CS 695: Programming assignment (P1)

Md. Ridwan Hossain Talukder
Department of Computer Science
George Mason University
Fairfax, USA
mtalukd@gmu.edu

I. Bug Algorithms

A. Bug 0

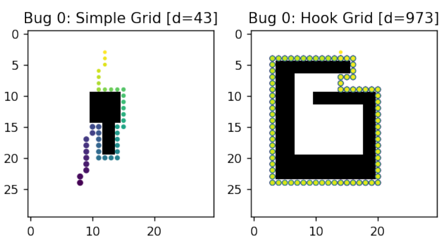

Fig. 1. Running Bug 0 algorithm on Simple Grid and Hook Grid

1) Answer: To go around the obstacle the bug either needs to go clockwise or anti-clockwise but the decision must be uniform. In the hook grid scenario, the bug 0 goes clockwise and to its left when it first hits the obstacle and continues till it finds the m-line inside the object again and starts following it. But it hits the obstacle again from inside and this time also follows the obstacle clockwise which leads it further away from the goal. It continues to follow the obstacle on the outside until it reaches the starting point again and keeps doing the same thing until it reaches the maximum limit of the steps. That's why the path length in the Hook Grid example is very long and does not reach the goal.

B. Bug 1

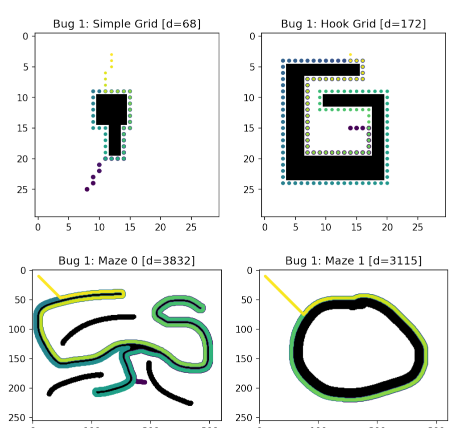

Fig. 2. Resulting plots for Bug 1

```python
import math

def bug_1(grid, start, goal, verbose=False, max_steps=1000):
    all_positions = []
    line = bresenham_points(start, goal)
    robot = BugRobot(position=start, orientation=np.array([1, 0]))
    follow_line = True
    looping_back = False
    for _ in range(max_steps):
        all_positions.append(robot.position.copy())
        # Break when we reach the goal
        if (robot.position == goal).all():
            break

        if verbose:
            print(f"Position: {robot.position} | "
                  f"Orientation: {robot.orientation} | "
                  f"Is On Line: {robot.is_on_line(line)}")

        if follow_line:
            # Follow the line until we cannot
            did_encounter_obstacle = robot.follow_line(grid, line)
            if did_encounter_obstacle:
                follow_line = False
                # The robot remembers where it started.
                obstacle_start_position = robot.position.copy()
                obstacle_closest_position = robot.position.copy()
        elif looping_back:
            # Follow the object until we reach the closest point
            # and recompute the line.
            robot.follow_object(grid)
            if (robot.position == obstacle_closest_position).all():  # When should you stop looping back around the object and follow the line again?
                line = bresenham_points(robot.position, goal)
                looping_back = False
                follow_line = True
            if (robot.position == obstacle_start_position).all():
                break
        else:
            # Follow the object until we loop back to our start position
            # Then set 'looping_back' to True
            robot.follow_object(grid)
            dist_current = math.hypot(goal[0] - robot.position[0], goal[1] - robot.position[1])
```

```
            dist_old = math.hypot(goal[0] - obstacle_closest_position[0], goal[1] - obstacle_closest_position[1])
            if dist_current < dist_old:
                obstacle_closest_position = robot.position.copy()
            if (robot.position == obstacle_start_position).all():
                looping_back = True

    return np.array(all_positions).T
```

C. Bug 2

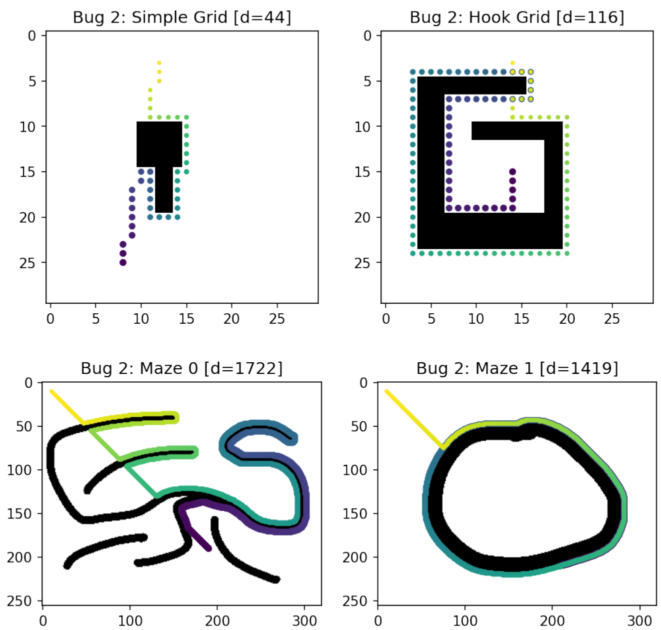

Fig. 3. Resulting plots for Bug 2

1) Answer: Bug 2 performs better in each of the maps. Firstly unlike bug 1 it does not have to go back to the starting point to start traversing again so the path length is smaller. Also bug 2 has a early termination condition that makes it different from bug 1 where it can not reach the goal. So either it reaches the goal with minimum path length or it terminates if it can not.

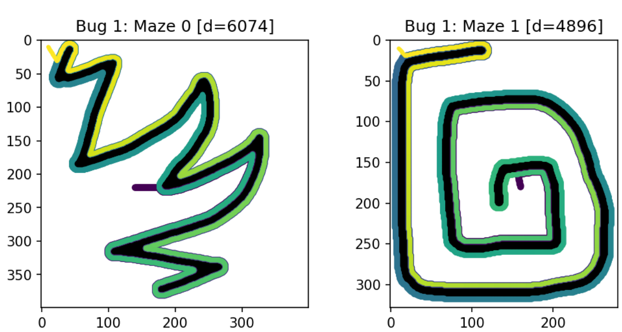

Fig. 4. Custom Map: Bug 1 outperforms Bug 2 in Maze 1

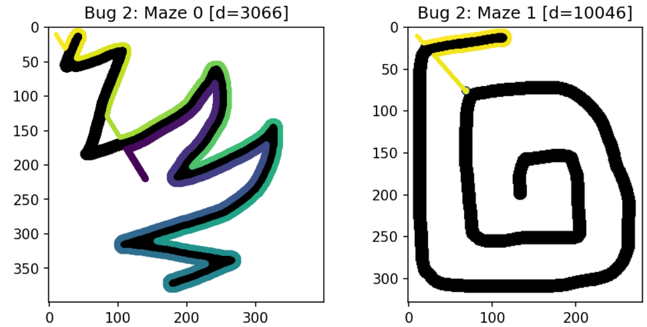

Fig. 5. Custom Map: Bug 2 outperforms Bug 1 in Maze 0

II. State Space Search: Sliding Puzzles

1) Answer 1: In the 3x2 grid the total number of states reached are 360 after 30 iterations. So the ratio is 360/720 = 0.5 for 3x2 grid which is same for the 2x2 grid as well. (12/24 = 0.5). So from this two grids it seems like the number of states that can be reached is approximately 50% of the total number of states that a puzzle can have.

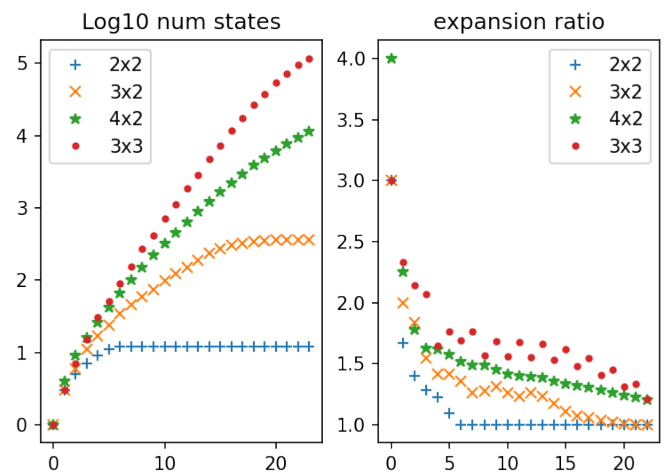

Fig. 6. Plotting: Expanding the states

2) Answer 2: The number of states levels off and asymptotically approaches a constant value because the possible number of states that can be extracted gets fewer after each iteration and it reaches the value if we can not expand the states any more, means the entire state space is generated. For 2x2 grid after 6 iterations we see the extracted states are 12 which is the maximum states that can be expanded and it stays there after 24 iterations too. The expansion ratio reaching to 1 means the same as above that no more expansion is possible.

```
def breadth_first_search(start, goal, max_iterations=100000):
    Q = [[start]]
    visited = set([start])
    stime = time.time()
    for ind in range(max_iterations):
        # First get N from Q and update Q
        N = Q.pop(0)
```

```
        # Check if the goal has been reached
        if N[-1] == goal:
            return {
                'succeeded': True,
                'path': N,
                'num_iterations': ind,
                'path_len': len(N),
                'num_visited': len(visited),
                'time': time.time() - stime,
            }

        vertices = N[-1].get_children()
        for vertex in vertices:
            if vertex not in visited:
                visited.add(vertex)
                Q.append(N + [vertex])
        # Then add new paths to Q from N (and its children)
    return {'succeeded': False}
```

```
def depth_first_search(start, goal, max_iterations=100000):
    Q = [[start]]
    visited = set([start])
    stime = time.time()
    for ind in range(max_iterations):
        # First get N from Q and update Q
        N = Q.pop()
        # Check if the goal has been reached
        if N[-1] == goal:
            return {
                'succeeded': True,
                'path': N,
                'num_iterations': ind,
                'path_len': len(N),
                'num_visited': len(visited),
                'time': time.time() - stime,
            }

        vertices = N[-1].get_children()
        for vertex in vertices:
            if vertex not in visited:
                visited.add(vertex)
                Q.append(N + [vertex])
        # Then add new paths to Q from N (and its children)
    return {'succeeded': False}
```

TABLE I
RESULTS FOR 3 DIFFERENT SEEDS

		2x2		3x2		4x2	
Planner	Seed	L	I	L	I	L	I
BFS	695	1	0	13	139	5	17
BFS	710	3	4	15	203	13	588
BFS	111	7	11	15	243	15	951
DFS	695	1	0	71	76	9	8
DFS	710	3	2	83	89	6003	6667
DFS	111	7	6	117	155	3139	3402

3) Answer 1: The path length of BFS was shorter (or at least equal) over different seeds. As for DFS, it is not guaranteed to have the shortest path always as DFS explores the children (left to right) of the node before exploring the siblings of that node. On the other hand BFS explores all the siblings before exploring the children, thus using BFS will always gives us the shortest path from the goal, but DFS may not. So the path length obtained for a particular search problem from BFS will be shorter or equal to the path length obtained from DFS.

4) Answer 2: I would have used BFS as it is complete and will always give the optimal solution in terms of path length (if a solution is possible).

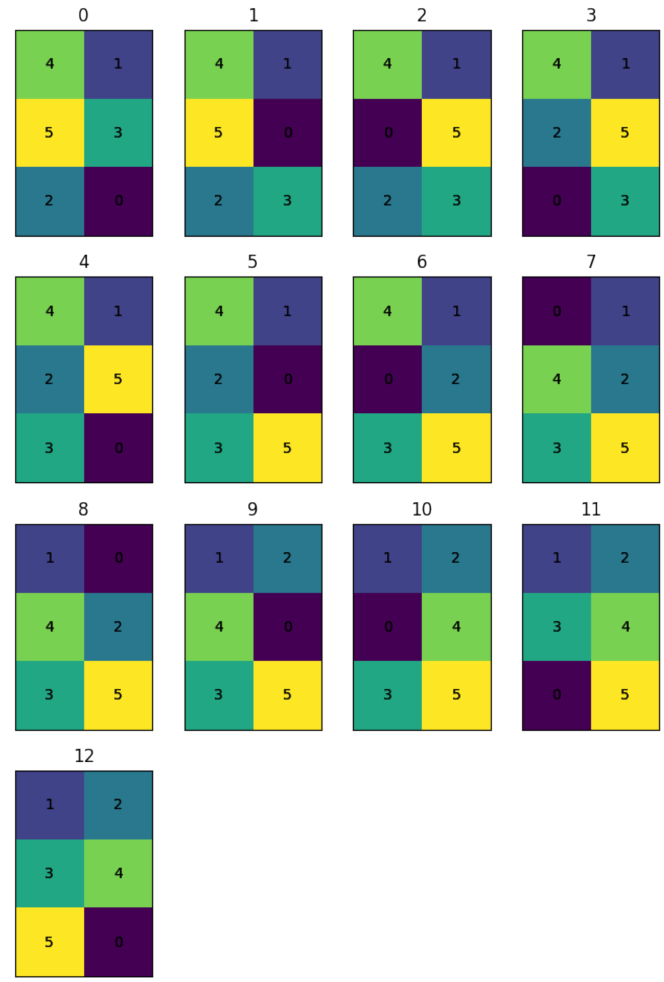

Fig. 7. Solution path vizualization for a 3x2 puzzle using BFS

TABLE II
STATISTICS GENERATED FOR 101 DIFFERENT SEEDS

Dimension	Planner	Length	Number of States
2x2	BFS	4.176 ± 1.937	7.461 ± 3.578
2x2	DFS	6.33 ± 3.369	7.206 ± 3.583
3x2	BFS	11.392 ± 3.87	146.451 ± 97.98
3x2	DFS	89.725 ± 50.516	186 ± 119.409
4x2	BFS	16.745 ± 5.263	4157.01 ± 4777.542
4x2	DFS	89.725 ± 50.516	10758.725 ± 6563.863

III. DOCKER, CGAL, AND VORONOI DECOMPOSITIONS

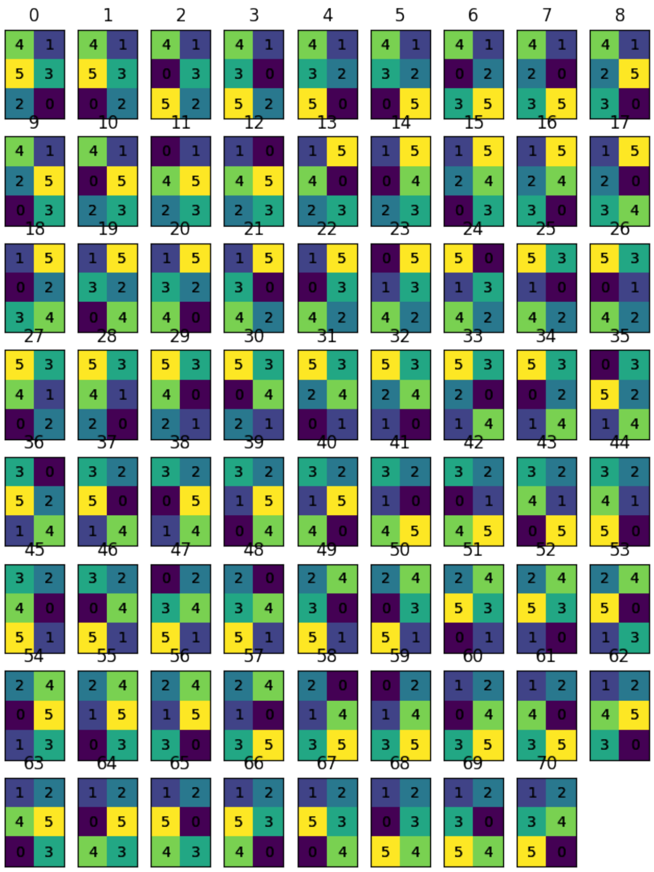

Fig. 8. Solution path vizualization for a 3x2 puzzle using DFS

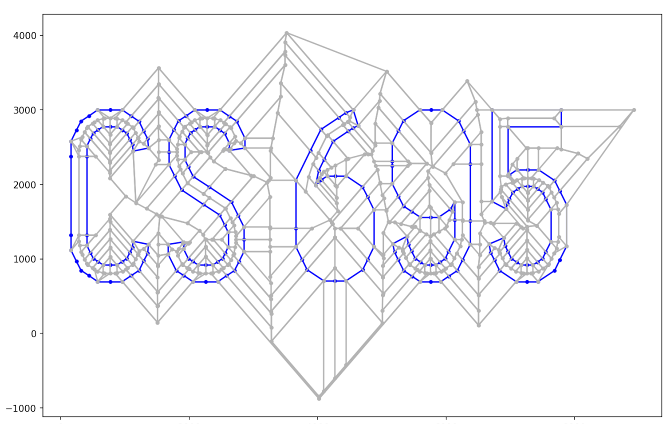

Fig. 9. Using Docker, CGAL, and Voronoi Decompositions