Intro to AI: Project 1 - Maze on Fire

Ashwin Haridas (ah1058), Ritin Nair (rrn32)

February 2021

1 Introduction

In this assignment, we are tasked with implementing the following three path-finding algorithms:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. A* Search (with a Euclidean Distance metric as the heuristic)

In the first part, we implement and run these algorithms in a maze to compare and analyze them. In the second part, we move away from static mazes and try to design and implement strategies (that make use of these algorithms) for an agent to get through a maze that is "on fire".

Note: running our code will produce visuals (created using Matplotlib) for the three path-finding algorithms and three fire strategies. To run our code, use the "main.py" file. The comments in the file have specific instructions as to what the command-line arguments should be.

2 Part 1 – Preliminaries

2.1 Problem 1

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

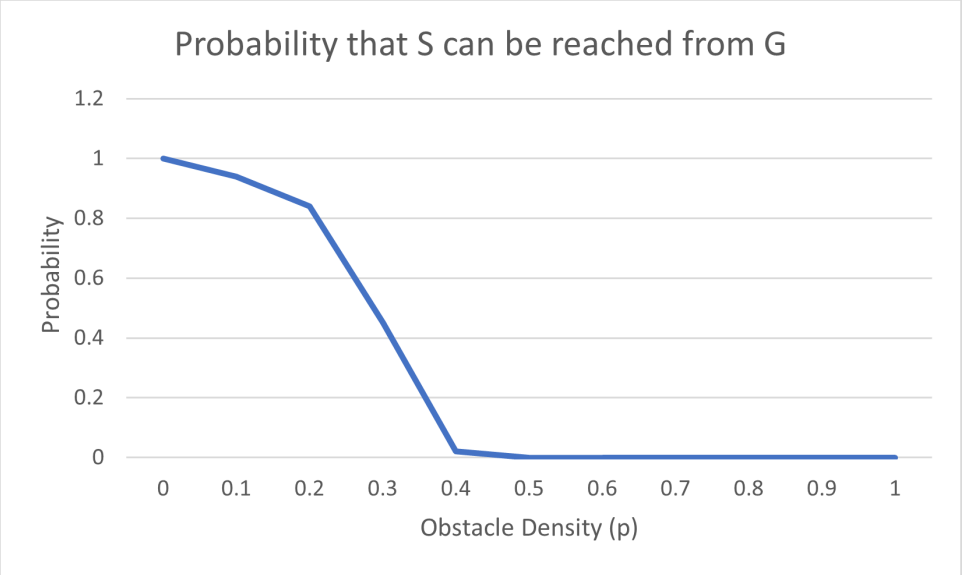
We use a two-dimensional array of integers to represent the cells of the maze. The elements of the maze are all initially set to 0. The start of the maze is denoted by a cell with the value 3 and the end of the maze is denoted by a cell with the value 4. To assign the walls of the maze each cell of the array is iterated through. At each cell if some random number between 0 and 1 is less than the desired obstacle density, p, of the maze, then that specific cell is assigned the value 1 to represent that it is a wall. Using this method results in the cells randomly being assigned as walls at approximately the same rate as the value of the obstacle density.

2.2 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'

In our DFS algorithm we use a stack structure as our fringe to store cells that need to be explored. The algorithm runs until the fringe is empty, at which point the search has failed and there is no valid path, or until the goal cell is popped from the fringe at which point a valid path has been found. By storing the parent cells of each cell that is explored we can also trace back and find the entire path. The DFS algorithm uses a stack as the fringe, so it will explore the last item added to the stack, or the most recent cell explored. It adds this cell's neighbors to the stack and then starts immediately exploring these neighbors too. This results in the traversal going deeper into the maze and there being less total elements in the fringe to be explored at a single time. DFS is a better choice than BFS for this problem because it will find a valid path from the start to the goal faster than BFS will if such a path exists. This is because the traversal will keep going deeper through the maze exploring the neighbor of a cell, the neighbor of this neighbor, and so forth. This is unlike the way BFS operates as the BFS algorithm will instead go through all the neighbors of a single cell before moving on to cells deeper into the maze due to the way a queue is used as the fringe. Therefore, DFS will have less elements in the fringe on average and will return the goal from the fringe quicker than BFS will.

The following trials were run on randomly generated mazes with dimensions of 100 and obstacle densities from 0 to 1 in 0.1 intervals. 100 trials were done for each obstacle density, and the probability that a successful path was found from S to G was recorded for each obstacle density. The results are shown in the following graph:



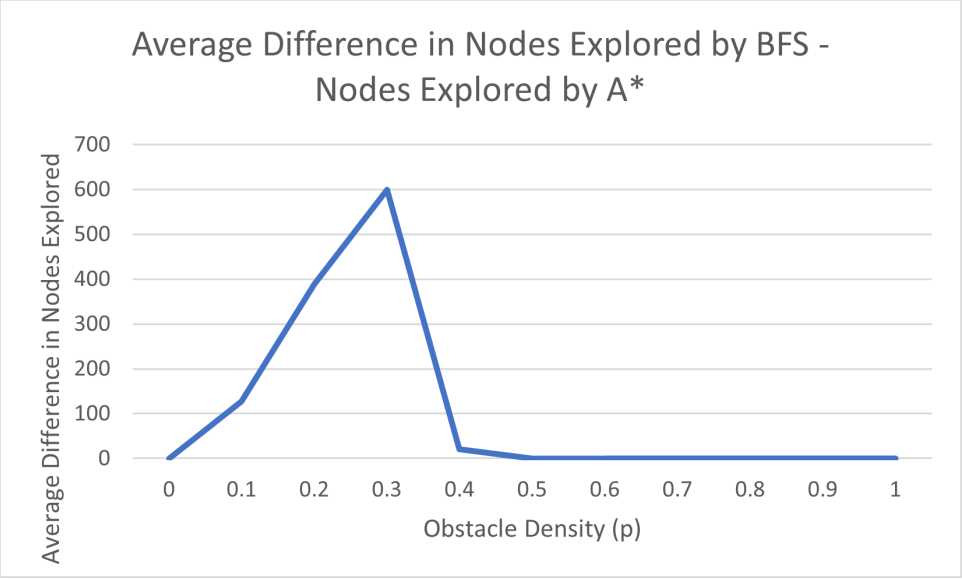
As seen in the graph, it appears that the probability that S can be reached from G decreases

as obstacle density increases. Something interesting that can be deduced from the graph is that when $p \geq 0.5$, the maze is pretty much impossible to solve!

2.3 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 would this difference be?*

In our BFS algorithm we use a queue as the fringe. The algorithm is run until the fringe is empty, in which case no valid path from S to G exists, or until the goal is dequeued from the fringe, in which case a valid path does exist from S to G. By tracking the previous element of each explored cell, we can trace back from the goal to find all the cells on this valid path. The valid path is also guaranteed to be the optimal path by BFS because this algorithm explores all the neighbors of a cell consecutively due to the queue structure of the fringe, thus meaning that the first path that finds the goal is the shortest one. In our A* algorithm we use euclidean distance as a heuristic to estimate the approximate distance to the goal from the current cell. The fringe in this algorithm is a priority queue, represented by a minimum heap, that prioritizes by distance traveled so far + euclidean distance to the goal. This means that the cell with the smallest amount for this value will be removed from the fringe and explored first. Using this method allows certain paths to be pruned out due to them not being able to be shorter than any already existing paths. The A* algorithm only prunes out nodes to be explored after a valid path has already been found. Therefore, if no such valid path exists, then A* will explore the same number of nodes as BFS explores. Thus, the average difference between the nodes these two algorithms explore can be expected to be zero in this case.



The above trials were run on randomly generated mazes with a dimension of 100 and obstacle

densities from 0 to 1 in 0.1 intervals. At each obstacle density, 100 trials were run recording the number of nodes explored by BFS - number of nodes explored by A* and averaged out.

2.4 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?*

The largest maze solved with DFS had a dimension of 14,000. The largest maze solved with BFS had a dimension of 3,500. The largest maze solved with A* had a dimension of 3,250. We initially expected A* to solve a higher dimension than BFS, since it will not go to any unnecessary nodes because of pruning due to the heuristic chosen. However, after analyzing the time complexity of our implementation of A*, we realized that there is a lot more overhead needed in order to perform the search algorithm compared to BFS. In our implementation of A*, we use a Min-Heap as our priority queue. Insertion and deletion from a Min-Heap takes $O(logn)$. Our implementation of BFS uses a regular Queue, which takes $O(1)$ for insertion and deletion. As we increase the dimension size of the maze to solve, the time adds up, so overall it will take longer for A* compared to BFS. This explains why our results are against our initial assumptions.

3 Part 2 – The Fire

3.1 Problem 5

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

Our strategy 3 is called "A.P.E", which stands for **A*** with **P**robability and **E**uclidean (clever, right?). In A.P.E., we use a modified version of A* which uses a heuristic that is based on the probability of cells in the maze being on fire some number of time steps into the future. First of all, this algorithm uses a function that determines the probability that each cell will be on fire some n amount of steps into the future. This works by first creating a probability maze from the fire maze where every cell on fire is assigned a probability of 1, because every cell initially on fire will stay on fire, and every other cell is assigned the probability 0. Then the following formula is applied to find the probability that each cell in the maze will be on fire in the next time step:

$$p = 1 - (1 - q)^k$$

In this formula p refers to the probability that the cell will be on fire, q is the flammability of the fire in the maze, and k is the expected number of neighbors of the cell that are on fire. The expected value of the number of neighbors that will be on fire is calculated by adding the probability of all the neighbors of the cell being on fire in the previous time step. Repeating this process n number of times results in producing a probability maze that stores the chance of each cell being on fire at n time steps into the future. We use this probability maze to build a heuristic that is used in our modified A* search algorithm to also account for dodging regions of the maze that have a high likelihood of being on fire in the future. Additionally, we choose how many time

steps to predict ahead in the future for based on how close the runner is to the goal in terms of euclidean distance. The formula for this value is given by the following equation:

$$n = \left\lceil \frac{d}{5} \right\rceil$$

In this equation, n is the number of steps to predict ahead for, and d is the euclidean distance from the current cell to the goal. After experimenting with other values for the constant to divide the distance by, we chose the value 5 because we felt that it would allow us to predict the fire maze's state further into the future when at the start of the maze while at the same time not over-predicting when the runner is close to the goal. The reason we want the number of steps to predict ahead for to scale with distance to the goal is because at the start of the traversal process it makes more sense to avoid large regions of the maze that will likely end up on fire and end up trapping the runner by surrounding them with fire and blocking off a path to the goal. In order to do this effectively it means that we also need to account for what the fire will look like more steps into the future. Conversely, predicting too many steps into the future could result in the runner not prioritizing taking a path to the goal when they are close to it because the prediction assumes that the area around the goal will be on fire in the future. However, in this case it would realistically make more sense to rush to the goal using the shortest path and only narrowly dodge cells that have a high chance of being on fire in the next step. This means that the algorithm only uses the probability maze prediction to avoid cells that would catch on fire in the time step immediately after the one where the runner reaches the cell if the runner is already close to the goal. This logic of prioritizing the shortest path instead of picking the seemingly safest one based on the fire maze prediction when the runner is close to the goal should in theory help increase survivability by allowing the runner to reach the goal quicker before the fire inevitably blocks the goal in the future.

The modified A* search algorithm uses a fringe that stores cells to be explored in the future to find a path to the goal. This fringe is represented as a min-heap of cells and an associated priority value. The priority value is determined by some combination of the distance traveled so far along the path to reach the current cell and a heuristic that combines some factor of the distance traveled so far and the probability that the cell will be on fire at n time steps in the future. This heuristic is given a higher weight if the probability of the cell being on fire in the future is relatively high. This is because this results in it being popped from the fringe and being explored later on in the algorithm, and potentially having any path that includes it completely pruned if a better path is found first. The algorithm does this to essentially dodge around areas of the maze that have a high chance of being on fire when constructing a path to the goal. The heuristic is given a lower weight if the chance of the cell being on fire in the future is relatively low because this indicates that the cell belongs to a safe path that will not pose that much of a risk for the runner to take. It results in the cell's priority value being dominated by the distance traveled so far which means that the path that contains the cell will be relatively short. This is favorable because if an area of the maze is not likely to be on fire in the future then it makes the most sense to find the shortest path through that area to the goal before the fire ends up spreading too much and blocking off potential paths. The priority value for the fringe used in this modified a-star algorithm is given by the formula:

$$\text{priority} = d + (d/c) * p$$

In this equation d is the distance traveled so far on the path to reach the cell, c is a constant to scale the distance to in order to give the probability of the cell being on fire a lower or higher weight depending on if the probability value is low or high respectively, and p is the probability of the cell being on fire at some n time steps into the future. More specifically c is 2 if p is between 0 and 0.3, 1 if p is between 0.3 and 0.6, and 0.5 is p is between 0.6 and 1. We picked these values after experimenting to see what would give the heuristic the most optimal weight to prioritize finding a shorter path through areas that are not likely to be on fire and a safer path around areas that are likely to be on fire.

This algorithm runs the modified A* search starting at the start of the maze to find an initial path to the goal while also avoiding the general area of the maze that will be on fire in the future. The runner then takes one step along the path returned by the modified a-star search and the fire is advanced one step. The process of calculating a new probability maze and recomputing a path with A* to see what cell to go to next is repeated at each time step. Every time the runner moves to a new cell the fire is also advanced one step. This process is repeated until the runner dies to the fire, is trapped in the maze because the fire blocks off a path to the goal, or escapes the maze successfully. The reason the algorithm recomputes the probability maze and a modified A* at each step is to use a more accurate and updated state of the fire maze in the calculations to find an optimal path to the goal.

We account for the future of the fire maze in strategy 3 by creating a matrix filled with the probabilities of its corresponding cells in the maze being on fire at some n time steps in the future. These values are used to help build a heuristic for our modified a-star algorithm to create paths to the goal that avoid areas of the maze that will be on fire. The amount of steps into the future we account for depends on how far the runner is to the goal, as we don't want to be overly cautious when the runner is close the goal already, but we still want to initially avoid areas of the maze that will be on fire later in the future.

Overall, by accounting for the future in our strategy 3 algorithm we are able to theoretically increase the chance that the runner will successfully escape the maze when compared to strategy 1 and strategy 2.

3.2 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?

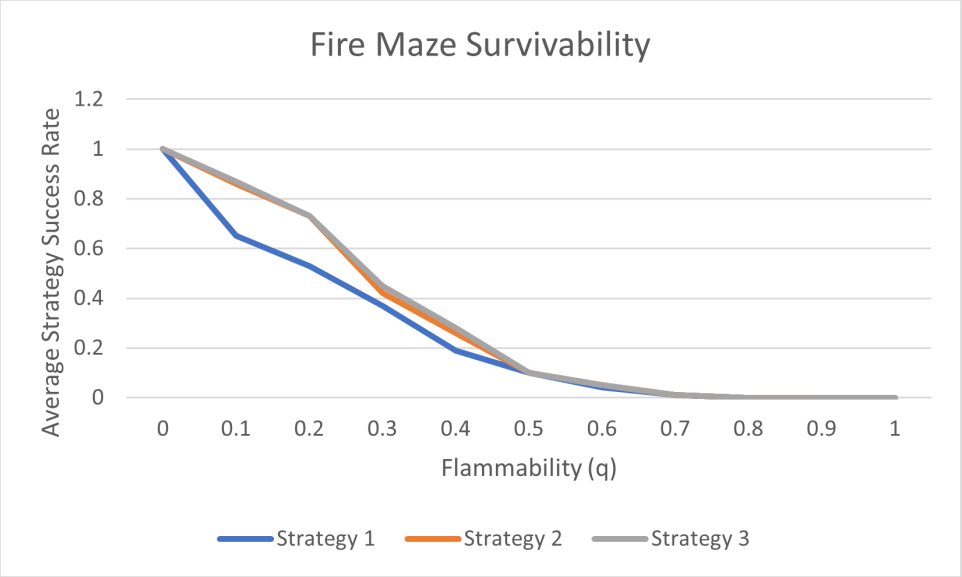
As seen below, we plotted each strategy on the same graph as it is easier to visualize how the strategies perform at different flammability relative to each other.

We notice that at $p = 0$ and at $p \geq 0.5$, all the strategies perform the same. This makes sense, because at $p = 0$, the fire will not move at all so the agent is pretty much guaranteed to make it to the goal (the only time where the agent would fail is if the initial fire blocks the goal, but we will not consider that scenario). At $p \geq 0.5$, the fire spreads too rapidly for any of the strategies

to succeed.

From $p = 0.1$ to $p = 0.2$, both Strategy 2 and Strategy 3 perform around the same, and are much better than Strategy 1. This is because in Strategy 2 and Strategy 3 the search algorithms to find the goal account for the current state of the fire in the maze to reduce the chance of the runner moving to a cell that is already on fire. In Strategy 1, however, this is not the case as the path to the goal is only calculated once and could result into the runner moving to a cell that is already on fire.

From $p = 0.3$ to $p = 0.5$ Strategy 3 slightly outperforms Strategy 2. This is because it accounts for the future state of the fire in the maze when creating paths that the runner should take to the goal, whereas Strategy 2 only considers the current state of the fire maze. This difference is very slight though because at these flammability the fire already starts to spread through the maze relatively fast.



3.3 Problem 7

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

To improve on Strategy 3 and take advantage of the infinite computational resources at our disposal we would need to take a different approach to the problem than we initially did for our original strategy 3. The way this new algorithm would work is by finding every possible path to the goal at each step and storing these traversal paths in a list.

To calculate every possible path to the goal we would use an algorithm similar to DFS, but instead of simply returning the first traversal path that reaches the goal we would instead add this path to some result list and keep the DFS procedure running until the fringe is empty.

The first path in the list would be used in a simulation to see if the runner moving along this path would have a high probability of them being able to successfully escape the maze. The simulation is run in a copy of the original fire maze, and any steps the runner takes in the simulation result in the fire advancing one step in this copy of the maze. If the runner dies in the simulation for this path, then the point they died at is stored and the next path is tested in the list of all the possible paths to the goal. This is repeated until one of the paths the simulation is run on successfully reaches the goal or until every path in the list has been tested. If a path reaches the goal in the simulation, then the runner advances one step along this path in the original fire maze and the fire is advanced one step in the original fire maze as well. If no path reaches the goal, then the one where the runner died closest to the goal based on euclidean distance is chosen and the runner moves one step along this path instead. Intuitively, we can assume that this path was the most successful one as it got them the closest to the goal and has the highest probability of letting them escape the maze in the traversal procedure for the original maze. After one step is taken along the chosen path and the fire is advanced one step, every possible path to the goal from the current cell is recalculated again and this process is repeated. It is important to note that the runner dying, being blocked off, or reaching the goal in a simulation does not mean that any of these things actually happen, but rather just act as an indicator for if the current traversal path being checked is good or not.

This approach requires a large amount of computational resources because at every step of the algorithm we are storing every possible path to the goal from the current cell instead of just one path, meaning significantly more copies of the maze are required to be made and stored in memory. Furthermore, the actual process of finding all possible paths to the goal is longer too as DFS no longer returns the first path that is found, but rather explores the entire fringe to find every possible path. This algorithm is an improvement to strategy 3 because it more accurately predicts the future state of the fire maze. This is because in our original strategy 3 we base our predictions and path to the goal on the probability that the next cell on the maze will be on fire at some time in the future. In this newer version of the algorithm, however, we instead run a simulation of what the fire could actually look like and if the runner would be able to successfully reach the goal along a certain path. This method of actually simulating the traversal process instead of basing the path on the probability of cells being on fire in the future allows us to better account for the evolution of the fire in the maze and potentially have a higher chance of survival.

The following pseudo-code describes the general algorithm and approach for this strategy:

Algorithm 1: `New Strategy 3`

```
Input: maze, fire_maze, start, goal, q
Output: status, fire_maze
current = (0, 0)
pathList = find_all_possible_paths(fire_maze, current)
//note: the cell in the fire_maze that equals 4 is the goal
while fire_maze[current] is not at the goal:
    for each maze in the pathList:
        status, fire_maze, final_cell = simulate(current, goal,
                                                 fire_maze)
        if status == 'Success':
            break
        else:
            add the maze and the final_cell to a failedList
    if no path succeeded:
        go through all the mazes in failedList and pick the one with
        the lowest euclidean distance to the goal to set as the current
        path
    current = next cell on the current path
    advance_fire_one_step()
    if dfs(current, goal, fire_maze) == 'No path possible':
        return 'No path possible', fire_maze
    if fire_maze[current] == 7 // (runner catches on fire)
        return 'Died', fire_maze
    pathList = find_all_possible_paths(fire_maze, current)
return 'Escaped', fire_maze
```

3.4 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.

Recall that in Strategy 3, our main improvement from the previous strategies was to compute a probability maze n steps ahead (n determined by the current distance from the agent to the goal) and then run A* using a heuristic based off this probability maze so that we could avoid where the fire was going to spread in the future. Although this strategy is an improvement, it falls in the area of amount of computation needed: computing the probability maze n steps ahead and running A* at each time step takes quite some time. We took this into consideration when we were developing Strategy 4, which is meant to do less computation as we only have a limited amount of time between each move.

In this strategy, we will still make use of the probability maze, but instead of calculating the probability that a cell will be on fire n steps ahead, we only calculate the probability that the cell is on fire *one* step ahead. We do this because it is true that computing the probability maze

for more than one step ahead takes a good amount of time, so by only computing for one step ahead, we still account for the future state of the fire. However, this takes less computation and therefore less time. Another change that we could make is instead of using a global search algorithm like A*, we use a local search algorithm. What we can do is use the probability maze that we computed earlier and look at the the probability that the direct neighbors of the cell that we are currently on will be on fire (we are looking one step into the future). Then, we choose the neighbor with the lowest probability of being on fire in the next step and move there. This will definitely reduce computation (and also space complexity) as we will always only be looking at the direct neighbors and making a move based on that.

In summary, our strategy works like this: we start at our initial cell and compute the probability maze for the next step. We then look at our neighbors and choose the neighbor with the lowest probability to move to on the next time step. We repeat this until we either touch the fire or reach the goal. Overall, this will definitely reduce the computation overall needed, but what are the drawbacks or trade-offs? First, since we are only calculating the probability maze for one step into the future instead of n steps into the future, we may actually have a lower success rate. This is because we may not be predicting far enough into the future, and so the path we choose to go down may ultimately lead to touching the fire. Also, since we greedily pick the cell that has the lowest chance of being on fire in the next time step instead of using a global search algorithm, there is no guarantee that it won't get stuck at a dead end or waste too many steps in order to reach the end (wasting too many steps may also cause the fire to block the goal, which also means the agent fails).

Overall, our Strategy 4 will make use of less computation compared to the other strategies. Although there are some trade-offs we have to make, they are needed in order to limit the computation needed to get through the maze and avoid the fire.

The following pseudo-code is a general idea of how our strategy would work:

Algorithm 2: `Strategy 4`

```
Input: maze, fire_maze, start, goal, q
Output: status, fire_maze
current = (0, 0)
while fire_maze[current] is not at the goal:
  if current is at fire:
      return 'Died', fire_maze
  probability_maze = generate_prob_maze(fire_maze, q)
  current = find_neighbor_with_lowest_prob(fire_maze, probability_maze)
  fire_maze = advance_fire_one_step(fire_maze, q)
return 'Escaped', fire_maze
```

4 Contributions

The work for this assignment was split evenly. Specifically, Ashwin did the implementation for generating the maze, displaying the maze visually after plotting it, DFS, A*, part of the modified

A* algorithm, initializing and advancing the fire in the maze, and strategy 1. Ritin did the implementation for BFS, part of the modified A* algorithm, initializing and calculating a probability maze, strategy 2, and strategy 3. As for the report, Ritin typed up the answers for questions 1, 2, 3, 5, and 7. Ashwin typed up the answers for questions 4, 6, and 8 and performed the associated trials and data collection necessary for answering these questions. We tried to divide up the workload for this assignment evenly between each other. We also thought of most of the general algorithms and approaches we decided to take for the problems in this assignment together.

5 Honor Code

This assignment was done on our own. None of the code or the report was copied or taken from online sources or any other student's work.