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.

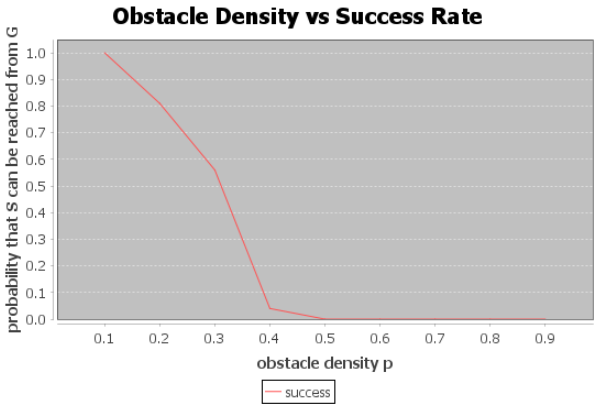

Figure 2: obstacle density p vs probability that S can be reached from G

3 BFS and A* Algorithms

For our BFS algorithm, we created the method mazeBFS(int[][] maze). It returns an ArrayList of points that create a minimal path that exists from the start to goal state. In order to avoid an endless traversal of the same path, another ArrayList is used to keep track of the Points that have already been explored.

For our A* algorithm, we created the method mazeAStar(int[][] maze). It uses a priority queue that prioritizes points based on the greatest heuristic. The heuristic for a point is determined by a 3-step formula, de-prioritization of paths that take steps backwards, i.e. 'left' and 'up', prioritization of paths closer to the goal state, and extra priority for the paths that have open space in either forward direction, i.e. 'right' and 'down'. Essentially, the idea behind the algorithm was to focus on paths with the most open space up ahead that can take the agent the furthest, closer to the goal, and provide much more alternatives in the case of an obstructed path. We determine the open space ahead by looking in the direction the point's parent would take to get to that point and if its right or down, we count all the open cells ahead of it and add it to the heuristic. If the path is obstructed and the only possible movement is a backward step, we take

this into consideration by taking the minimal steps from the head of that path to the goal minus the steps taken thus far and the steps remaining. If the path did not have to make any backward steps this calculation should be 0, otherwise it will be negative, and this value will be magnified by multiplying it by 10, which would allow the algorithm to explore other paths with a greater heuristic. In this case, the algorithm will then opt for an earlier point in the path that is still relatively close to the goal state as we give a slight edge to furthest points by taking the steps taken thus far minus the steps remaining until reaching the goal. If this calculation is positive, then it has passed the halfway mark of the maze and so we give it a slight advantage by taking the calculation and dividing it by 2. If it hasn't past the halfway mark, we don't give it any advantage at all but we also don't give it a disadvantage because we would still like to consider these earlier points if longer paths aren't making their way to the goal state. These calculations can be found in our getHeuristic() method.

In our driver we use the method generateAStarVsBFSAnalysis() to generate the plot of the average 'number of nodes explored by BFS – number of nodes explored by A*' vs 'obstacle density p'. Essentially both algorithms are put to the test in a 10 test trial series that runs 10 different mazes for each probability 'p', recording the difference of the 10 outputs which is then stored in an array keeping track of the average by taking these results and dividing it by 10. This concludes 1 test and this process repeats for 9 more tests. Once all 10 tests have concluded, the average of the 10 tests are taken and used to generate our plot which will be displayed after the method has finished execution.

Typically, what we see is our A* is a dramatic improvement to the BFS algorithm as demonstrated in the graph above. If there is no path from S to G, then the difference is 0 as both algorithms will end up exploring all possible paths that don't lead anywhere. This can be seen in the graph at obstacle density 0.4 and up.

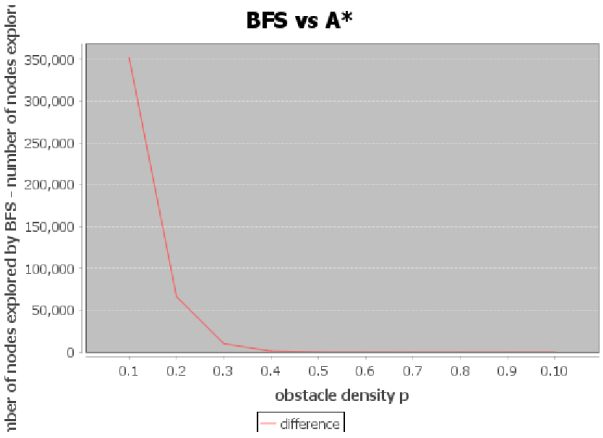

Figure 3: number of nodes explored by BFS - number of nodes explored by A* vs obstacle density p

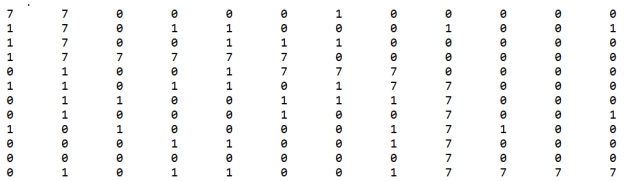

Figure 4: BFS Shortest path represented by 7 dim = 12 p = 0.3

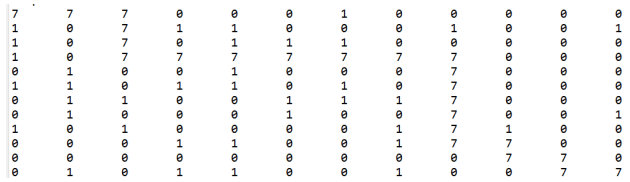

Figure 5: A* Shortest path represented by 7 dim = 12 p = 0.3

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 = 650
- **BFS** dim = 20
- **A*** dim = 2100

5 Strategy 3

For our implementation of Strategy 3, in the method strategy3(), we took an approach that would only re-compute a new path when a point in the path is currently on fire or is at risk of being on fire soon. Upon each step, our algorithm uses the helper method findFirePoints() to scan the maze and return an ArrayList of Points where the cell is on fire. It then scans the path ahead of the agent's position using the helper method scanPath().

This method will initially traverses the path to see if any path point is on fire. If a point in path is on fire, it attempts to recompute a path, using mazeAStar(), if possible, otherwise returns null as nothing more can be done. This is necessary because we would like to prioritize avoiding points that will lead to death first before avoiding points that are at risk.

After scanning the path and avoiding any fires that obstructed it, we then create two important arrays of the same length (size of path points ahead of agents position), one to keep track of the minimal path from a fire point to

a path point, and another to keep track of its respective probability of that fire obstructing the path point. scanPath() will then traverse the list of fire in respect to the path points, ahead of the agents position, filling these arrays with information we will consider in the following step. The minimal path is found using our modifiedMazeBFS() method which takes a start point, for the fire point, goal point, for the path point, the current state of the maze, and the number of steps the agent would need to take to reach the path point. The algorithm runs just as a regular BFS search except that, it will terminate and return null, i.e. no path, if it is taking too long (after 5 seconds), which would likely mean no path or a significantly long path, and it will return null if the path is longer than the number of steps before the agent passes it. Outside of these two cases, it will operate as a regular BFS search returning a minimal path if one is found or null if none exists. With the minimal path found from the current fire point to path point, we then fill its respective index in the two arrays with the minimal path and its probability of fire if, the minimal path is empty or the recent minimal path found takes less steps than the previous path. The probability of fire is found using our calculateProbability() method which takes the current state of the maze, the minimal path the fire point can take to the path point, i.e. fire path, and the probability of flammability 'q'. It traverses the fire path and multiplies the probability for each step up until the path point, also considering the adjacent neighbors on fire along the path which will further increase the probability of the path point catching on fire.

Once a minimal path and probability is calculated for each path point, scanPath() will then traverse these two arrays considering if the probability is above a certain threshold, if the minimal path takes less than 4 steps, and if the number of steps for the agent to pass the path point is greater than the steps the minimal path would take from the fire point to path point. The probability threshold is dynamic and different for flammability 'q' being between 0.1-0.3, 0.4-0.6, and 0.7-0.9. We determined which thresholds performed the best in these three areas through trial and error and assigned the most successful thresholds respectively. If the three conditions mentioned are met, we then consider this path point to be at risk and attempt to avoid them by simulating the fire path the fire would take to the path point in a temporary copy of the maze and attempt to recompute a path using mazeAStar(). If one is found, then we update the path scanPath() will return, and maintain the simulated fire so future path updates can continue to avoid this risk point, otherwise, we keep the current path and reset the temporary maze to its previous state before simulating the fire. No path would essentially mean our agent will have to take the risk of crossing this path point at risk.

After traversing the minimal path and probability arrays, scanPath() will then return to strategy3() which will update the current path the agent is on with the optimal path provided to it by scanPath(), if one exists, otherwise, it will keep following the same path and take another step forward, repeating the same process.

```
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

The algorithms perform similarly when the flammability q is bigger than 0.7. At that rate it is very difficult to find a path since the fire is spreading really fast. At lower flammability rates strategy 3 performs better that both the first strategies. Since our strategy tries to predict the points that are going to be on fire next using the probability threshold.

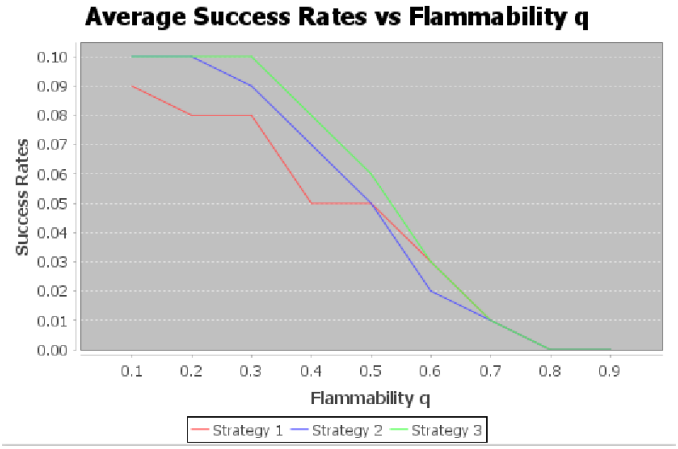

Figure 7: Strategies Analysis created for mazes with dim size 50 and p =0.3

```
You died in the fire!
7   7   0   1   1   0   1   2   1   0   1   0
0   7   1   0   0   1   0   2   1   0   0   1
1   7   0   1   0   0   1   2   2   0   0   0
0   7   7   7   7   2   2   2   1   0   0   1
1   0   1   1   7   9   0   2   2   0   1   0
1   0   0   1   0   1   1   2   2   0   0   1
0   0   0   1   0   0   1   0   0   0   0   0
0   1   0   0   0   1   1   0   0   0   1   0
1   1   1   0   0   1   1   0   0   1   1   1
0   0   1   1   0   0   0   1   0   0   1   0
0   0   0   0   0   0   1   1   0   0   0   1
0   0   1   0   0   0   0   1   1   0   0   0
```

Figure 8: Strategy 1 failed attempt at dim = 12, p=0.3 q=0.3 7-represents path 9-represents agent catching on fire 2-represents fire

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

Figure 9: Strategy 2 Success at dim = 15, p=0.3 q=0.3 7 represents path 2-represents fire

7 Unlimited Computational Resources

If we had unlimited computational resources, we would likely look to improve strategy 3 by adding an extra level of estimation of probability before resorting to finding new paths that avoid risky points. Essentially, we would change our Strategy 3 by getting an accurate number of steps from each fire point to each path point using our modified BFS algorithm right away, rather than

counting the cells in-between as an estimate. We would then be able to calculate an accurate probability for each path point being set on fire: $q^k(k = number of steps in the fire path)$ while also considering adjacent fire neighbors that would increase the likelihood of points in the fire path being set on fire, further increasing the probability of flammability for the path point. We would only consider the probabilities with the least amount of steps to the path point. If the probability of flammability is above a certain threshold, we then count the steps before the agent passes this point at risk, if it's more than the amount of steps before the fire reaches the path point, we will then opt for a new path by simulating this fire path and avoiding it if possible.

This would improve strategy 3 by resulting in less unnecessary re-computations of paths. Rather than re-computing simply based on the number of steps in between the fire point and path point, it will also consider the probability of that path point being set on fire in x amount of steps. Re-computing less unnecessary paths also means a lower likelihood of having to take backward steps to avoid the 'possible' fire and reaching the goal state much quicker.

8 10 Seconds to Make a Move

If we had 10 seconds to make a move, our strategy 3 would not word since it will take up a lot of time to calculate the probabilities of points and recompute a new path if it has too. We would need to come up with a new strategy that works against the time constraints and stay as far away from the fire as possible. Using DFS would work quickly because it would only need to keep track of the path it is taking initially. To optimize the DFS algorithm for account for the fire we could add a simple heuristic to move along the maze much more efficiently. Since there is a time limit the heuristic cannot be computational heavy. We can make a heuristic that maximizes the distance from the fire and and minimizes the distance from the goal point. Since DFS explores less points it would only need to calculates the heuristic based on the path it is taking. It greedily chooses the the next point it takes, moving along the path as efficiently as possible between the 10 second intervals.

9 Concluding statements

Andy Rivera worked on Problems 3,5,7

John Juarez worked on problems 1,2,4,6,8

I,John Juarez, honor that all work is my own and not copied or taken from online or any other student's work.

I,Andy Rivera, honor that all work is my own and not copied or taken from online or any other student's work.