Anthony Triolo
Section 3
NetID: ajt182

Biyun Wu
Section 5
NetID: bw366

Problem 1

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

In order to generate the matrix, we have two conditions of what can be put in each spot: it should be a 1 (blocked) with probability p and a 0 (available) otherwise. We do this at the specified size n. Then, we make sure that the starting and ending cells are both marked as available by setting their values to 0. We then return a numpy array since it is easier to work with. The code segment for this algorithm is as follows:

```
import numpy as np
import random

def generate_maze(n, p):
    """generate the matrix for the maze game
    :return: a numpy 2d array with 0s(available cells) and 1s(blocked cells)
    """
    mat = [[1 if random.uniform(0, 1) <= self._p
        else 0 for _ in range(self._dim)] for _ in range(self._dim)]
    mat[0][0] = 0
    mat[self._dim - 1][self._dim - 1] = 0
    return np.array(mat)
```

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'.

2.1 The Algorithm

When writing the DFS and BFS algorithms, we decided that it would be easiest to write a generalization of an algorithm that checks if a target point is reachable from a starting point. For this, the algorithm takes in a boolean value to tell the algorithm if this is being used for DFS or not. We then use this boolean value to tell the algorithm how to use the fringe; as a stack for DFS and as a queue for BFS. We then call this algorithm when we call the DFS function, with the boolean value being $True$.

For DFS, this algorithm starts by putting the starting position, represented as a Coord class which associates the maze to the x and y coordinates of a point, on the stack. This Coord is then popped off the stack, and the first check is done to check whether or not this Coord has the x and y of the destination cell. If it is, then we're done, and return $True$ to signify that a path was found. If not, we check for the neighbors of this position, and add them to the stack. Once we pop a Coord off the stack, we mark it as visited to make sure we don't revisit the same point in the maze. The Coord structure also contains a reference to the coordinates of its parent cell, or in other words, the cell that came before it. This allows us to do backtracking at the end of the algorithm to find the path that was found. We continue to pull nodes off of the stack until we find the destination. If the

fringe ends up being empty and we haven't found the goal, $False$ is returned because this means that we can't reach the goal and we have searched all reachable cells in the maze. The code segment for the algorithm is as follows:

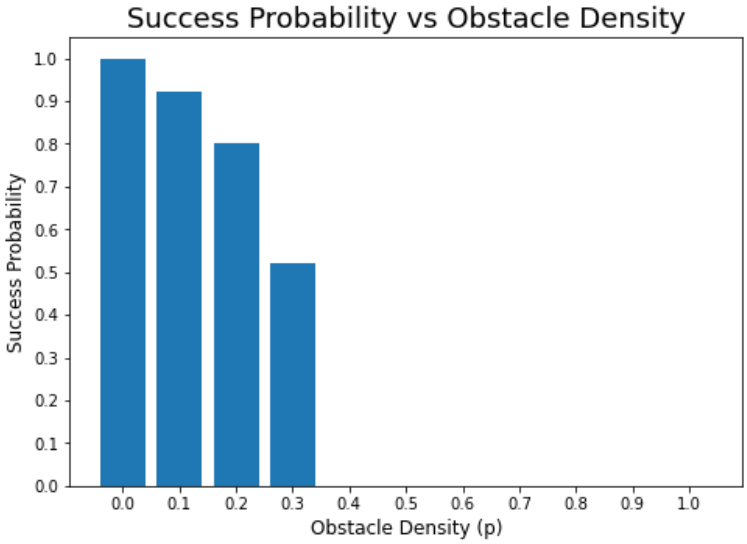
```
def __is_reachable(self, start: Tuple[int, int], goal: Tuple[int, int], is_dfs: bool):
        """check whether goal(goal) is reachable from s(start) in the maze
        :param start: int tuple contains starting coordinates
        :param goal: int tuple contains goal coordinates
        :param is_dfs: True/False for dfs/bfs
        :return whether start and goal are reachable from each other
        """
        self._nodes_explored = 0
        self._matrix = self._m.copy()
        m = self._matrix
        if m[start[0]][start[1]] == 1 or m[goal[0]][goal[1]] == 1:
            return False  # not reachable because the start/target c is occupied.
        fringe = [Coord(m, start[0], start[1])]  # stack/queue
        while len(fringe) != 0:
            cell = fringe.pop(-1) if is_dfs else fringe.pop(0)  # dfs/bfs
            if not cell.is_available():
                continue
            cell.close()  # add restriction
            self._nodes_explored += 1 # count cell as explored
            if cell.get_coords() == goal:
                self._path = cell.get_path()
                return True
            else:  # add valid neighbor cells to fringe
                neighbors = cell.get_neighbors()
                for ne in neighbors:
                    ne.set_prev_to(cell)
                    fringe.append(ne)
        return False
```

2.2 Use over BFS

For finding whether or not there exists a path from the start to the finish, DFS will in most cases be more useful and faster than BFS. This is due to the fact that we are not trying to find an optimal path, we are just trying to find the presence of a path. So, rather than checking every cell in the maze, we can just go down one branch of the decision tree. This is beneficial in the case that this tree is very deep, in which BFS would take a long time to find the goal node, while DFS could get to it relatively quickly as long as there aren't too many twists and turns throughout the maze.

2.3 Performance

In order to test the performance of DFS at each obstacle density value p, we need to check the probability of successfully finding a path at these values of p. To do so, we decided to run DFS on 25 different mazes of size 500 for each p. We then find the number of successful runs out of these 25, and calculate the probability using this value. This is done for each p from 0 to 1. This produces a graph of probability vs obstacle density.



From the graph, we can see that from $p = 0.0$ to $p = 0.3$, the success rate decreases relatively linearly, where this rate is always over 0.6. We then see a massive falloff from $p = 0.4$ onward, where the algorithm can never find a path. This can be attributed to the fact that because the density gets so high, either the starting point is blocked from the rest of the maze, or the ending is blocked off from the start.

Problem 3

Write BFS and $A^\star$ 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 $A^\star$' vs 'obstacle density p'. If there is no path from S to G, what should this difference be?

3.1 The Algorithms

As mentioned earlier, the BFS algorithm uses the same *is_reachable* function discussed in the DFS section of this report. The only thing that changes is that a queue is used instead of a stack. In this way, every node in the maze is explored and an optimal path is determined based on the least number of nodes to get from start to finish. For $A^\star$, things get a little more complicated. We begin by using a priority heap. This heap prioritizes nodes that have the lowest estimated cost to get to the goal when running through it. This estimated cost function $f(n)$ is made up of the sum of two different functions in order to estimate the cost of going through a specified node; $g(n)$ and $h(n)$. $g(n)$ denotes the cost that it has already taken to get to the specified node. In the case of a maze, this is the number of steps it had taken while searching to get to this node. $h(n)$ is the Euclidean distance from the specified node to the goal. By finding nodes with the lowest value of $f(n)$, we will ultimately find an optimal path from start to finish, while generally exploring fewer nodes than BFS. We use a heapq in python, which is a type of prority heap. The way it works is that when something is pushed onto the heap, it resorts the heap based on a certain specified function denoted by $\_\_lt\_\_(other)$, where "other" is every other node on the heap. In this case, the function is the $f(n)$ function mentioned earlier. The function for calculating Euclidean distance and the "lt" function code segments are as follows. These functions are stored in a "EuclideanCoord" class, which is a subclass of the normal Coord class used in BFS and DFS:

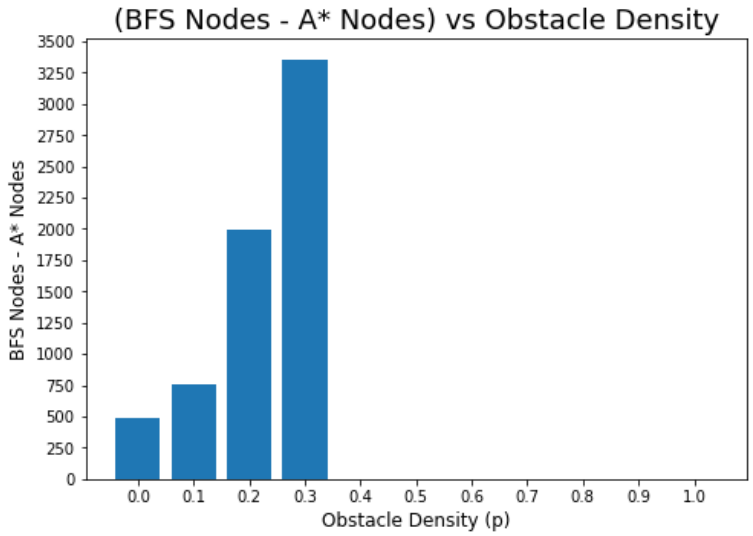
```
def __euclidean_distance(self):
        """heuristic
        :return: euclidean distance from the cell to the goal
        """
```

```
        return math.sqrt((len(self._m) - self._x - 1) ** 2
                         + (len(self._m) - self._y - 1) ** 2)

def __lt__(self, other: 'EuclideanCoord'):
    """less than function, for peer comparison
    :param other: an EuclideanCoord object
    :return: whether this cell has better heuristic value over the other cell
    """
    return self._steps + self.__euclidean_distance()
            < other._steps + other.__euclidean_distance()
```

3.2 Performance

In order to evaluate the performance of these two algorithms at different p values, we can take a look at how many nodes were explored, or visited, by each algorithm. We know that BFS will always explore all possible nodes in the maze, while A^* may not. To produce the necessary graph, each algorithm was run on the same 25 mazes at each value of p, and the difference in the number of nodes were recorded for each of these mazes. The mazes were of size 250, as this was the largest size that resulted in a reasonable runtime. The average difference at each p were calculated, and then plotted against p.



We can see that consistent with the graph of success for DFS, starting at $p = 0.4$ the difference in nodes is 0. This means that no path was found, since both BFS and $A^\star$ had to explore every possible node, before finding that there is no way to reach the goal. For $p = 0$ to $p = 0.3$, we see that the difference rises almost exponentially. This means that $A^\star$ is finding a shortest path without exploring as many nodes as BFS, meaning it is more efficient as expected. The reason it starts so low is because with no blocking nodes, which means that there is nothing preventing $A^\star$ from exploring outwards. Because we are required to use Euclidean distance for the heuristic, the number of nodes $A^\star$ explores at these lower p values is most likely higher than it could be. This is due to the fact that the number of steps, $g(n)$, ends up dominating $h(n)$, causing the nodes explored to be more dependent on steps taken to get to a certain node, rather than distance to the end. Something like Manhattan distance would be a better heuristic in this case because we are only able to move in cardinal directions as opposed to diagonally. When we begin to increase p, the blocking nodes can filter out nodes that may have had a shorter $g(n)$ before, but are not considered since they are blocked. This potentially prevents $A^\star$ from exploring parts of the maze that are further away from from the goal, while BFS continues to explore these nodes.

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 $A^\star$ at $p = 0.3$ in less than a minute? For each search algorithm, if we only ran a certain size maze once,

we may get inaccurate results, due to the fact that the algorithms may get mazes that are favorable to them, and thus artificially run in less time than they usually do. To combat this, each algorithm is run on 5 different mazes, and the time is then averaged and evaluated. We also decided to only take into account successful runs, since failed runs have a fractional time to solve most of the time, which would skew the average. Also, we determined that ßolved"meant finding an actual path. For each algorithm, an initial size was determined to start at based on external tests, and the increase of size per iteration is determined in this way. Once the average time exceeds 1 minute, we know approximately what size we can reliably solve for each algorithm.

Algorithm	**Max Size**
BFS	2800×2800
DFS	7200×7200
$A^\star$	1250×1250

We see that DFS is able to have the largest size before the search takes a minute on average. This is due to the fact that DFS can end up searching down one branch of the search tree and find the goal, while BFS has to search every available node before finding the goal since the goal node is located on the lowest level of the search tree. DFS ends up generally visiting many fewer nodes, which means that it will end up finding a path faster since there is less to check. When running the tests, there were instances where DFS would find a path in about 2 seconds on mazes of size 7000+. This means that there were times where DFS found the path on its first run, exploring only the necessary nodes along the path, and most likely never had to backtrack to check a different node in the decision tree. We see the $A^\star$ was able to solve the smallest sized mazes in under a minute. This can be attributed to the fact that pushing an element to a priority heap generally takes $O(\log(n))$ time, while adding to or popping off a regular queue/stack is $O(1)$. So even though $A^\star$ may explore fewer nodes than BFS overall (as shown in earlier graphs) at $p = 0.3$. This does not make up for the $\log(n)$ time penalty for very large n. So as n increases, $A^\star$ slows down significantly. Also, as mentioned earlier, we may be exploring more nodes than necessary, due to the fact that we are using Euclidean distance instead of something more useful for this case like Manhattan distance. If we were able to explore a minimal number of nodes, we may be able to make up for the $O(\log(n))$ time for adding element to the priority heap.

Problem 5

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

When considering a good strategy, we had to look at what strategies 1 and 2 are good and bad at. Strategy 1 is good in the sense that it only does one total search, which is pretty fast. The problem is that this strategy does not take into account the fire actually spreading. It just takes an initial path and iterates along it. This means if the initial path is close to the fire, and the fire spreads, that it will likely run into or be absorbed by the fire on one of the steps. Strategy 2 does a decent job taking this flaw into consideration by recalculating the path on every step. By doing so, it looks where the fire now prevents it from going and avoids these spaces. There are two issues with this: it is slow to calculate a path every step, especially for large mazes, and the agent is potentially wasting time moving towards the fire initially, and then having to backtrack to move away. So we need a strategy that not only does a search only once, but also considers where the fire may go when calculating this path.

What we came up with was something called "Simulation Assisted Search" (SAS). What this entails is that for a given maze and initial fire position, multiple simulations are done without the agent in order to try and help predict how the fire might spread. This is done by checking, on average, how many steps did it take for the fire to spread to each point in the maze throughout all of the simulations. This information is stored in its own matrix, where the average number of steps is stored in the respective coordinates location within the matrix. We then use an algorithm similar to $A^\star$, in that we use a heuristic based on Euclidean distance, and a cost function based on the number of steps taken. However, we scale the total estimated cost by multiplying $h(n) + g(n)$ by the inverse of the average number of steps taken before the specified node was on fire in thew simulations. This allows us to take into account the unknown future of where the fire may be, while keeping the path close to optimal. Since we use a min-heap (priority queue), smaller values are better, which is why the inverse of the average steps taken until the point was on fire, is used. In this way, spaces that are generally further away from the fire are preferred, so no time is wasted moving toward the fire, which is the flaw of strategy 2. We also only do one path formation, like strategy 1, however this time we take into consideration where the fire might be at the next step based on the cost function defined earlier.

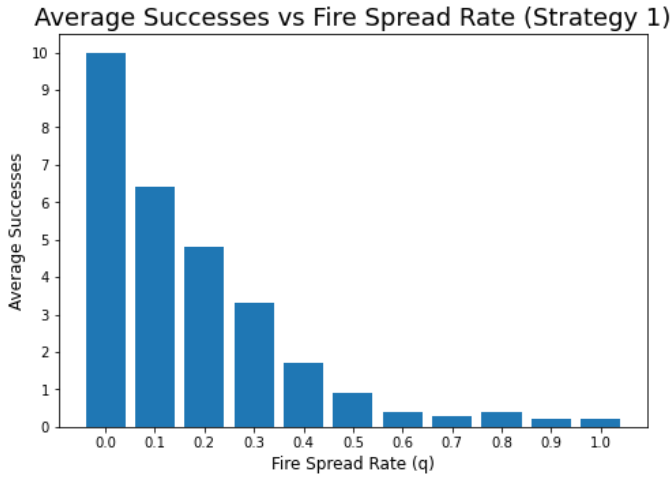
Problem 6

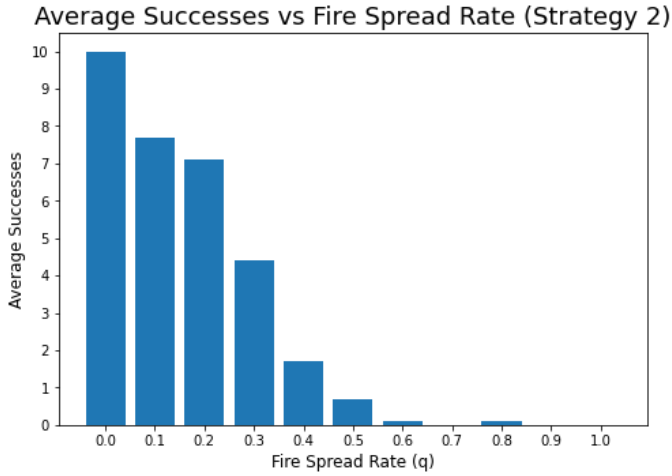
Plot, for Strategy 1, 2, and 3, a graph of 'average strategy success rate' vs 'flammability q' at $p = 0.3$. Where do the different strategies perform the same? Where do they perform differently? Why?

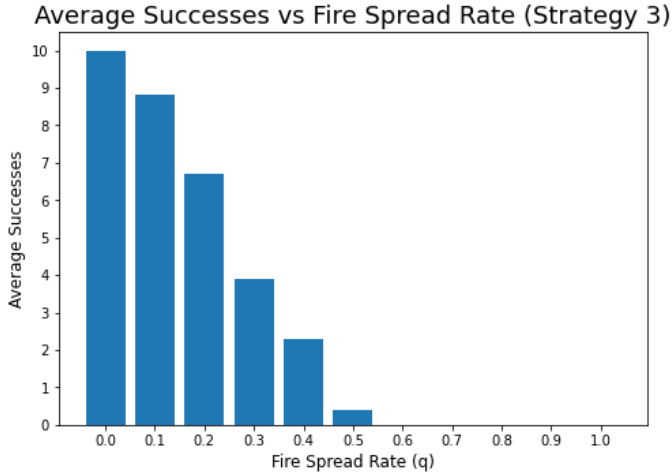
6.1 Methodology

In order to do testing on each strategy, for each value of q, 10 mazes were generated. Then in order to try to minimize the luck factor of fire starting positions, each maze was run with 10 different fire starting positions. For strategies 1 and 2, BFS was used since the assignment says that we should try to find the *shortest* path while surviving. Any maze the was unsolvable, or where the fire could not reach the starting location were discarded and not taking into account. Then, the average number of maze successes (out of 10) was calculated, and stored for each value of q. These averages were then plotted against their respective q values. We decided to use mazes of size 75, as this was the largest maze size without the strategies taking too long to run. For strategy 3, we decided to simulate each maze's fire spread 10 times.

6.2 Plots







6.3 Analysis

First, let's talk about run time. Strategy 1 was by far the fastest algorithm by far, due to the fact that it does a single path calculation, and then just iterates through it. Strategy 2 is significantly slower since it has to calculate a new path every iteration through the maze. And in this way, the more the fire spreads, the more it has to change the path. Strategy 3 takes almost the same amount of time as strategy 2 due to the fact that it needs to run the 10 simulations per maze, which can take some time to do. It is a little bit slower due to the reasons for slow down stated in the section for problem 4. However, when it comes to time between steps, strategy 3 is much better than strategy 2 due to the fact that all calculations are done prior to stepping through.

Next, lets discuss actual performance. If we take a look at the graphs, we can see that all strategies are always

successful at $q = 0.0$, due to the fact that the fire does not spread, so any initial path found is the correct path. As we move along, we see that strategy 1 decreases in success much more quickly than 2 or 3. This can be attributed to the fact that it has a high likelihood of running into the fire along the only path it calculated, especially if the fire starts near the initial path. Between strategies 2 and 3, strategy 3 is always more or as successful at these lower q values. We can see that this is especially the case at $q = 0.1$ and $q = 0.4$. Strategy 3 is successful in approximately 1 more maze than strategy 2 on average. This means that our tactic of simulating where the fire may go is effective in staying away from the fire at these lower q values. This is because no time is wasted potentially moving towards the fire, as in strategy 2, since we will generally move away from the fire. As we approach the higher values, we see that all strategies approach 0 successes. This means that the fire is spreading so quickly that no strategy can keep up. In fact, any difference in success between strategies can be attributed to the luck of the fire starting location, as well as where the blocking spaces are. If we were able to do more mazes and more fire starting locations, as in if we weren't limited by time and resources, we may see them all approach 0.

We tried many different tactics to try to improve our strategy 3 even more, but there appears to be a sort of upper limit to how good an algorithm can be in surviving, especially where $p = 0.3$. We tried using Manhattan distance instead of Euclidean, as well as manipulating the data we got from the simulations to try and change the estimated total cost function $f(n)$ to be more favorable. However, at $p = 0.3$ paths that could potentially yield better survival results at lower p values may be blocked off by blocking squares at $p = 0.3$. This limits how far we can stay away from locations where the fire may spread, and also causes the path to go in a more round about way that allows the fire to spread to more cells.

Problem 7

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

If there were no limits on computational resources, the strategy would look something like a stochastic game tree. What we mean by this is that for each step, there is a deep decision tree based on the possible paths an agent might take from that step to the end. Based on a spreading fire, some of these branches and paths may result in the agent catching on fire and not reaching the end. This means that for each step made from a certain space in the maze, there is a certain probability that we will succeed if we take a certain next step. For example, if we take a step, we can look at the maximum 4 possible direction the agent can go (up, down, left, right). For each of these, there is a certain probability that any path that results from taking this next step ends up succeeding. This is based on potential spreading patterns of the fire, and the other routes we take.

So, what we do is we start at the starting node with an initial fire position. For this node, we do a simulated exploration of the possible directions we can go for this starting node. For each of these directions, we run a specified number of trials, which if we had unlimited resources, could be an extremely large number. What these trials do is run a step-by-step simulation from the specified direction to the end node using a bfs while spreading the fire. Between each fringe node analysis during the bfs, the fire spreads. By doing this, we will get a certain number of successes and a certain number of failures for each direction from the initial node. We then pick the direction that produced the most successes, and choose this node as the next node in the final path. This node now becomes the next node we do simulations on, with the fire location being where the fire spread after taking the step to this new node. By the end, we will have a path made up of the most successful directions from each parent node. This seems like it would be extremely successful since it considers were the fire might spread after each step in the maze, and what the most successful direction to go are while the fire spreads from an initial location. This is even better than our strategy since it takes into account the fire spread between steps, as well as where the fire may go if we go in a certain direction, and what this direction should be in the final path.

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.

A potential strategy 4 could be one of two things. One could be something similar to our strategy 3 that we came up with. There is no computation done between steps, so it is extremely quick between them. What can take a while is the computation done during simulations before stepping though. So how can we get around this? Well, we can move back to doing computation between steps, but in a less intensive manner than the strategy mentioned in problem 7. What we could do is run the simulations that we did in strategy 3, but run very few. These simulation do not take that long, as they just spread fire, and store how many steps it took to reach each node. For each step, we can run this simulation, taking into account any fire that has spread in previous steps, and recalculate the probabilities of spreading to other nodes. We would then rerun the search algorithm, which is also not very time intensive on relatively small mazes, and calculate a revised path. This is similar to strategy 2, in that it recalculates a path every time, however it takes into consideration the possible future of the fire by simulating what the fire might do. This allows the agent to stay away from the fire, unlike the potential of moving towards it in strategy 2. This method, like any method that does computation between steps, would slow down as the maze size increases. There is really no way around this since we can have mazes of infinite size.

Contributions

For this project, Biyun (bw366) started by creating the class structure for this code, that is, creating things like Coord and EuclideanCoord for use in the search algorithms. He also programmed these search algorithms, and then continued to test and help analyze the algorithms throughout the project. Biyun was also able to come up with the module 'FireMaze.py', which allows us to do the fire simulation for our strategy 3. Biyun was critical in continuing to evaluate the methods that Anthony implemented to make sure that everything was going as expected. This allowed us to find errors and fix them.

Anthony (ajt182) was the one who programmed and implemented the three strategies. This included programming the actual SAS algorithm for strategy 3, which uses the data given by the simulation module. He also created the actual fire generation function, along with the function that allows us to begin running a maze with the fire in it. Anthony also programmed the necessary tests for both the normal search algorithms, and the performance of each of the three strategies. This includes creating the required plots for these tests. Anthony then took these plots and put them in the report, along with the necessary evaluation. Anthony continued with this report, and formally wrote up all of the answers to the problems.

Honor Pledge

We, Anthony and Biyun, certify that all work is our own, and that no work is copied or taken from online or any other student's work