CS 440: Introduction to Artificial Intelligence

Project 1: This Maze is on Fire

Atharv Kulkarni, Niyati Patel , Rishika Sakhuja
Professor Cowan
February 19, 2021

* Multi-Section Group: Atharv (section 3), Niyati (section 1), Rishika (section 1). Submitted by Atharv*

Introduction

The objective of this project is to find the shortest path that will allow an agent to go from the upper left corner to the lower right corner in a square maze where some cells are blocked off. The agent can only travel in the up, down, right, or left direction. We implemented the DFS, BFS, and A* algorithms and tested all the different strategies to find the shortest path that the agent could take.

Problem 1

To create a maze with a dimension and obstacle density, p, we created a function called mazeGen with the parameters size and density. We start off by defining a maze which is a 2d array where the column length and the row length are equal to size and all the values in the 2d array are zero. Then, we implement a nested for loop to fill the maze with either ones or zeroes using the random.uniform() function. For each position in the 2d array, we will get a floating-point number between zero and one. If the value that we get for the position is greater than the obstacle density, then we set that cell value to one. Otherwise, it remains zero.

While some of the cells in this maze are blocked, there is also one cell that is on fire when we first generate a maze. To randomly put one cell on fire, we created a variable called count, which we initialized as one. The purpose of this variable is to make sure that we only set one cell on fire. We go through a while loop where we randomly generate two values, i and j, which are between zero and size minus one. Then, we check the maze array to make sure that the i,j position is not the upper left corner or the lower right corner and that the value at the i,j position is not one. If it satisfies all of these conditions, then we can set the value at that position to five and we exit the while loop. If it does not satisfy these conditions, then we check to see if the value of count is equivalent to twice the size. If it is, then we exit the while loop, otherwise, we increment the value of count and continue to go through the while loop until we set one cell position in the maze on fire.

Problem 2

The DFS algorithm that we implemented has the parameters maze, the initial state, the goal state, and the size of the maze. We create an array called fringe which is where we will store all of the cell positions that we can access. The first position that we append in the fringe is the initial state. We create a new matrix, prev, which is what we will use to find the shortest path from the initial state to the goal state. At the initial position in the prev matrix, we set the value to NONE. We also need to keep track of the visited cell positions so we defined a new set called closed.

We created a while loop and the only time we would enter the while loop is if the fringe was not empty. The first step in the while loop was to pop the first value in the fringe and set that equal to current. Initially, the first value in the fringe is the initial state. We started off by checking to make sure that the current value is not equivalent to the goal state. If this were the case, then we would exit the while loop and return the length of the path. If the current value is not equal to the goal state, we have to check the neighbors of the cell. We also have to take into consideration that not all of the cells have four neighbors. The order that we checked the neighbors in was the left neighboring cell, neighboring cell above it, right neighboring cell, and the neighboring cell below it. For all of the neighbors of the cell, we have to make sure that it is within the bounds and has not already been visited. If both of these conditions were satisfied, we then checked to see if the value of the neighboring cell is zero. If the cell was zero, we were able to store the current value in the prev matrix. This value would be stored in the position of the neighboring cell that we are checking in prev. We then added the neighboring cell to the fringe. However, if both of the initial conditions were not satisfied, we would not go through the if statements and would continue to go on to check the next neighboring cell. Once again, we made sure that the cell was within the valid bounds and that it already had not been visited and repeated the same steps. We did this for the cell to the right of the current cell and then the cell below the current cell. Once we checked all of the neighboring cells of the current cells and added all of the valid neighbors to the fringe, we added the current cell to the closed set. We continued to enter the while loop until it was not empty or until the value that was popped from the fringe was equivalent to the goal position.

The purpose of this DFS algorithm is to determine whether or not it is possible to find a path between an initial state and the goal state. In this algorithm, we start at the initial state and go through the maze as far as we can before we have to backtrack. With BFS, we explore all of the neighboring states before we start again at the next cell. Also, with BFS, we are finding the shortest possible path while, with DFS, we are just checking to make sure that a path exists between the initial state and the goal state which means that it will take up less memory since it only has to keep track of the cells that are in the path, therefore, having more of an advantage over BFS.

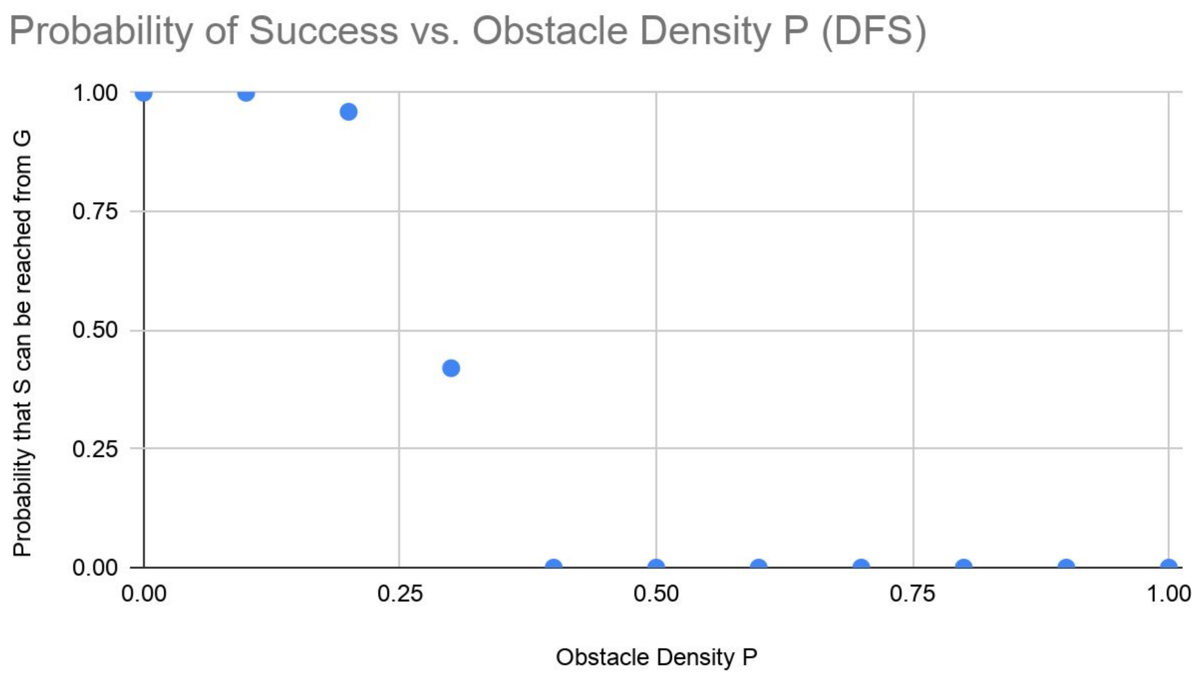
The size of the maze that we will use for generating a plot is 1000 x 1000.

Probabilities for Depth First Search

Obstacle Density P	**Probability that S can be reached from G**
0.0	Success: 50 Fails: 0 Probability: 50/50 = 1
0.1	Success: 50 Fails: 0 Probability: 50/50 = 1

0.2	Success: 48 Fails: 2 Probability: 48/50 = 0.96
0.3	Success: 21 Fails: 29 Probability: 21/50 = 0.42
0.4	Success: 0 Fails: 50 Probability: 0/50= 0
0.5	Success: 0 Fails: 50 Probability: 0/50= 0
0.6	Success: 0 Fails: 50 Probability: 0/50= 0
0.7	Success: 0 Fails: 50 Probability: 0/50= 0
0.8	Success: 0 Fails: 50 Probability: 0/50= 0
0.9	Success: 0 Fails: 50 Probability: 0/50= 0
1.0	Success: 0 Fails: 50 Probability: 0/50= 0

Graph



Problem 3

The BFS algorithm is similar to the DFS algorithm, however, the difference is that instead of using fringe.pop(), we use fringe.pop(0). This makes it so instead of popping the first newest entry in the fringe, we pop the first entry in it- thus transforming the fringe into a queue. In addition, there is the difference within the while loop that checks for whether or not the fringe is empty, we have an additional if statement prior to checking the neighbors of the current cell. We check to see if the current cell has already been visited. The reason that the if statement is needed is because in BFS there are occasions where a node is added to the fringe even though that node is already in the fringe. This can happen even though the node is in the fringe, it has not been popped yet and added to the closed set. Thus this additional check is required as to not balloon the runtime by including nodes already explored.

In order to develop the a* algorithm, we passed the parameters maze, initial, goal, size, and heuristic. The heuristic input is only used in strategy 3 and should be None in situations pertaining to a*. For the a* algorithm, we defined the fringe as a priority queue, importing PriorityQueue() from the queue library. From there, we followed a very similar outline as laid out by Professor Cowen in his notes detailing uniform cost search. The values for distance, processed and previous were stored in one array with each node holding a 3-tuple holding these values. All values in this matrix are initialized as (∞, 0, None), marking a distance of ∞, the node not being processed and there being no previous node. The heuristic used is the euclidean distance from the goal. We then proceed to move through the algorithm by adding the initial node to the fringe along with its distance+heuristic and heuristic value to allow easy subtraction. If the node popped is not the goal, then we check if it processed. If it isn't, we check to see if the neighbors of that node are eligible to be added to the fringe, and update the neighbors distDonePrev values, as well as the current nodes disDonePrev values. If the goal node is reached, then 1 is returned as well as the path, its length and the total nodes explored. Otherwise, if the fringe is empty, we know that there is no path and return 0 and the total nodes explored.

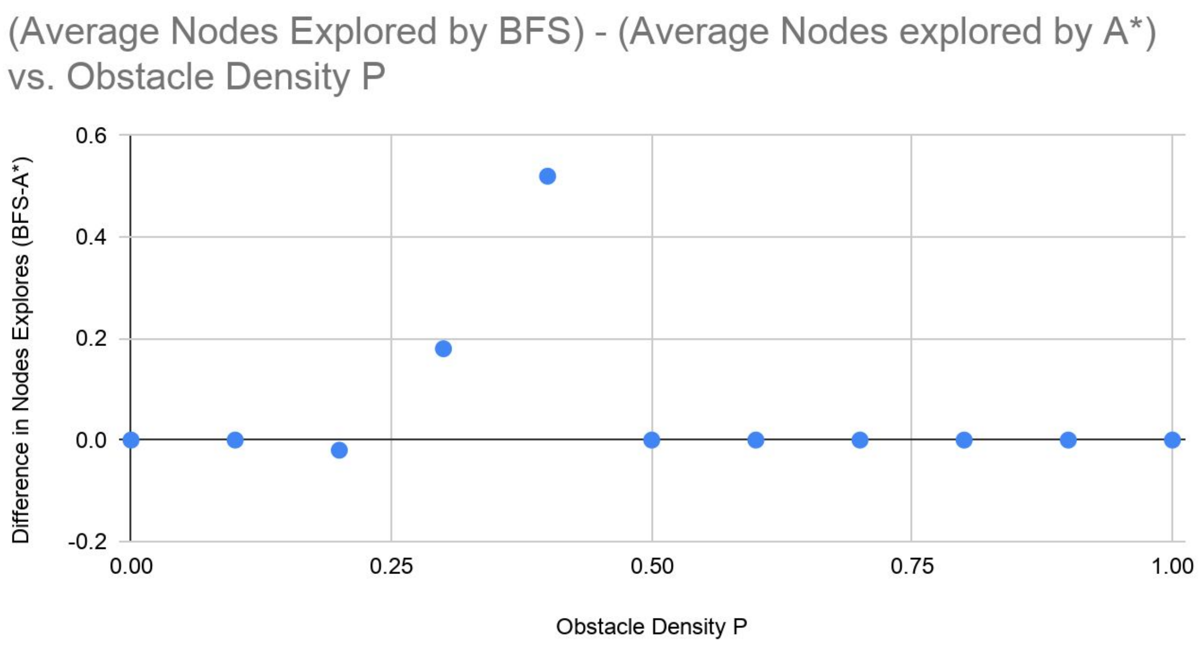
For the test in comparison, we used 50 trials per density value with the size of each maze being 500 by 500. From the graph below, you can see that the differences are very minimal with the average essentially being zero nodes explored. We believe that this is because the weight for each move is just 1 making it so the a* algorithm essentially follows the bfs algorithm in the nodes that it explores. If there is no path, the difference would be zero as the algorithms would explore every node that they can, thus making the difference zero.

Nodes Explored by BFS - Nodes Explored by A*

Obstacle Density P	**(Average Nodes Explored by BFS) - (Average Nodes explored by A*)**
0	0

0.1	0
0.2	-0.01999999999
0.3	0.18
0.4	0.52
0.5	0
0.6	0
0.7	0
0.8	0
0.9	0
1	0

Graph



Problem 4

To find the amount of time it took to execute DFS, BFS, and the A* algorithm, we imported the time library. We defined start_time to time.time() and end_time to time.time(). We then called the DFS, BFS, and A* function and printed end_time - start_time. To find the largest dimension that we could solve using DFS, BFS, and A* under a minute with a probability of 0.3, we had to test multiple values for the size. We tested the amount of time it would take to get from the upper left corner to the lower right corner.

DFS

SIZE	TIME	PATH LENGTH
3000	3.31224608421	11483
6000	11.7254440784	22417
12000	45.3009572029	47881
13500	57.6402368546	59633
13550	60.1279056231	60394

The largest dimension that we can calculate at p = 0.3 in under one minute using the DFS algorithm is between sizes of 13500 and 13550

BFS

SIZE	TIME	PATH LENGTH
100	0.0179319381714	199
1000	1.9086971283	2019
2500	27.9930560589	5003
3010	55.1910860538	6031
3059	56.1002230644	6127
3087	58.0483188629	6177
3096	58.3213250637	6197
3097	59.8123102188	6207
3098	60.486082077	6207

The largest dimension that we can calculate at p = 0.3 in under one minute using the BFS algorithm is of size 3097.

A*

SIZE	TIME	PATH LENGTH
1000	7.56058502197	2003
2000	34.013406992	4015
2450	56.5352418423	4923
2477	57.7005240917	4981
2481	57.773786068	4991
2500	59.3077220917	5033
2510	61.3213231564	5049

The largest dimension that we can calculate at p = 0.3 in under one minute using the a* algorithm is of size 2500.

Problem 5

For our third strategy we created a function called strategy3 that has the parameters maze, initial, goal, size, and fire. First we create a pointer called pntr that is set equal to initial. Next we create a matrix with all values initialized to 0 called probMat to store the probabilities of each cell to catch on fire. We then made a copy of the maze, and simulated 5 fire steps on the copied maze. To populate the probMat matrix we traverse through the matrix one cell at a time and check if the matrix is on fire or blocked. For each cell we check all of its four neighbors to calculate the probability of the cell catching on fire and store it in the probability matrix. If the cell is already on fire we set the cell value of the probability matrix equal to1.

After finding the probabilities, we run the aStar with probability matrix for the heuristic input to find the shortest path to the goal state. While this is no longer aStar, we believe that doing so would lead us through the path of highest survivability. We check if there is a possible path till the goals state if not we return 0 indicating failure. We then run advanceFire using the initial maze as the input. We set the pointer to the first position in the path and check if the cell is on fire. If the cell is on fire we return 0 indicating failure and if the cell is the goal state we return 1 indicating success.

If the goal state is not reached repeat the above steps of creating a probability matrix, finding the shortest path and checking the status of the current cell until we either reach the goal state or there is no path to goal state.

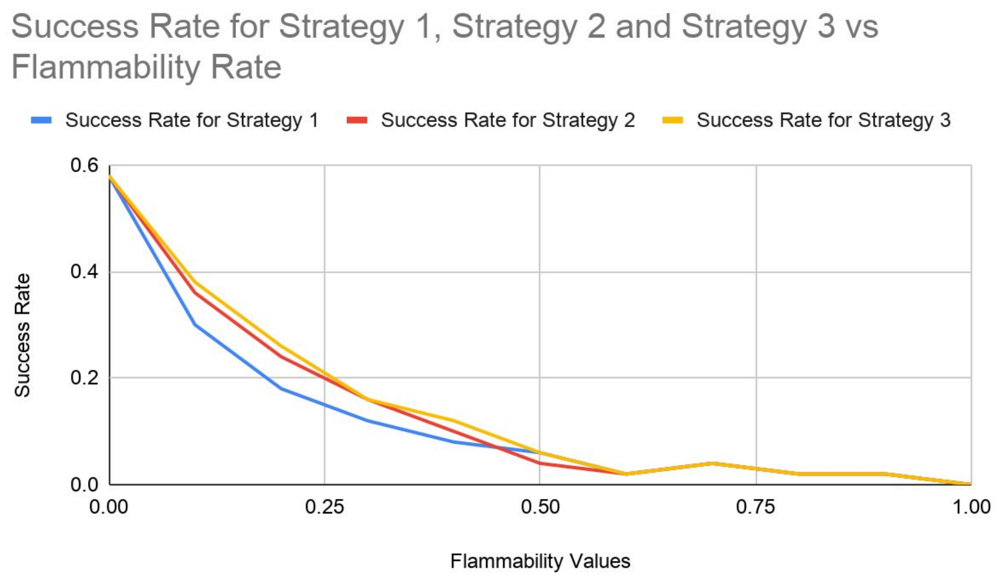
This strategy accounts for the future by finding the probability of each cell catching fire every one time step and then running aStar with the probability matrix as the heuristic input to find the shortest path.

Problem 6

Because of the time complexities of the strategies coupled with the fact that we ran 50 trials per flammability value for 11 flammability values, we decreased the size of the maze into a 50 by 50 maze.

Flammability Values	Success Rate for Strategy 1	Success Rate for Strategy 2	Success Rate for Strategy 3
0	0.58	0.58	0.58
0.1	0.3	0.36	0.38
0.2	0.18	0.24	0.26
0.3	0.12	0.16	0.16
0.4	0.08	0.1	0.12
0.5	0.06	0.04	0.06
0.6	0.02	0.02	0.02
0.7	0.04	0.04	0.04
0.8	0.02	0.02	0.02
0.9	0.02	0.02	0.02
1	0	0	0

Chart



The above plot is the graph of the average success rates of the each strategies vs the flammability, q. The blue graph represents the rate for strategy 1, orange graph represents the rate for strategy 2, and yellow represents the rate for strategy 3. When the flammability is greater than 0.5, we can see that some strategies have a higher success rate than others. From our plot above, we see that strategy 3 is the most efficient strategy when the flammability is smaller. We can see that the graphs converge to similar values as we increase the flammability, q. The success rate for each strategy starts equally the same after the flammability value is greater than 0.5. This is because after a certain point the fire spread so quickly which makes it harder and harder to find a path to the goal state.This eventually makes it impossible for any strategy to find an optimal path from the initial state to the goal state.

Problem 7

Currently our strategy 3 only calculates probMat for only five steps ahead. If we had unlimited computational resources, we would implement a method that could calculate simulated probabilities for more than five steps ahead. Ideally, we can plan many steps in advance by simulating many fire steps and then use the simulated maze to calculate the simulated probabilities several steps in the future. Using these simulated probabilities, we should be able theoretically find the safest path using our strategy. We would just keep performing the calculations for each run. However, as we do not have the computational capabilities to do this, we only ran the strategy for the next predicted fire step and thus it was not as effective as we hoped. One notable criticism of our strategy is that at really high flammability values, it essentially boils down to bfs. However, based on the graph as seen above, all the strategies converge on to the same values at high flammability anyways, so it does not seem to be too significant.

Problem 8

If we only had a limited amount of time between moves a possible fourth strategy would be similar to our strategy 3 but would predict less moves in advance as to save time. Our current implementation of strategy 3 has the ability for easy changing of how many moves in advance you want to simulate. Furthermore, as seen above, DFS was able to solve mazes significantly faster than the other algorithms. Thus, as an ideal path is not needed and time is the main constraint, we could mix strategy 3 with DFS by simply simulating a few steps in advance and then performing DFS on the simulated maze to find a path to the goal. Then, we can advance the fire on the real maze and then resimulate a few fire steps in advance and continue. If even less time is needed, we could switch to a local search algorithm, however this is not ideal as there is no guarantee that it doesn't simply get stuck.

Contributions

- Atharv Kulkarni
    - Exercise 1 :Maze Generation Algorithm

    - Exercise 3: aStar Algorithm
    - Exercise 2: DFS Algorithm
    - Strategy 2 Algorithm
- Niyati Patel
    - Exercise 5: Strategy 3 Algorithm
    - Exercise 7
    - Exercise 8
    - Exercise 6: Strategy 3 Test
    - advanceFire Algorithm
- Rishika Sakhuja
    - Exercise 3: BFS Algorithm
    - Strategy 1 Algorithm
    - Exercise 4
    - Exercise 6: Strategy 1&2 Test

Honor Statement:

On our honor, we have neither received nor given any unauthorized assistance on this assignment. This is our own work and not copied or taken from online or any other student's work. 100% this assignment was done entirely by Atharv Kulkarni, Niyati Patel, and Rishika Sakhuja. All code was collectively thought up by us.