CS440: The Maze is on Fire

Andy Rivera, John Juarez

February 19, 2021

1 Maze Generation and Project Setup

To generate a maze we used the function **generateMaze(int dim,double p)** it takes in the parameters of *dim* to construct a dimxdim 2D array and p to determine the probability of a block being filled(1) or not(0).

To set up the project for the path finding algorithms we created an object **Point** with the following attributes:

- Point parent (Previous location of agent)
- X and Y to keep track of location of agent
- stepsTaken (Amount of steps taken to get to the current location.)
- Hueristics (Estimate for A* algorithm)

```
Set dimension 'dim' to create maze
10
Set probability 'p' to create barricades
0.1
0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0
0       0       1       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       1       1       0       0       0
0       0       0       0       0       0       0       0       0       0
1       0       0       0       0       0       0       0       0       1
0       0       0       0       1       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0
0       0       0       1       0       0       1       0       0       0
Set dimension 'dim' to create maze
10
Set probability 'p' to create barricades
0.6
0       0       0       0       0       0       0       1       1       1
0       0       1       1       0       1       1       0       0       1
1       1       0       0       1       1       0       1       1       1
0       1       0       0       0       1       0       1       1       0
1       1       0       1       1       1       1       1       1       1
0       0       0       1       1       0       1       1       0       1
1       1       0       1       1       1       0       0       1       0
1       0       1       0       1       0       1       0       0       0
0       0       1       0       1       0       1       0       1       1
1       0       1       0       1       0       1       0       1       0
```

Figure 1: Maps generated with p = 0.1 and 0.6 respectively

2 DFS Algorithm

We created the method **mazeDFS(int[][] maze, Point start, Point goal).** It takes in 2 points and returns a boolean, True if there exists a path and false if no path exists between the starting point and the goal point.

With two arbitrary points in the maze, DFS would be a better option because we are just determining if there is a path between the two points and not necessarily the shortest path, it will save space in memory since it adds less points into the fringe by not exploring all of its neighbors.

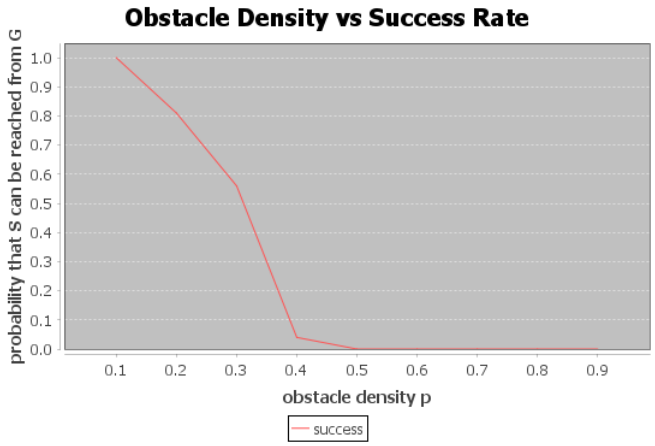

Figure 2: obstacle density p vs probability that S can be reached from G

3 BFS and A* Algorithms

For our BFS algorithm, we created the method **mazeBFS(int[][] maze).** It returns the ArrayList of the points that make up the shortest path.

For our A* algorithm, we created the method **mazeAStar(int[][] maze).** It uses a priority queue that prioritizes points based on an estimation that is determined by adding up the amount of steps taken(cost so far) + euclidean distance + the de-prioritization(Steps to the left and steps up will have less priority) of steps that take you further from goal.

If there is no path between the starting point and the goal point the difference between nodes explored by BFS and A* would be 0 because both algorithms would keep trying to explore every point in the maze.

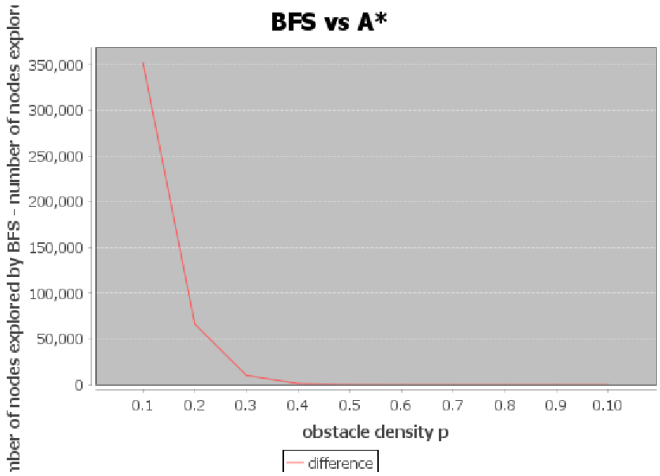

Figure 3: number of nodes explored by BFS - number of nodes explored by A* vs obstacle density p

```
7  7  0  0  0  0  1  0  0  0  0  0
1  7  0  1  1  0  0  0  1  0  0  1
1  7  0  0  1  1  1  0  0  0  0  0
1  7  7  7  7  7  0  0  0  0  0  0
0  1  0  0  1  7  7  7  0  0  0  0
1  1  0  1  1  0  1  7  7  0  0  0
0  1  1  0  0  1  1  1  7  0  0  0
0  1  0  0  0  1  0  0  7  0  0  1
1  0  1  0  0  0  0  1  7  1  0  0
0  0  0  1  1  0  0  1  7  0  0  0
0  0  0  0  0  0  0  0  7  0  0  0
0  1  0  1  1  0  0  1  7  7  7  7
```

Figure 4: BFS Shortest path represented by 7 dim = 12 p = 0.3

```
7  7  7  0  0  0  1  0  0  0  0  0
1  0  7  1  1  0  0  0  1  0  0  1
1  0  7  0  1  1  1  0  0  0  0  0
1  0  7  7  7  7  7  7  7  0  0  0
0  1  0  0  1  0  0  0  7  0  0  0
1  1  0  1  1  0  1  0  7  0  0  0
0  1  1  0  0  1  1  1  7  0  0  0
0  1  0  0  0  1  0  0  7  0  0  1
1  0  1  0  0  0  0  1  7  1  0  0
0  0  0  1  1  0  0  1  7  7  0  0
0  0  0  0  0  0  0  0  0  7  7  0
0  1  0  1  1  0  0  1  0  0  7  7
```

Figure 5: A* Shortest path represented by 7 dim = 12 p = 0.3

4 Algorithms in less than 1 minute

At p = 0.3 the largest dimension size the algorithm can find a path in less than one minute is...

- **DFS** dim =
- **BFS** dim =
- **A*** dim =

5 Strategy 3

The strategy that we implemented finds an initial path from the starting point and the goal point. As the agent advances through the burning maze it also scans the path ahead before it advances. In a helper method **scanPath(int[][] maze, int currIndex, ArrayList¡Point¿ path, ArrayList¡Point¿ fire-Points, double q)** it scans ahead of the agent by counting the steps of from the closest point that is on fire and determines the probability of the point catching on fire.

The more burning neighbors the point has the greater the probability of the point catching fire next. Using that as a threshold, if the point surpasses the threshold, the agent will try and generate a new path with less risk of catching on fire if there is one.

```
Congrats you made it out the fire
7  7  1  2  2  2  2  2  2  1  2  2  1  0  1
1  7  7  0  1  2  2  2  2  2  2  1  1  0  0
0  1  7  0  0  2  1  1  1  1  2  1  1  0  0
0  1  7  1  0  0  0  0  1  2  2  1  0  0  0
0  0  7  7  7  0  0  1  2  2  2  2  1  0  0
0  0  0  0  7  0  1  0  1  2  1  1  0  0  1
1  0  0  1  7  1  1  1  2  2  2  1  0  0  1
0  1  1  0  7  7  1  2  2  2  1  0  0  1  0
1  0  0  0  1  7  7  1  2  2  1  0  1  1  1
1  0  0  0  0  1  7  7  1  2  2  1  0  0  0
0  0  0  0  0  0  1  7  1  2  1  7  7  7  1
0  0  0  0  0  0  0  7  7  7  7  7  1  7  1
0  1  0  1  0  0  0  0  1  0  1  0  1  7  7
1  1  0  0  1  1  0  1  0  1  0  1  0  0  7
1  1  1  1  0  0  0  1  1  1  1  0  0  0  7
```

Figure 6: Strategy 3: dim = 15, p=0.3, q=0.3 7-represents path 2-represents fire

6 Strategies Success Rates

7 Unlimited Computational Resources

If we had unlimited computational resources, our strategy 3 would be able to scan and calculate the probability of all the points in the maze based on where the fire is starting and not only the initial path. Essentially adding all the points in the maze that are not on fire in it's fringe. It will almost always find a path since it can accurately predict all the points that are going to catch on fire and modifies it's path accordingly.

8 10 Seconds to Make a Move