CS 440: Introduction to Artificial Intelligence **Rutgers: Spring 2021**

Project #1: This Maze is on *Fire*

February 20, 2021

Names: Aamna Farooq (af704), Nada Elshamaa(nhe12), and Asma Makhdoom(aam355)

Problem 1. Write an algorithm for generating a maze with a given dimension and obstacle density p.

Solution. Please see generateMaze function within generateMaze.py in the code.

Problem 2. Write a DFS algorithm that takes a maze and two locations within it, and determines whether one is reachable from the other. Why is DFS a better choice than BFS here? For as large a dimension as your system can handle, generate a plot of 'obstacle density p' vs 'probability that S can be reached from G'

Solution.
If you want an algorithm that takes a maze and two locations within it, and determine whether one is reachable from the other DFS is a better choice than BFS because in this algorithm the optimal path does not matter. The purpose of BFS is to find an optimal path but since we don't care about that we should choose the algorithm that uses less compute power, which is DFS.

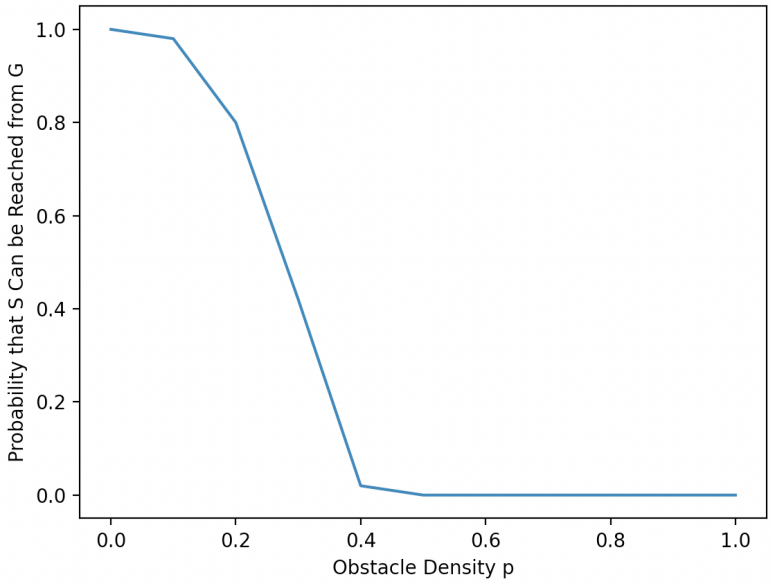

Figure 1: Plot using a dimension of 100, average of 50, 0<p<1 with step = 0.1

Problem 3. Write BFS and $A*$ algorithms (using the euclidean distance metric) that take a maze and determine the shortest path from S to G if one exists. For as large a dimension as your system can handle, generate a plot of the average $'number\ of\ nodes\ explored\ by\ BFS$ - $number\ of\ nodes\ explored\ by\ A*$' vs $'obstacle\ density\ p'$. If there is no path from S to G, what should this difference be?

Solution. If there was no path from S to G then there would be no difference between the number of nodes explored. This is because A* and BFS each would try to exhaust every possible option and would therefore have the same number of nodes explored.

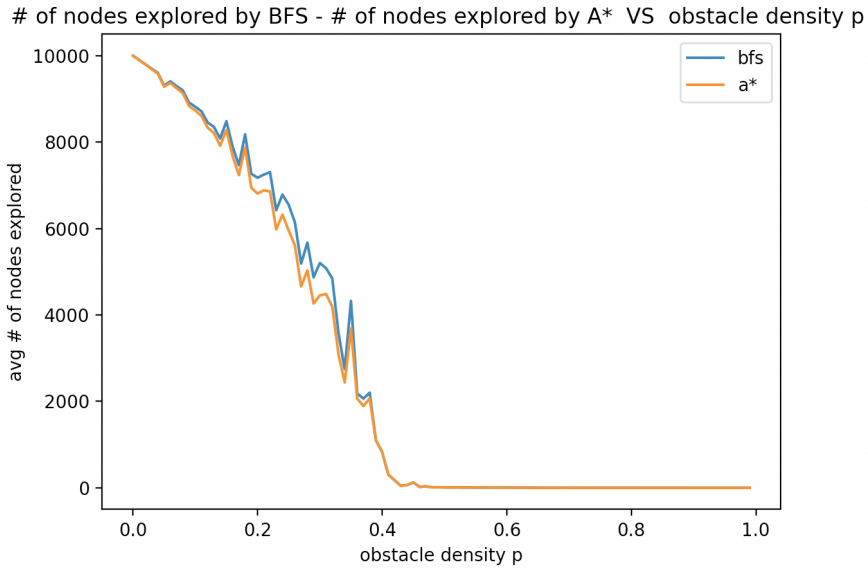

Figure 2: Plot using a dimension of 1000, average of 50, 0<p<1 with step = 0.01

Problem 4. What's the largest dimension you can solve using DFS at p = 0.3 in *less* than a minute? What's the largest dimension you can solve using BFS at p = 0.3 in *less* than a minute? What's the largest dimension you can solve using $A*$ at p = 0:3 in *less* than a minute?

Solution.
Largest dimension we can solve using DFS at p = 0.3 in *less* than a minute: 4480
Largest dimension we can solve using BFS at p = 0.3 in *less* than a minute: 1680
Largest dimension we can solve using $A*$ at p = 0.3 in *less* than a minute: 2500

Problem 5. Describe your improved Strategy 3. How does it account for the unknown future?

Solution. Our Strategy 3 algorithm SPOOFFs (Simulates Probability Of Our Future Fire) because TGIF(This Grid Is on Fire). It tries to account for unknown future by trying to predict the path of the fire. We do this in 2 ways.

1. We employ a deterministic approach where we assume the fire will spread 100% of the time to all nodes it could possibly spread to for each time step. Based on this we generate a 3-D maze which has a maze for each time step. The size of the 3-D maze is dependent on the distance from the current position to the fire which is calculated using Manhattan distance. Based on the number of time steps we traverse through the 3-D maze and try to get as far as we can with a deterministic approach.
If this deterministic approach is unable to reach the goal, we use 2.

2. We employ a probabilistic approach where we assume where the fire will spread by generating an average of 10 possible mazes for each time step with the actual value of q. The average maze holds a probability for each cell catching on fire. Using each resulting average maze we generate a 3-D maze that holds an average maze for each time step, the number of time steps depending upon the distance of the current position on the path from the fire. From the last position of the path we traverse through the maze, choosing the cell that has the least probability of catching on fire at each time step.
Once we have traversed the length of the time steps calculated using the distance from the fire, if we have not reached the goal we revert to the deterministic approach once again.

Our strategy 3 is done using $A*$. $A*$ typically uses a heuristic value that is derived from calculating the euclidean distance of the current position from the goal and adding that to the traversed distance. This value is appended to a priority queue, where values are then chosen and dequeued based on their priority (smaller distances have higher priority). We altered this algorithm for strategy 3 by instead including a priority value that consisted of a tuple of 2 values. Our first value in the tuple consisted of the probability of that cell catching on fire, as calculated in our 3-D maze, summed with the probability of fire on that path so far. The second value in the tuple was of the distance heuristic (euclidean) that is typically used in $A*$. This tuple sorts the queue in ascending based on the first value and sub-sorts in ascending order based on the second value. We employed the use of a tuple in our priority queue because we wanted to prioritize survivability over getting the shortest possible path.

To optimize our algorithm and to preserve compute power we performed a DFS algorithm on our initial maze before the fire has spread (time step = 0) to ensure that finish is reachable from start initially. Otherwise, it returns the maze because there is no possible path to finish.

Problem 6. Plot, for Strategy 1, 2, and 3, a graph of 'average strategy success rate' vs 'flammability q' at p = 0.3. Where do the different strategies perform the same? Where do they perform differently? Why?

Solution.

No Figure Yet

Figure 3:

Strategy 1, Using a dimension of 100, average of 50, p=0.3, q with step of 0.1.

Strategy 2, Using a dimension of 100, average of 50, p=0.3, q with step of 0.1.

Strategy 3, Using a dimension of 10, average of 50, p=0.3, q with step of 0.1.

Strategies 1, 2, and 3 perform the same from p = 0 to p = 0.2. Strategy 3 spikes up around p = 0.3. Strategy 1 has a higher value of success compared to Strategy 2 for p = 0.4. Strategy 3 has the highest success value at p = 0.6 and p = 0.9 compared to Strategies 1 and 2. This is because Strategy 3 should perform better than Strategies 1 and 2 with higher p values.

Problem 7. If you had unlimited computational resources at your disposal, how could you improve on Strategy 3?

Solution.
If we had unlimited computational resources we wouldn't need to rely on a deterministic approach to get a path. The purpose of our deterministic approach was to find a path from start to finish in less time because

if there exists a path when q=1, that path exists for all other values of q. Using the deterministic approach in our algorithm reduces our computational power because using this approach does not require calculating an average of mazes. Therefore, we would completely rely on the probabilistic approach in order to find the path from start to finish. The probabilistic approach is a more accurate representation of the maze because it uses the real value of q instead of q=1 in order to sample what the fire looks like in future time steps.

Another approach we could take if we had unlimited computational resources is to calculate the state of the fire as far into the future as possible (i.e. until the goal can be reached). In our current algorithm, we calculate the number of time steps to look ahead in the future by how far we are from the fire currently. This caps how far we look into the future but if we had unlimited computational resources we would not need to place a limit on the number of steps to look ahead.

Lastly, we would want to increase the number of samples we use to average each probability maze for each time step. Currently, we are only sampling 10 mazes for each time step. By gathering more samples, we will have a more accurate picture of what the fire could look like at that time step.

Problem 8. If you could only take ten seconds between moves (rather than doing as much computation as you like), how would that change your strategy? Describe such a potential Strategy 4.

Solution.
If we can only take 10 seconds between moves we would want to reduce the compute time our algorithm is taking.

Assuming the fire advances one step every 10 seconds, we would compute a path using BFS for as many neighboring layers we could reach within 5 seconds. From the last set of neighbors, we would compute the Manhattan distance from the goal and choose the path whose final layer leaves it closest to the goal.

We would repeat this code until we either reach the goal, cannot move further due to a blockage, or the goal is on fire. We are assigning 5 seconds to get as far as we can with our path because we anticipate that calculating the Manhattan distance and path of each neighbor will have a time complexity of less than or equal to 5 seconds.

Work Distribution
The work is our own and not copied or taken from any other students.

To work on this project we would meet up over video calls daily and discuss problems and our solutions. One person would then screen share and code while the others would contribute and also assist in coding using the request remote control feature in zoom. We would alternate in screen sharing and upload to git for version control.

The report was done similarly. We each took on a plot and a question to complete on our own. We then met to complete the rest as a group over video call.
Asma Makhdoom : Problem 2
Aamna Farooq : Problem 3
Nada Elshamaa : Problem 4 and Problem 6