CS 440

PROFESSOR WES COWAN

Fire Maze

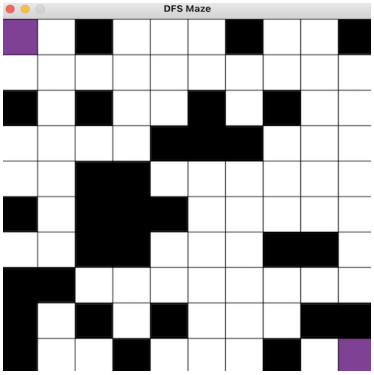
FEBRUARY 19, 2021

NABHANYA NEB (NN291)

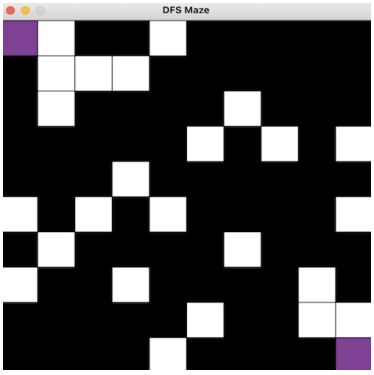
ARPITA RAY (AR1516)

I. Maze Introduction

Our code generates a 2D array of a state object that we defined. It stores the x coordinate, the y coordinate, the value, and the previous state visited. The value attribute is initialized as 0. We loop through the array, and with each iteration we generate a random number between 0 and 1. If this number is $\leq$ p, then that state becomes restricted, and we set the value attribute of the current state to 1. Thus, when finding paths in our maze, any state whose value attribute equals 1 is considered restricted, and cannot be used. The initial state and the goal state both always have a value of 0. The previous attribute is used to determine the final path in the end for each algorithm.



Obstacle Density: 0.3

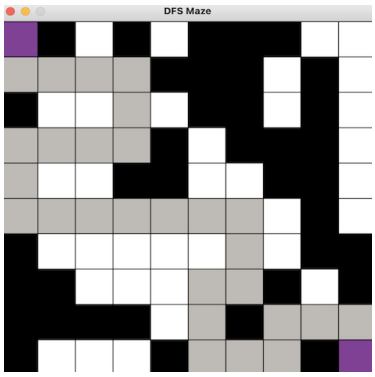


Obstacle Density: 0.7

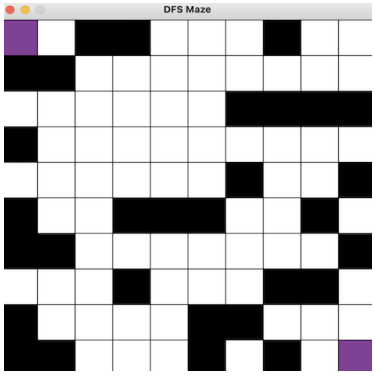
II. Depth First Search

The first search algorithm we implemented is the Depth First Search, otherwise known as DFS. DFS does essentially what its name says. It starts at the top and progresses downwards, following a path until the goal state is found, or it has reached the bottom. If the goal state is found, it ends the search. Otherwise, it backtracks to the top-most starting node, and repeats this process.

In our program, the DFS function takes in the dimension, starting state, the goal state, and the maze itself as parameters. We create a stack, the data structure we actually use is a LIFO Queue, which is referred to as the fringe, and push the starting state onto this stack. We create a list to keep track of the visited nodes, and we add the starting state to this list as well. Next, we initialize a while loop that continues until the fringe is empty. The most recent node is popped off the fringe, and added to visited. If the x and y coordinates of this state is equal to that of the goal state, then a path has been found and we break out of the while loop. Else, we find all valid and unvisited neighbors of this state, and push them to the fringe, and iterate the loop again. If a path was found, we use a pointer to backtrack to the starting node, and print out the final path.



DFS: Path Found

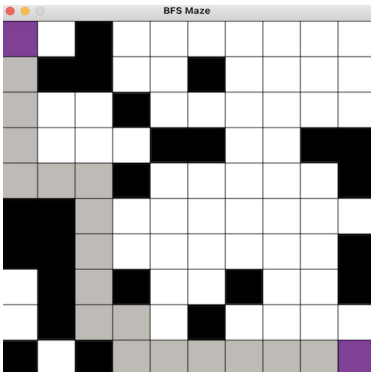


DFS: No Path

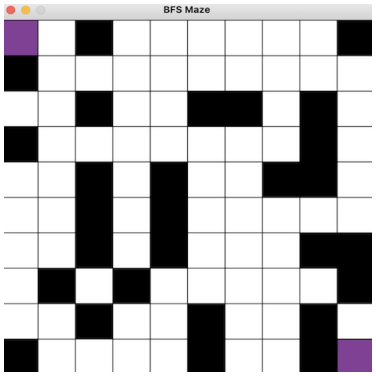
III. Breadth First Search

Breadth First Search, or BFS, is the next algorithm we implemented. BFS explores all children of a level from left to right before moving to the next level. BFS ends when the goal state has been found, or all the nodes have been visited.

Similarly to DFS, our BFS function takes in the dimension, starting state, the goal state, and the maze itself as parameters. The biggest difference is that BFS uses a queue, as opposed to a stack, as the fringe. We enqueue the starting state, create a list for visited nodes, and add the starting state to this list. Then, we initialize a while loop that continues until the fringe is empty. The oldest node is dequeued from the fringe, and added to the visited list. If the x and y coordinates of this state match that of the goal state, then a path has been found and we break out of the while loop. Else, we find all valid and unvisited neighbors of this state, and enqueue them to the fringe, and iterate the loop again. If a path was found, we use a pointer to backtrack to the starting node, and print out the final path.



BFS: Path Found

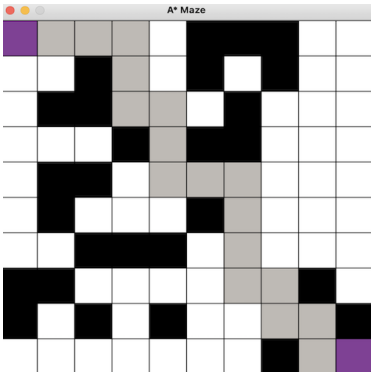


BFS: No Path

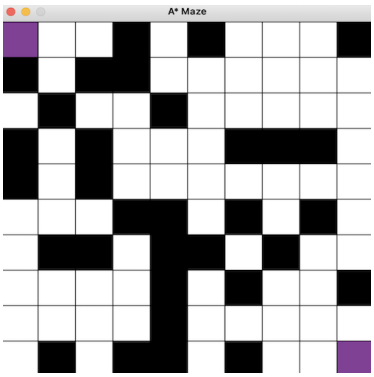
IV. A*

Next, we implemented the A* algorithm. A* uses heuristics to determine the path. It prunes out sections of the maze to visit the least possible nodes. Heuristic calculations and total steps taken are used to calculate the priority.

Similarly to DFS and BFS, our A* function takes in the dimension, starting state, the goal state, and the maze itself as parameters. The main difference is that A* uses a priority queue. The data structure we chose to code this use is a heapq, as opposed to a stack or standard queue, as the fringe. We push the starting state, create a list for visited nodes, and add the starting state to this list. Then, we initialize a while loop that continues until the fringe is empty. The node with lowest priority is popped off the fringe, and added to the visited list. If the x and y coordinates of this state match that of the goal state, then a path has been found and we break out of the while loop. Else, we find all valid and unvisited neighbors of this state, and push them to the fringe, with their heuristic added to the steps taken as the priority. The heuristic is the distance between the current state and the goal. If a path was found, we use a pointer to backtrack to the starting node, and print out the final path.



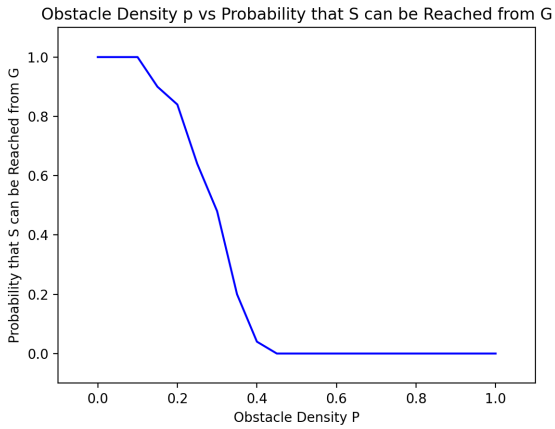
A*: Path Found



A*: No Path

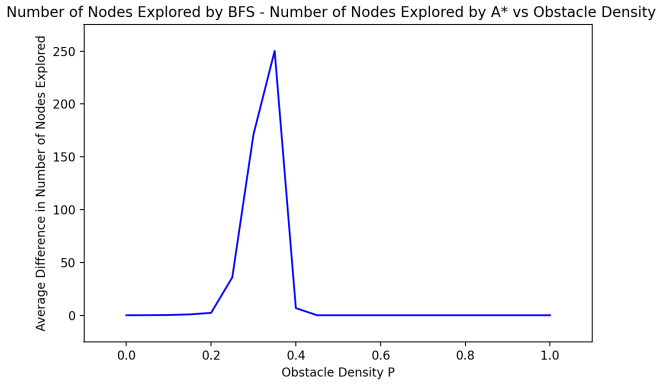
V. Static Maze Questions

- DFS is better than BFS to determine whether one location in the maze is reachable from another. When deciding what algorithm to use, it is important to consider time and space complexity. Though they technically have the same Big O, DFS is usually faster. Big O represents the worst-case scenario, not all scenarios. DFS does not explore as many states as BFS, because DFS has to find any path, while BFS always finds the shortest path. Additionally, DFS has a space complexity of $O(b * k)$, where b represents the maximum number of neighbors a State can have, and k represents the dimension of the maze. This is much better than the Big O of BFS, which is $O(b^k)$.



Plot 1: Represents the success rate of finding a path from S to G at dimension = 100. When the obstacle density increases, the success rate of finding a path sharply decreases.

- The plot below represents the average difference in nodes explored by BFS and A* for varying obstacle densities. If there is no path from S to G, this difference should be 0.



Plot 2: Represents the success rate of finding a path from S to G at dimension = 100. As the obstacle density increases, the success rate of finding a path sharply decreases.

- BFS returns the shortest path, A* is always the same length or longer as the BFS path, and DFS is the same or longer as the A* and BFS paths.

- The largest dimension that our system can handle in under a minute is 200 for DFS, 165 for BFS, and 180 for A*.

- For a dim x dim maze, there are only dim^2 nodes at max, and since nodes do not pass through the fringe more than once because of the list that keeps track of visited nodes, the fringe's maximum size is dim^2.

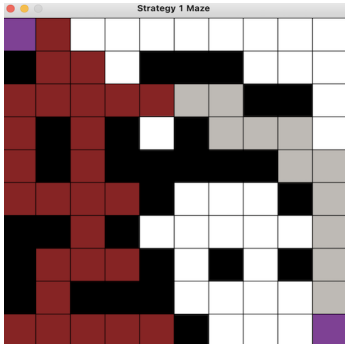
VI. Advancing Fire

Advancing fire for the second part of this project is dependent on flammability rate q. The starting spot for the fire is generated randomly. If a cell has k burning neighbors, the probability that it will be on fire in the next time step is $1 - (1 - q)^k$. We followed the pseudocode to implement this function.

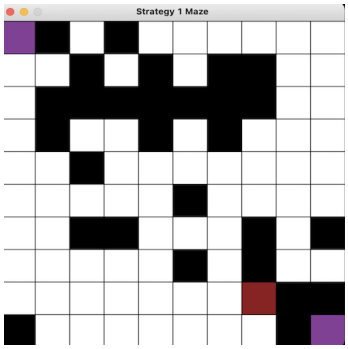
VII. Strategy 1

The first approach, referred to as Strategy 1, calculates the shortest path, using the aforementioned BFS algorithm at the start. It keeps this same path even as the fire is spreading, and just tries to make it to the end is the shortest way possible.

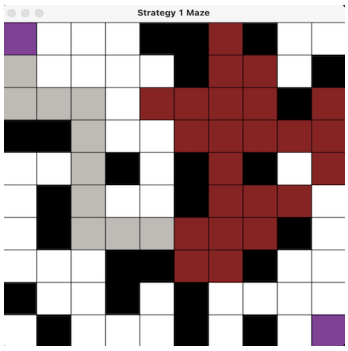
Strategy 1 takes in the dimension, p and q values, initial and goal states, and the maze. It calculates the BFS path, stores this in a list, and returns an empty list if there was no solution found. It loops through the BFS path, adds current position in BFS to the final path, and advances the fire one step. If the current position catches on fire, we state that the agent burned down, and we return the path until this point. Else, this loop continues until it successfully makes it out.



Strategy 1: Path Found



Strategy 1: No Path

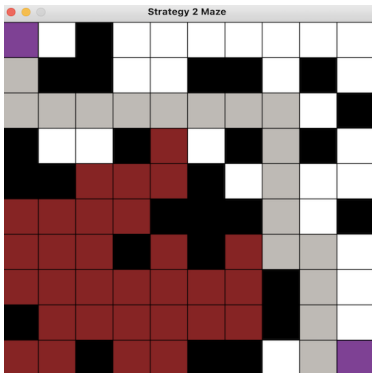


Strategy 1: Burned

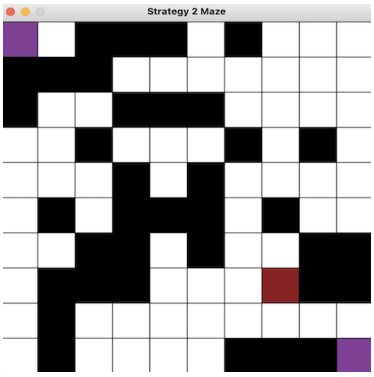
VIII. Strategy 2

Strategy 2 works similarly to Strategy 1. It calculates the BFS path at the beginning and starts going through this.

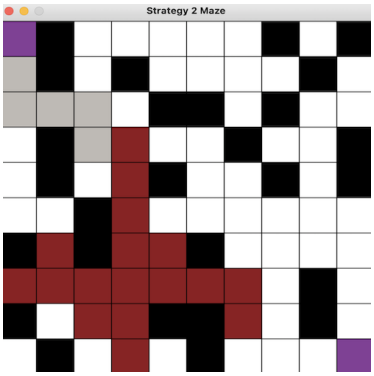
The difference between this Strategy and Strategy 1 is that it recomputes the shortest path with each step to avoid updated fire locations. If there is no solution from the current position to the goal, we state that the agent has gotten blocked by obstacles and fire, and return the steps taken so far. Otherwise, we continue until the agent either burns down or makes it to the goal.



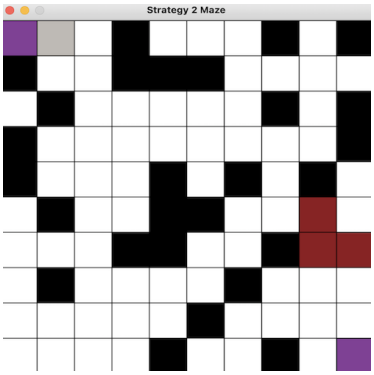
Strategy 2: Path Found



Strategy 2: No Path



Strategy 2: Burned



Strategy 2: Blocked

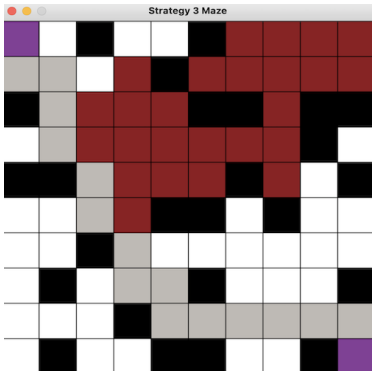
IX. Strategy 3

LATGA stands for Look Ahead to Get Ahead. This is essentially the approach we took to write Strategy 3.

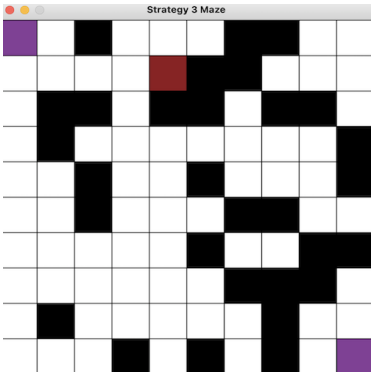
Unlike Strategy 2, LATGA predicts what the fire will look like in the future and recomputes accordingly. Of course, this is just a prediction, but it increases the chances of making it out.

For this approach, we calculate the BFS path, and start iterating through it. With each iteration, we recompute based on the current position of the fire. Then, we create a copy of the current array and call it next, and advance the already advanced fire, to hold what the fire maze might look like in the next iteration. We create a temporary BFS path, and send in the ”next” array instead of the current array, because the ”next” array has more fire locations. If returned path is valid, we update the BFS path to try to ensure that in the next iteration, the agent will not burn down, despite the additional fire. This loop continues until the agent gets blocked, burns down, or makes it out.

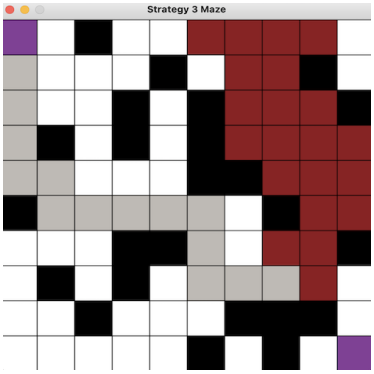
While LATGA increases survivability rate, it takes much longer than the previous two strategies. It has to recalculate two paths each time, as opposed to zero or one. It also advances the fire twice each time. Thus, LATGA has many benefits but also some negatives.



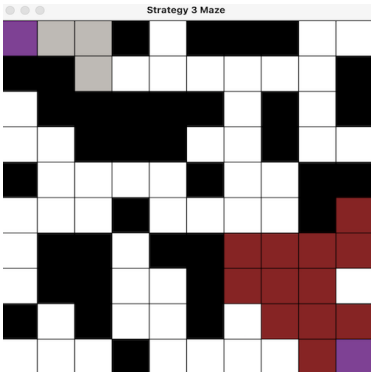
Strategy 3: Path Found



Strategy 3: No Path



Strategy 3: Burned



Strategy 3: Blocked

X. Fire Maze Questions

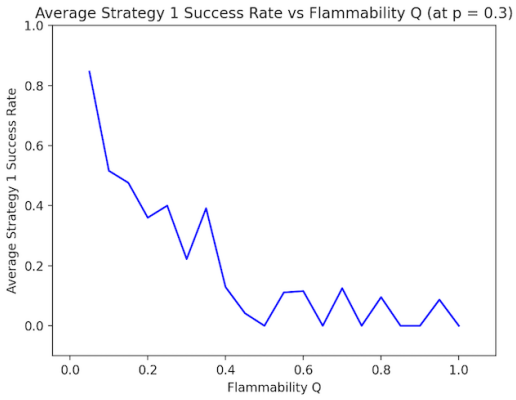
- In terms of success rate, Strategy 3 is the superior strategy, Strategy 2 falls in the middle, and Strategy 1 is the weakest. However, Strategy 1 takes the least amount of time, Strategy 2 is in the middle again, and Strategy 3 takes the longest.

  The three strategies perform similarly at the end. This is because with higher flammability rates, there is a very low chance of escaping the maze without burning down or getting blocked. Regardless of whether the agent recomputes their path or predicts the future location of the fire, it is almost impossible to make it out with the fire spreading so rapidly.
  Strategy 2 and 3 perform similarly from $0.05 \leq q \leq 0.1$, with a nearly perfect success rate. Since the fire is not spreading much at these stages, there is little to no recalculation needed, so both the strategies do the same thing until $q = 0.1$.

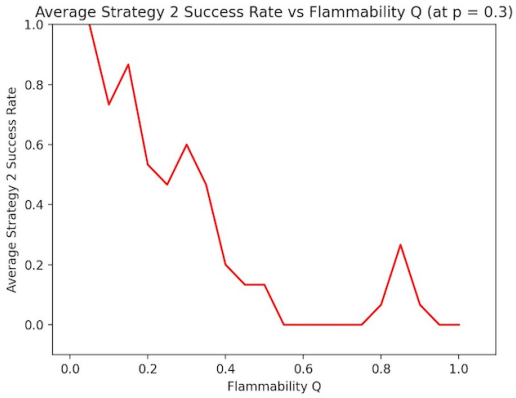
  The main difference in the three mazes how they perform for q values in the middle, from $0.2 \leq q \leq 0.8$. In this range, Strategy 1 has the lowest success rate, never going passed $y = 0.4$. It also has the most peaks and falls, since it is never a guarantee if the initial path, which is the only path calculated, will catch on fire or not. In addition, Strategy 1 starts off much lower than Strategies 2 and 3, despite the low flammability rate.

  In the same range, Strategy 2 reaches a success rate of $y = 0.6$, which is 1.5x better than Strategy 1. This is due to recomputing the path once with every time step. Recomputing clearly helps the agent stay away from the fire much better. Strategy 2 also has a peak at approximately $q = 0.8$.

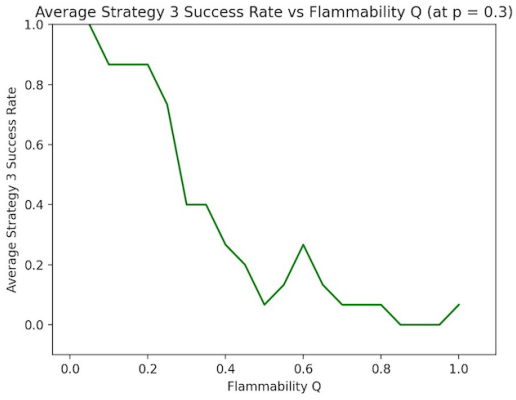
  Finally, within the same range, Strategy 3 reaches a success rate of $y = 0.9$. This is 1.5x better than Strategy 2, and 2.25x better than Strategy 1. Since Strategy 3 recomputes the path twice with each time step, once with the current maze and once with the predicted future maze, it not only aims to keep the agent away from the current fire locations, it also tries to keep the agent away from future fire locations in a precautionary manner. Strategy 3 has a peak at approximately $q = 0.6$.



Plot: Strategy 1 at dimension 50



Plot: Strategy 2 at dimension 50



Plot: Strategy 3 at dimension 50

- If we had unlimited computational resources, we would improve Strategy 3 by increasing how many steps ahead we calculate predictions of the fire. This would likely increase our success rate to make it closer to perfect. Our Strategy 3 only looks one step ahead, whereas if we had unlimited computational resources, we would aim to calculate more predictions. We could do this similarly to how we currently do it, but repeat the process for enough additional time steps to strengthen our prediction. This would give a more accurate guess of where the fire may spread.

- A potential Strategy 4 that takes ≤ 10 seconds per move would calculate as many fire predictions as possible, as mentioned above, but stay within the time constraint. This value will change every time, but would be time efficient and lead to a decent success rate given the time constraint. This is a more realistic approach than the one above, and still would do better than our current Strategy 3 would.

Acknowledgements

Nabhanya Neb (nn291):
We did both the code and the report together using Zoom. Because of this, we did not have to split anything up and we both contributed equally while working on call. We alternated who was coding/typing to make it fair.

Arpita Ray (ar1516):
The work was evenly divided because we did all of the code on screen-share. We took turns on who would be the person typing each time. For the report, we used Overleaf and did the write up the same way. We worked together on it and alternated who wrote.

Honor Statement:
We have read and abided by the rules of the project, we have not used anyone else's work for the project, and we did all of the work by ourselves.