion ratio reached one, it means no longer we find a unvisited state (new states) and we have done the discovering the map.

Question Run expand_states for a 3x2 grid for num_iterations=30; what is the total number of states reached after 30 iterations? How does this value compare to (3 * 2)! = 720? Is this ratio the same for the 2x2 grid and (2 * 2)! = 24?

Answer: As running the search on 3x2 grid with number_iterations=30, the number of state is 360, which is half of (3*2)! = 720. (From the iteration 21, the number of state

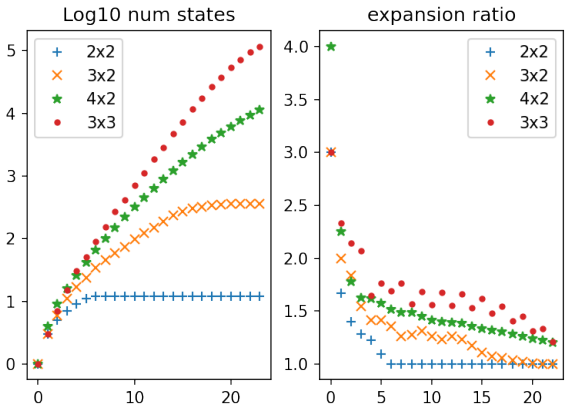

Fig. 8: State expansion and ration of state

remains unchanged). The same ratio is correct for 2x2 grid, whose total states is 12 = half of (2*2)!

B. Implementing Search

We have implemented the two algorithms: **Breath First Search** and **Depth First Search** whose codes are shown in Fig. 9 and 10.

```
def breadth_first_search(start, goal, max_iterations=1
    Q = [[start]]
    visited = set([start])
    stime = time.time()
    for ind in range(max_iterations):
        # First get path from Q and update Q
        path = Q.pop(0)
        state = path[-1]
        children_set = state.get_children()
        for child in children_set:
            path1 = path.copy()
            if not (child in visited):
                path1.append(child)
                visited.add(child)
                # Check if the goal has been reached
                if path1[-1] == goal:
                    return {
                        'succeeded': True,
                        'path': path1,
                        'num_iterations': ind,
                        'path_len': len(path1),
                        'num_visited': len(visited),
                        'time': time.time() - stime,
                    }
                # Then add new paths to Q from N
                Q.append(path1)
    return {'succeeded': False}
```

Fig. 9: Breath First Search's Code

The algorithm works as following. We initialized a list of discover 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*. Then we get the last state in the path list, and use function *get_children* to determine all the children of the node. A second for loop will run over all the children node set 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. After that, we check whether the path reaches the goal. If yes, it return the path and end the search algorithm. If no, we add the new path to the end of the path list Q, and repeat the outer for loop.

```
def depth_first_search(start, goal, max_iterations=100
    Q = [[start]]
    visited = set([start])
    stime = time.time()
    for ind in range(max_iterations):
        # First get N from Q and update Q
        path = Q.pop(0)
        state = path[-1]
        children_set = state.get_children()
        for child in children_set:
            path1 = path.copy()
            if not (child in visited):
                path1.append(child)
                visited.add(child)
                # Check if the goal has been reached
                if path1[-1] == goal:
                    return {
                        'succeeded': True,
                        'path': path1,
                        'num_iterations': ind,
                        'path_len': len(path1),
                        'num_visited': len(visited),
                        'time': time.time() - stime,
                    }
                # Then add new paths to Q from N
                Q.insert(0, path1)
    return {'succeeded': False}
```

Fig. 10: Depth First Search's Code

The only difference between BFS and DFS is the code line before the last line. In BFS, we add the new path at the end of the queue by command *Q.append(path1)*, meanwhile, in DFS, the new path is added at the beginning of the queue Q by command $Q.insert(0, path1)$.

The number of iteration and Path length for the BFS and DFS with 2x2, 3x2, and 4x2 and S1=685, S2=710, S3=111 are shown in Table I and II. As the grid size increases, the number of iteration and path length go up significantly. That is due to the number of states which rise significantly with the grid size. In this context test, BFS seems outperforming the DFS. Off course, the result depends on the initial state of the game. As we change the value of seed (change the initial stage of the grid), the results could be so different.

	2x2			3x2			4x2		
	S1	S2	S3	S1	S2	S3	S1	S2	S3
No. of Iteration	6	2	9	3	164	203	1230	417	681
Path Length	5	3	7	3	15	15	17	13	15

TABLE I: BFS's parameters

QUESTION: For the different seeds, was the path generated from Depth First or Breadth First shorter? Describe

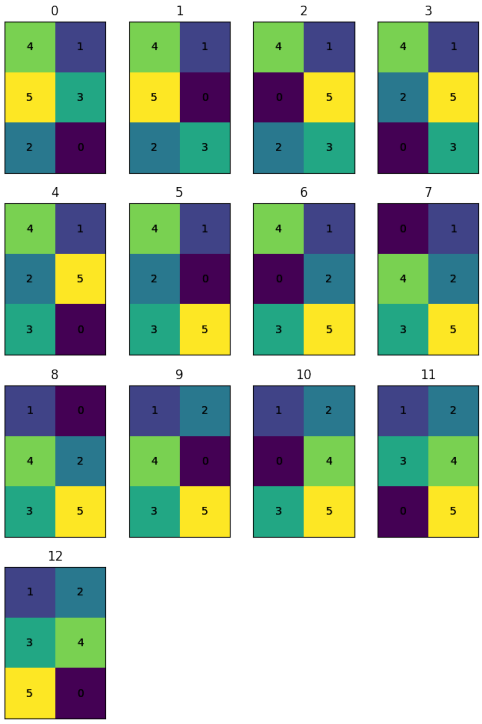

Fig. 11: Results of Breath First Search

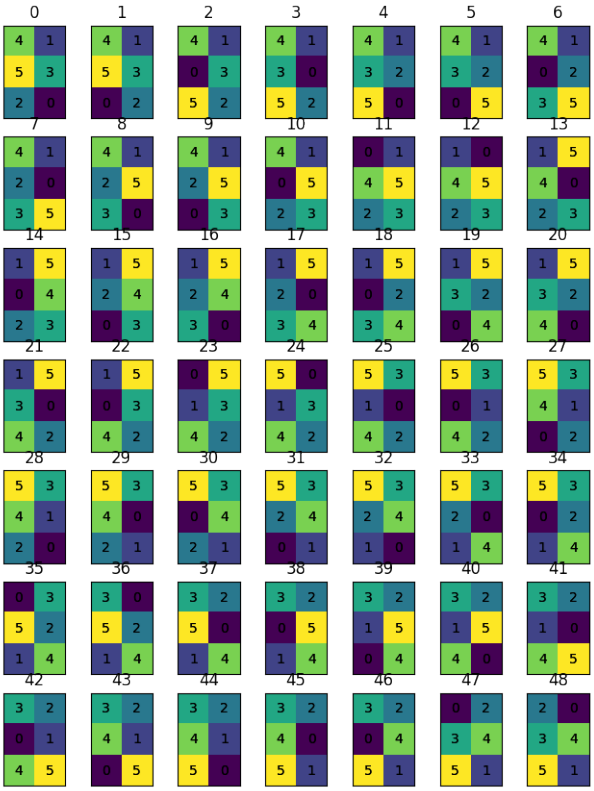

Fig. 12: Results of Depth First Search

	2x2			3x2			4x2		
	S1	S2	S3	S1	S2	S3	S1	S2	S3
No. of Iteration	3	1	5	1	88	154	4189	6666	3401
Path Length	5	3	7	3	83	117	3843	6003	3139

TABLE II: DFS's parameters

why this is the case using theoretical arguments from class
Answer: For different seed, the performance of BDF and DFS are different. That is due to the initial state making a huge influence to the performance. As growing up a search tree with the initial state as root, the goal location in the tree will make the search process fast or slow. If the goal is at the bottom left corner of the tree, DFS will win. Because the search will go as deep as possible on the left, and run from left to right. So, if the goal is on the left, it will find it out sooner. Otherwise, if the goal is on the right side of the tree, the BFS will win because it search by layer from top-to-bottom. If the goal is on the right side, DFS need to go to all the left nodes first, and go to the right. It makes it losing to BFS.

QUESTION: If I were to ask you to find the optimal path to solve a 3x3 sliding puzzle, which algorithm should you use? Why?
Answer: If taking DFS and BFS to compare in 3x3 grid, BFS will win the competition in aspect of time and shortest path. However, as the number of state in 3x3 grid is so huge (over 100000), so I don't think BFS or DFS is a good choice. I think Iterative Deepening (IDS) is the better choice comparing to both DFS and DFS.

The paths for the 3x2 grid are shown in Fig. 11 and 12. For the same initial state, BDS needs a path of 13 states to reach the goal, mean while DFS requires 49 state path to reach the same goal. It is fit with the theoretical calculation in the lecture.

C. Statistical Analysis

We have tested the DFS and BFS with 120 seeds on the 2x2, 3x2, and 4x2 grids, and the result as in the tables below. Overall, BFS is shown significant improvement comparing to the DFS (suitable to the theoretical computation). However, at the 4x2 grid, the mean of visited nodes is smaller than the standard deviation. It is a uncommon results, and an explanation is the vary of visited nodes for each seed are so different. For some seed, the path is so short, and for other case, it is too long. It could be right, because it the goal state is at the top part of the tree, the search process is very short. Otherwise, if the goal is at the bottom part of the tree, the search path will be come extremely long, and the search process needs to visit a lot of state. We can say that the distribution does not strictly follow the bell shape.

	2x2		3x2		4x2	
	Length	Visited	Length	Visited	Length	Visited
Mean	3.2	5.3	14.3	195.5	17.3	3823.6
Std	2.3	4.1	3.4	97.8	5.15	4019.78

TABLE III: The mean and std of path length and visited node of BFS

	2x2		3x2		4x2	
	Length	Visited	Length	Visited	Length	Visited
Mean	5.325	6.1	97.8	165.9	4206.8	8148.75
Std	3.92	4.23	43.5	80.3	3374.4	6542.3

TABLE IV: The mean and std of path length and visited node of DFS

III. 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. 13.

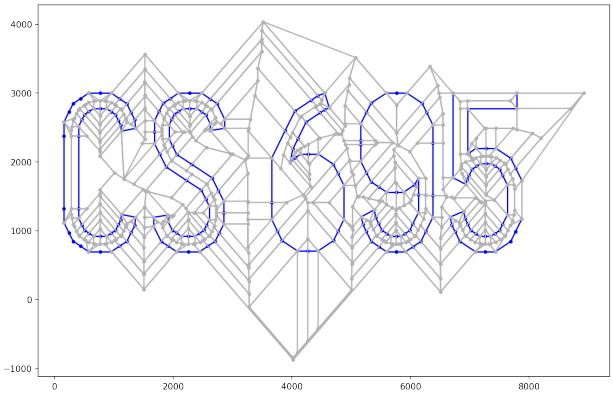

Fig. 13: CGAL - Voronoi diagram

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.