Introduction to Artificial Intelligence

Project 1: Maze on Fire

Brenton Bongcaron and Abe Vitangcol
NetIDs: bdb101 and alv88

February 19, 2021

1 Maze Generation

The premise of the project is an agent being trapped in a maze. They start at the top left of the maze and need to get to the very bottom right of the maze. To create the maze environment, we created a function called buildMaze within maze.py which makes the maze in the form of a matrix.

Building the maze needs only a few requirements: the size of the maze (dim), and the obstacle density of the maze (p). The obstacle density is between 0 and 1, exclusive, and our code is ready to output an error message should a value be inputed outside that range. The maze is generated as a matrix with all of its entries, from (0,0) to (dim - 1, dim - 1), are 1. Then, going through each tile one by one using a nested for loop, we used random() as a way to randomize the maze, and if the value obtained from random() was less than or equal to the obstacle density, it became the obstacle, which means the value of the matrix at that coordinate was 0. When it finishes going through all of the tiles of the maze, we need to make sure that (0,0) and (dim - 1, dim - 1) are not obstacles, as they serve as the start and goal spaces, respectfully. So, we simply force these two spaces to be 1 (non-obstacle spaces) so the agent can be loaded in and is able to walk on top of the goal space. The other part of this code is for the fire, which simply goes through the entire maze again until it has placed a space for the fire. If the space is a fire, then the maze at that point will equal two. Like the obstacles, if it was created on the start or goal spaces, it will be overwritten for a non-obstacle space.

2 Finding the path to the goal: DFS

For the first search algorithm, we were tasked at creating a Depth First Search algorithm and use it to help the agent navigate through the maze. To help us with creating this algorithm, we defined a helper function that says whether a move is valid or not, which we can use for both the DFS function as well as future functions too.

Simply called isValid, located within maze.py, it takes in the current maze layout and the coordinate of interest. If the coordinate takes the agent out of bounds of the maze, then it returns false. If the coordinate of the maze is a free space, meaning the coordinate at that matrix location equals 1, then it returns true. There is a third false for fire scenarios, as the second if statement will be skipped as the fire tile is resembled as a 2 on the matrix, not a 1.

Using this, we created the DFS algorithm in maze.py and named it DFS.

This DFS algorithm requires the current layout of the maze, which is given as the parameter maze. The other three variables are default variables in which if a value is not specified upon calling the DFS function, it simply sets the start as (0,0), the target as nothing, and an empty list for the spacesTraveled. For the start and target, this is for any scenario in which the start and goal are not the typical (0,0) and (dim - 1, dim - 1), allowing flexibility for randomized spaces, if we want to. The spacesTraveled default variable is there for when the fire starts to spread and helps to with visualization when we run simulate.py or render.py. The DFS algorithm examines the neighbors of the agent's node, with the agent's node called (currentRow, currentCol) in our code. The algorithm starts by pushing the current position of the agent into the stack, then later pops it, examining the popped node and seeing if the space is the goal. If so, it reports the time (currently commented out to save space in our command prompt) and returns the list of nodes the agent traveled through. If the popped node is not the goal space (always not so long as the dim value is greater than 1, which it always is), then it starts to check valid neighbors. It checks the north neighbor (called up child in our comments), thne west neighbor (called left child), next the south neighbor (called down child), and finally the east neighbor (called the right child). They are pushed into the stack in this specific order as to prioritize going towards the goal (going either down or right) upon popping the next node. It constantly repeats this until it either has exhausted all nodes and hasn't gotten to the goal or until it reaches the goal.

Below represents a graph of the obstacle density vs the probability that S can be reached from G using our DFS algorithm and a maze size of

While many times a Breadth First Search (BFS) would find an optimal path in getting to the goal, DFS is actually better in a situation like this simply because we already know where the goal is from the start. By simply knowing where the goal is at the beginning of the run, we can simply beeline over to that destination by going down one whole branch to find the goal instead of going through all of the layers of a BFS tree just to eventually see the goal at the end of one of the branches. Simply put, using DFS in this maze is faster than BFS as BFS is always going a worst-case scenario in having to check majority of the nodes to find the goal, sometimes taking about 20 times longer than DFS. If the goal was randomized each run, then BFS would be a better pick as the goal would be somewhere in the middle of its tree rather than at the bottom while DFS would have to search entire branches one at a time to find the goal node somewhere in the middle.

3 BFS and A* Implementation

After completing the DFS search algorithm, it acted as a skeleton for other search algorithms, like BFS and A*, simply because all of them go through the same process, but they just have a different fringe or organizational method to proceed through the maze. Both BFS and A* take in the same parameters as DFS, minus some unnecessary parameters such as the target or spacedTraveled. And they work relatively similarly to DFS.

As seen here, the BFS code has a similar structure to DFS. The main difference is how the fringe is used. In DFS, the fringe acted as a stack, having the nodes being pushed in and popped. In BFS, the fringe acts as a queue. When a node is popped (doing fringe.pop(0) is the same as dequeueing), it is checked to see if it is the goal node or not. If so, it does what DFS does with little changes. If not, it checks the node's neighbors. In terms of the priority of searching its neighbors, it does right, then down, then left, then up, appending each one after checking if it's valid. It constantly repeats appending and popping(0) the fringe until it finds the goal, which takes significantly more time than DFS, even if it finds the most optimal path to the goal.

The A* code (called aStar cause function declaration doesn't like asterisks) still maintains the structure of DFS. Like BFS, A*'s fringe performs differently as now it acts as a priority queue instead of a queue or a stack. Unlike the other two search algorithms, A* includes another array that records the cost and helps with doing priority queue actions. At first, when A* is called, it has a cost of 0 in distances (the helper priority queue) and the start node in the fringeNodes. A* finds the minimum in distances (done by doing min(distances)) and stores that value in the variable name lowestDistance. Next, we find the index at which this lowestDistance occurs in distances (by doing distances.index(lowestDistance)) and store that information in a variable called index. After obtaining the index of the lowest cost, we pop both priority queues at that index, with the value from fringeNodes being stored as (currentRow, currentCol) and the value from distances simply being lost. Now that a node has been obtained, it goes through the same procedure as BFS and DFS, where it checks if the current node is the goal or not, reporting similar things should it be the goal node. When it checks for its neighbors, order doesn't technically matter, but we have it set to the same order as BFS. To calculate the distances, we simply use euclidean distance from the node in question (the right / left / down / up child) to the goal and record it in a variable called nodeDistance. Once calculated, we append the node in question to fringeNodes and append nodeDistance to distances at the same time in order to keep both of these items at the same index. After checking all the valid neighbors, A* goes back to the top and finds the minimum distance, finds its index, pops both the node at that index in fringeNodes and in distances, and goes back to checking neighbors again. This repeats until it reaches the goal and take much less time than BFS, although it does explore more nodes than it.

4 Time is Everything

5 Problem 5

Answer here

6 Problem 6

Answer here

7 Problem 7

Answer here

8 Problem 8

Answer here

9 Notable Information

Additional things / for fun thing go here