Sunny Cheng, Maxwell Wang
19 Feb 2021

Fire Maze

Professor Cowan
CS 440

Intro

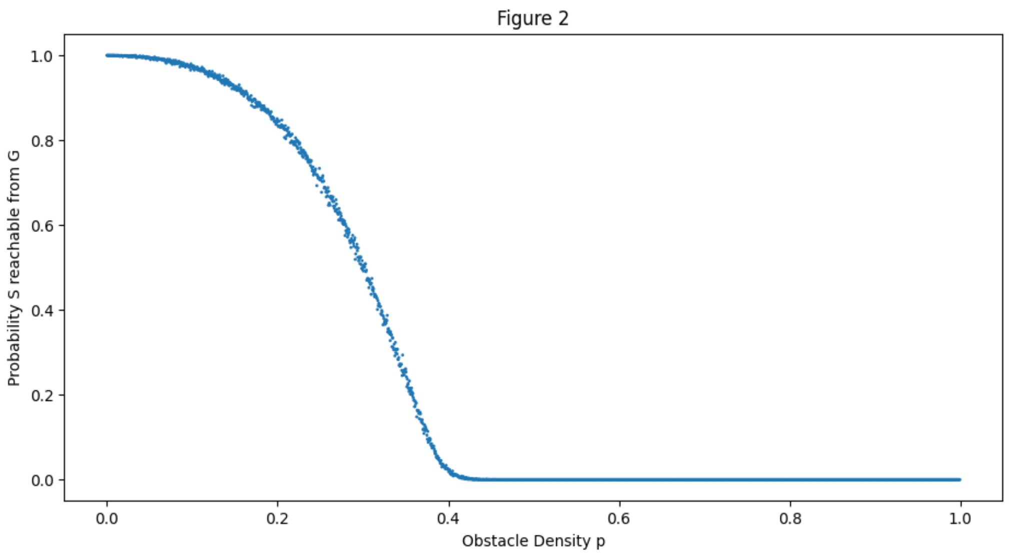
Group members: Sunny Cheng (sc2040) and Maxwell Wang (mlw195)
Group contribution: Each group member contributed equally. Maxwell primarily completed parts one through three. Sunny primarily did four through six. Maxwell did seven and Sunny did eight. We both worked on trying different options for strategy 3.
Group statement: All work here is our own, and we did not copy or take code online or from other students.

Problem 1

Please refer to the code, generate_maze function in util.py. It takes in dim and obstacle density p. It checks that dim is positive integer and p is float between 0 and 1. We represent the maze as a 2d list with 0's for empty cells and 1's for obstacles. We use the probability to populate the maze with obstacles, and finally set the start and goal cells as empty before returning our 2d list maze.

Problem 2

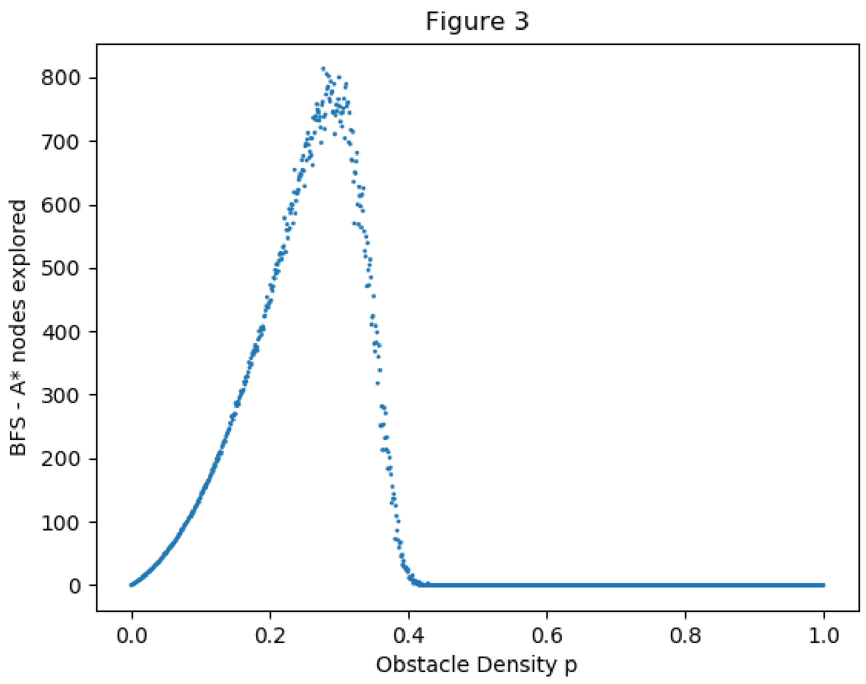
When trying to determine if some location is reachable from another in the maze, DFS is superior to BFS. BFS must search every cell that is closer in distance than the target cell before finally finding it. On the other hand, DFS does not have to do this and its search path moves freely through the maze. DFS also does not need to keep track of as many nodes in its fringe since it prioritizes depth first rather than filling the fringe with all nodes of the same breadth, so its dramatically smaller memory requirements also help determine reachability more easily. For a maze of dimension 100 by 100, we have used DFS to generate a plot of the probability that the top left cell S, can reach the bottom right G for various obstacle densities p.



As you can see, the probability of a path existing from S to G flattens out at 0 at approximately .4 obstacle density.

Problem 3

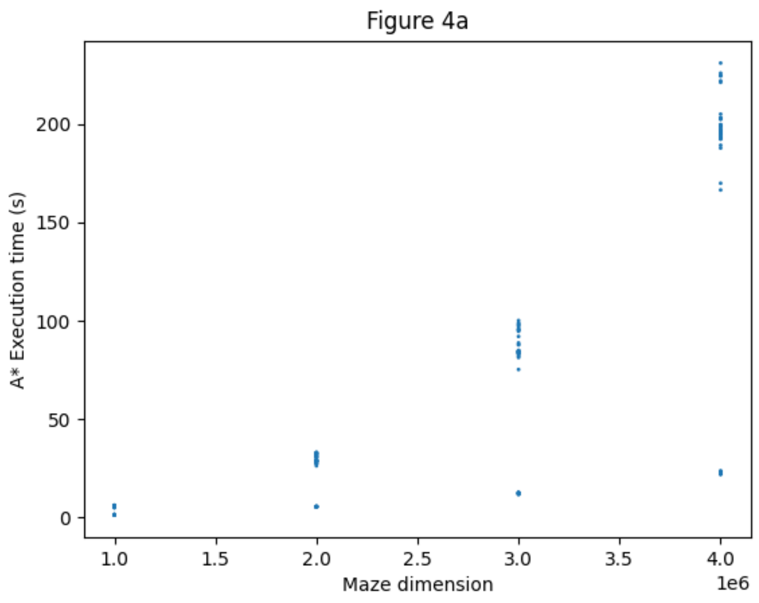
We now compare the number of nodes explored by BFS and the A^* algorithm respectively. Also for a 100 by 100 dimension maze, we plot the difference of the number of nodes visited by BFS and A^* for various obstacle densities. If there is no path from S to G, clearly the difference will be 0 since both algorithms will visit every possible reachable cell in an attempt to reach G before failing.

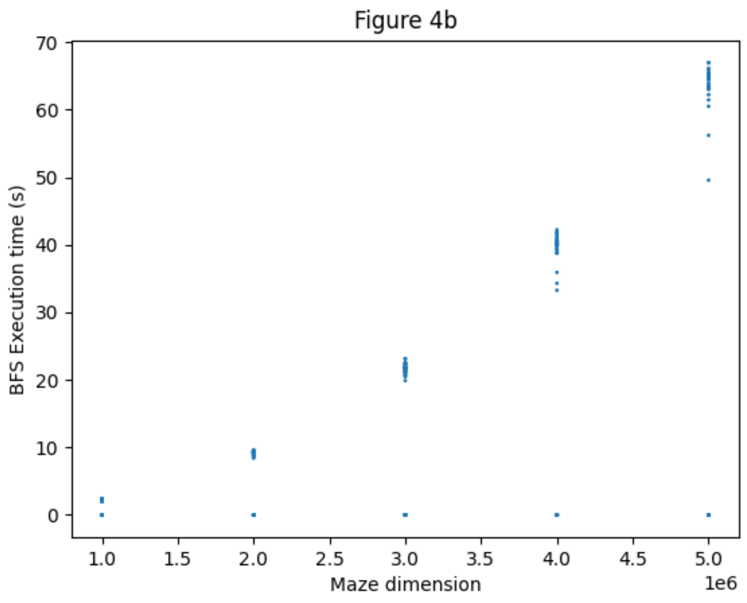


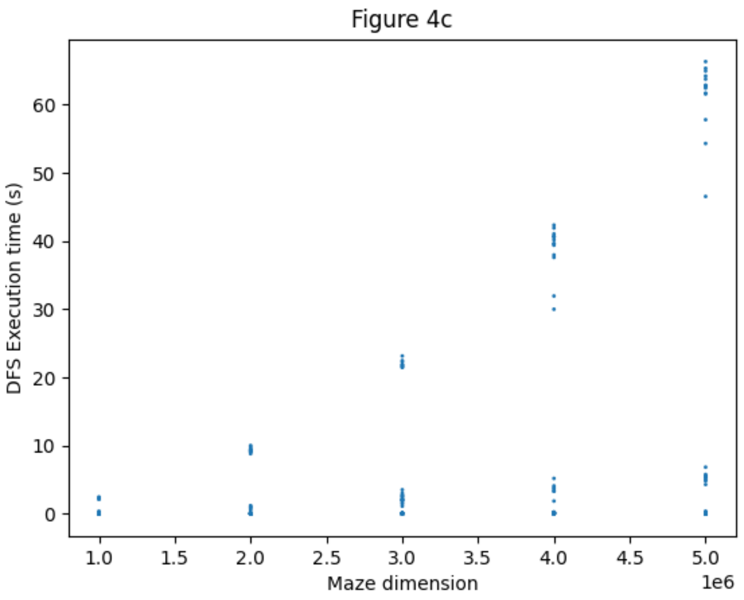
The graph indicates that at approximately .35 obstacle density is where BFS performs the worst relative to A^*, exploring a max of 800 more nodes than A^* has to in a 100 by 100 maze.

Problem 4

We tested several trials of each algorithm (A^*, BFS, and DFS) at various dimensions and timed them to see the largest size they could solve in under a minute.







Looking at the graphs, it seems that DFS can handle at most a 4,500 by 4,500 maze in under a minute. The same holds for BFS. This makes sense, both must search the whole maze in the worst case. Finally, A^* takes much longer, and it would only be able to do a 2,100 by 2,100 maze in under a minute. This too makes sense since A* has to keep track of extra information: for each node in the fringe, what has been the distance traveled so far to it and what is its estimated distance remaining to the goal. And every step, it must recalculate the best next node to explore whereas DFS and BFS do this much faster at the expense of not finding the shortest path and exploring too many suboptimal paths respectively.

Problem 5

We know information about the fire's spreading mechanism and want to leverage that information to augment our search algorithm. To do this, we first need a way to model the fire. Second, we need to include information from this model within our algorithm such that it will improve it.

We model the fire by providing a prediction of the fire s steps into the future. This is useful because we want to know how likely it is that once an agent arrives to a cell that it will be on fire at that time, so we can choose a path that avoids it. To do this, we compute the expected probability of the fire spreading rather than simulating whether it spreads or not. We previously tried simulating the fire spread 100 times to make our estimations, but this proved to be very slow and computationally expensive, motivating us to seek a faster estimation using expected values.

We are told the fire spreads with probability $1-(1-q)^k$. We are given q, so value we are interested in estimating is k: the number of neighboring cells on fire. To simulate some cell one step into the future, we look at the current simulated maze. Let p_1, p_2, p_3 and p_4 be the probability that each of the cell's neighbors is on fire currently. First, let's examine the case where all $p_i > 0$, so it's possible all four neighbors are on fire. Then, the probability $k = 4$ is $\prod p_i$. However, $P(k = 3)$ gets messy fast. Additionally, all the multiplication takes a lot of time. Thus, we simplify it by using an average probability: $p = avg(p_1, p_2, p_3, p_4)$. Then $P(k = 4) = p^4$, something much simpler. We have the expected value of k to be $E[k] = \sum_{i=1}^{4} i \cdot P(i \text{ cells on fire})$. This is simply $4 \cdot p^4 + 3 \cdot 4p^3(1-p) + 2 \cdot 6p^2(1-p)^2 + 1 \cdot 4p(1-p)^3$. However, remember that we assumed all $p_i > 0$. If there is some neighbor that is not on fire, we would do similar calculations with only the cells that are on fire.

Now that we have k, we can finally calculate $1-(1-q)^k$. However, we have to calculate this with many cells for many iterations, so we again sought to simplify this. We were able to generate a linear model: $p \cdot q(0.3 + k \cdot 0.7)$. After testing, this proves to be a fairly accurate estimation of the true value of $1-(1-q)^k$ using our estimation for k. Finally, because we are dealing with probabilities, we decided to round probabilities below 6.5% and above 95%. These bounds ensure not too many cells have to be updated while still being an accurate model. With this, we are able to provide what we hope are accurate predictions of the fire.

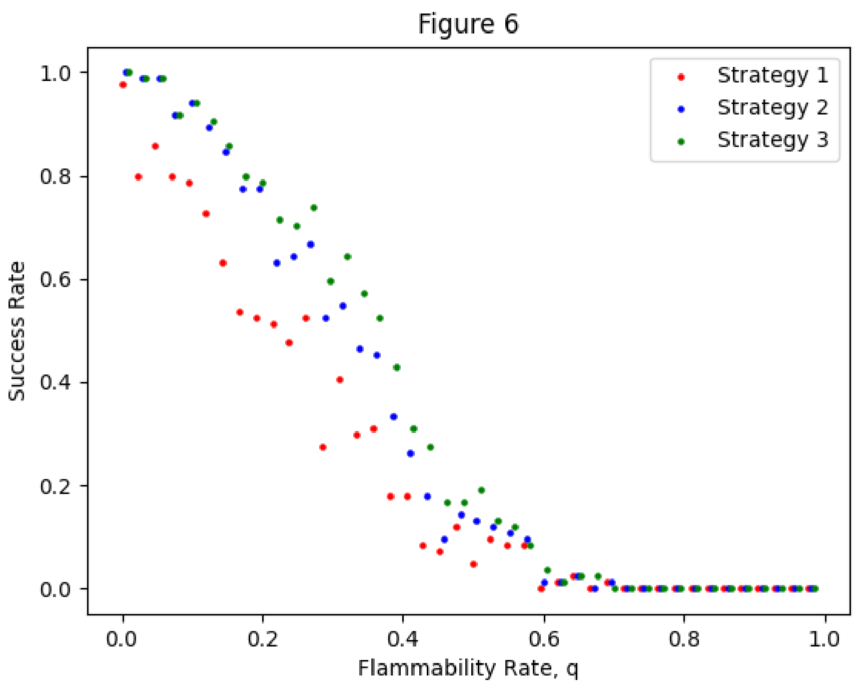
Another area we optimized is the computation of the fire. The given algorithm in the assignment is very slow, requiring iteration across every cell of the maze. This is not needed; only cells that are next to cells on fire are needed. Thus, we keep track of every cell of the fire, and only update cells which are neighboring those and the fire cells itself. Furthermore, if a cell is 100% on fire and is surrounded by cells 100% on fire, we no longer need to simulate it. Thus, we can remove the fire from the cell at that point.

Now that we have that information, we want to leverage it in our search algorithm. We chose the A^* algorithm as a starting point with the taxicab distance (horizontal + vertical distance) from the agent to the goal as its base heuristic. If we think about the heuristic as some cost that a particular cell has associated with it, and we want cells

to avoid fires, then it makes sense to include the chance of a cell being on fire in the heuristic. We found the best implementation would multiply the probability of fire by 50 and add that to the heuristic. This means that if some cell had a 5% chance of being on fire, it costs 2.5 more, and one can interpret that as that cell taking an estimated 2.5 steps to traverse to. The exact calculation is as follows: for some cell, calculate the taxicab distance, d from the agent to the cell. Access the d^{th} future maze and find the probability of that cell being on fire. Multiply that by 50 and add it to the taxicab distance from the cell to the goal cell.
We named this strategy SCORCH (Smart Circumvention Of Risk Convergence Hotspots) because we our algorithm is smart enough to increase the heuristic estimate for cells with high chance of being on fire when agent arrives, so we are able to circumvent this convergence of the agent with fire. Also, scorch and hotspots both refer to fire.

Problem 6

The following is a graph of all strategies and their success rates over several values of q. Strategy 1, 2 and 3 are red, blue, and green respectively.



For extreme values of q, all strategies are about the same. When q is small < 0.15, the fire spreads very slowly and it becomes unlikely for it to play a factor in the search algorithm, as most of the benefits from an intelligent search won't have to be applied if the fire never interferes with the path. When q is large, > 0.70, the fire spreads very rapidly, and it is quite possible it is impossible no matter the algorithm as the paths may have been cut off no matter which steps you take. The strategies perform the most differently in the middle, where $0.15 < q < 0.70$. With this medium fire spread, the intelligent algorithms show their strengths. Strategy 2 is able to recalculate the shortest path every time, and this allows it to avoid paths where the fire has passed through. Strategy 3 is able to do this, but also avoids where the fire might venture in the future, giving it an advantage over strategy 2.

Problem 7

In strategy 3, we are simply estimating the chances with several approximations. However, with unlimited compute time, we could simply try every path combination and simulate it millions of times and pick the path with the best chance of success. Relying on an immense amount of simulations would give us an even more accurate idea of the true probability any cell would be on fire once agent arrives there, giving us a clearer picture of which paths are safer and which are more dangerous. This algorithm would do the following every step to decide on its move:

1. Generate 4^{2*dim}, dim being the dimension of the maze, paths which is every combination of up, down, left, and right for $2 * dim$ moves.
2. Conduct infinite (or as many as possible) simulations in which you move one step according to the path then simulate the fire. Record the percent of time that the agent reaches the goal successfully.
3. Sum the success probabilities for paths starting with the same direction.
4. Move in the direction that has the highest total sum probability.

Problem 8

Our strategy already takes less than ten seconds per step. However, there are ways to optimize it further. One way is to not recalculate all the future mazes every step. Instead, we can recalculate once every ten or even 100 steps, tuning it to the amount of compute time we have. We also currently calculate $2 * dim$ mazes into the future. It is possible we do not need to calculate this many steps into the future, so we could also adjust how many steps into the future we look. Another way would be to only run A^* once every few steps, and use the generated path from the last run as steps. In a maze where the fire rarely spreads/changes, it makes sense to only consider a new path when the fire changes, otherwise maintaining the same path. Finally, for very large mazes, we could instead search a subset of the maze, say a 100 by 100 box, with the top left being our agent position. We would attempt to reach the bottom right of the box. The size and placement of the box could be adjusted, but doing this would allow us to search arbitrarily large mazes as we only care about a constant sized subset of it. However, this will come at the cost of accuracy, as it is possible there is a better global path we cannot find since we only focus on a local area.