This Maze is on Fire!

Sanjay Koduri and Jerry Yang

February 2021

1 Abstract

In this project, we worked on an AI that attempts to reach the goal in a maze. The maze is a square and the AI will be unable to travel through walls or "occupied cells". In addition, the maze may be set on fire. In this case, the fire will spread with each move of the AI.This project is done in Python and Pygames is used for the GUI. Numpy and matplotlib were used for the graphs. We will also have some pseudo code in the report, which is not going to be the code exactly as it is in the Python files for simplicity. Most of the following algorithms below can be found in the firemaze.py file. The startMaze.py is mainly for GUI purposes and will not encompass any of the algorithms below.

2 Problem 1: Maze Generation

To create a maze with dimension d and obstacle density p, we made the *generateMaze()* function which takes in the two variables. Below is a simple pseudocode. The algorithm starts off by

Algorithm 1: generateMaze(p,d)

```
1 maze = [[0]]*d for x in range(d)      // Generate 2D Array of 0s
2 totalCells = d * d //
3 cellDensityProb = p * 100      // Turn probability to number out of 100 for later rng
4 for cell in maze:       // the 2d Array, note a double for loop is used in actual code
5      cellDensityResult = randomInt(1,100)      // random number for cell density
6      if cellDensityResult less than cellDensityProb:
7           cell = 1      // Cell Occupied
8 maze[0][0], maze [d-1][d-1] = 0       Make Start and End Points open
9 return maze
```

generating a 2D array of 0s. 0s represent open or unoccupied cells, meaning that the AI is free to move onto these spots. Then based on the input of cell density p, the algorithm traverses through the maze, individually changing each cell to occupied accordingly. Below is the sample output for a 30x30 maze with an object density of 0.2. The first figure shows the output of the maze using the Pygames UI. The second shows what the maze actually looks like as a 2D array.

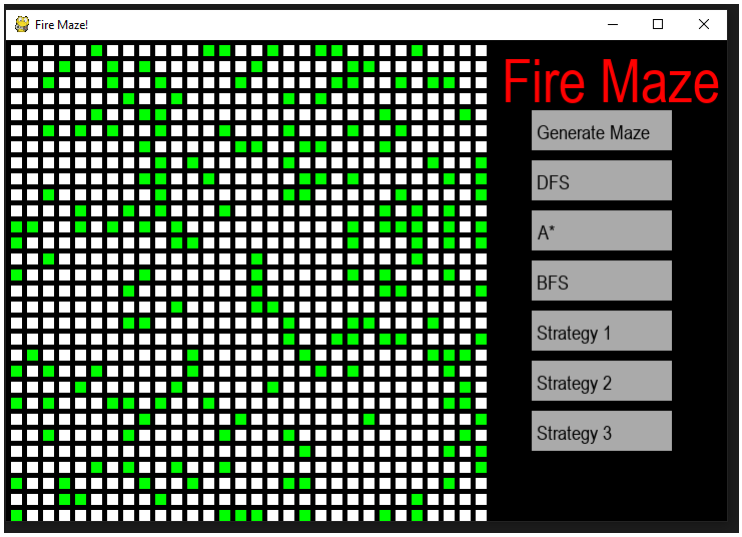

Figure 1: An output of a maze with the UI.

Figure 2: An output of the maze without the UI.

3 Problem 2: DFS

The DFS algorithm takes a maze and two locations and tells us whether or not one is reachable from the other. DFS is better than BFS in an algorithm like this because we do not care about optimality. The DFS algorithm saves space in terms of memory on the fringe, but gives up the guarantee that the solution found is optimal. However, since we just want to know whether or not one is reachable from the other, finding the optimal solution is unnecessary and so DFS is better. Below is a quick pseudocode of the DFS algorithm. The generateValidChildren function is used but not included in the below pseudocode. You can look at the source code but it is a function that takes in a point of a maze along with the path traveled so far. It takes the surrounding points of the children as a list, and then filters out the invalid routes (repeats, out of bounds, wall/occupied cell, fire, etc.) Below, a sample output is included as well

Algorithm 2: dfs(*maze*,*startPoint*,*endPoint*)

```
1 fringe = generateValidChildren(startPoint,maze)
2 while fringe != []:
3     current = fringe.pop()
4     if current == endpoint:      // endPoint has been reached
5         return True
6     else:
7         newChildren = generateValidChildren(current,maze)      // children of current
8         for cell in newChildren:      // Add the new children to the fringe
9             fringe.append(cell)
10 return False
```

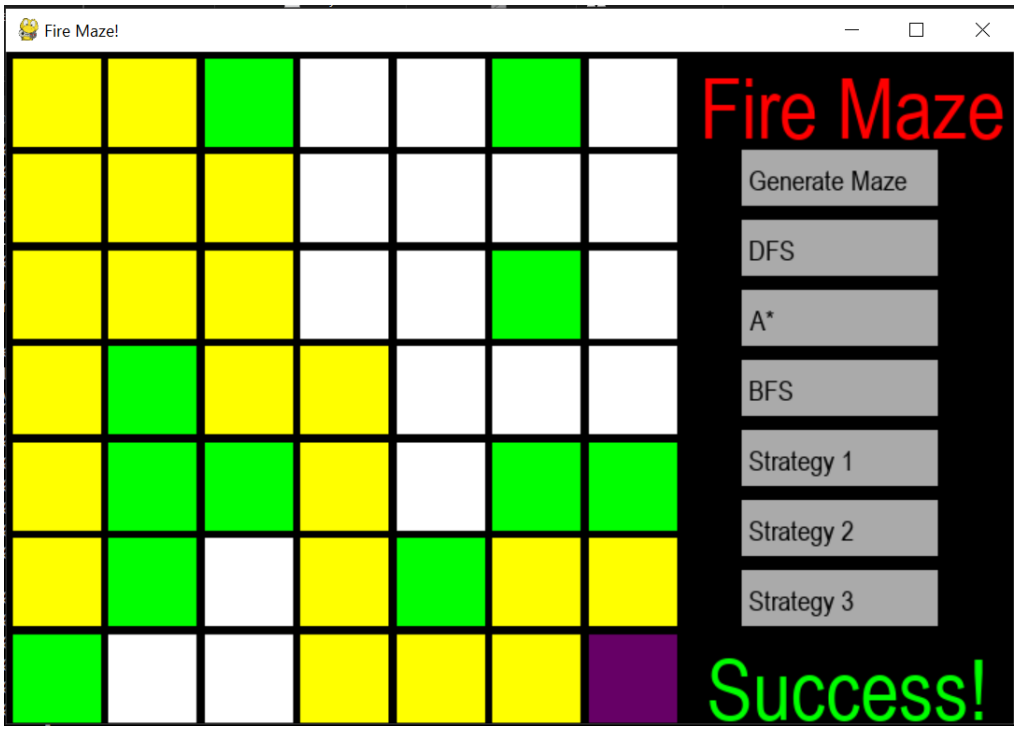

Figure 3: DFS Sample Output

How effective is this DFS and to what extent will the object density hinder the success of the AI? Below is a plot of object density p vs the success of reaching the endpoint from the start. The dimensions for the sample will be 50x50 mazes.The percentage of successes is based on a sample size of 500 per bullet point. It can be seen that at object density of 0.5 and higher, the chances of there being a path from the start to the goal is essentially nonexistent.

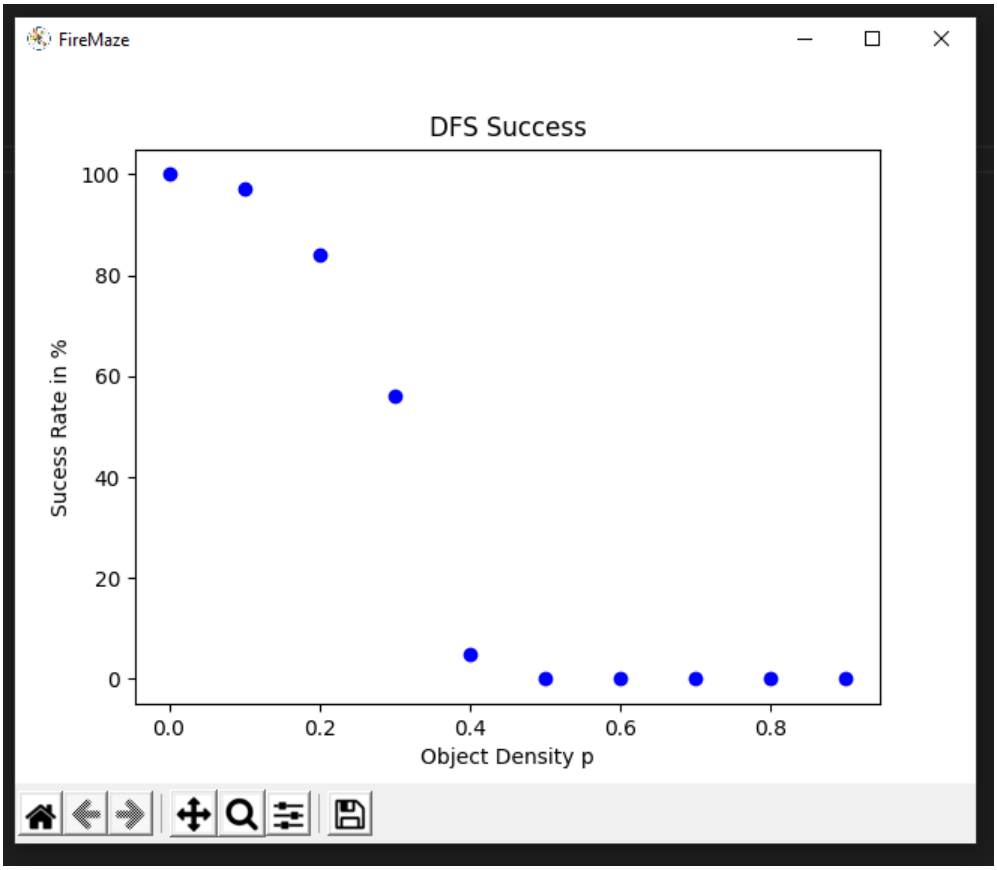

Figure 4: Obstacle Density vs Probability of Success

4 Problem 3: BFS and A*

The BFS and A* algorithms both takes in a maze and two points, and determines the shortest path from the start to the finish. The A* algorithms uses euclidean distance as a heuristic. While BFS promises optimality, it is quite costly in terms of the fringe because of the sheer number of nodes it has to explore. On the following pages are two brief pseudocodes of the two algorithms, along with a sample output from each. The generateValidChildren function is the same one used for DFS.

Once again the algorithms below can be found more specifically in the python files as the pseudocode should be viewed more as outlines. Both algorithms are quite similar, however, a* uses an additional heuristic when choosing from the fringe which path to take. This can be seen in the below a* algorithm through the lowF() function. Another thing to note is that both actually does produce the optimal path as a result.

Algorithm 3: bfs(*maze*,*startPoint*,*endPoint*)

```
1 open    // Create open fringe to hold points to be explored
2 past = [[Point(−99, −99)] ∗ (len(maze))for_inrange((len(maze)))]    // 2d array
  meant to show path
3 open.append(startPoint)    // Add startPoint to visited
4 while True:
5     current = open[0]    // Push and pop from open to close
7     open.pop[0]
8     if current.x == endPoint.x and current.y == endPoint.y:    // end reached
9         return past return the path
10    kids = generateValidChildren(current, maze, open)    // collect kids of current
11    open.extend(kids)
12    for prev in kids:
13        past[prev.x][prev.y] = Point(current.x, current.y)
14    if len(open) == 0:    // No more nodes, failed, return the path so far
15        return path
16 return path
```

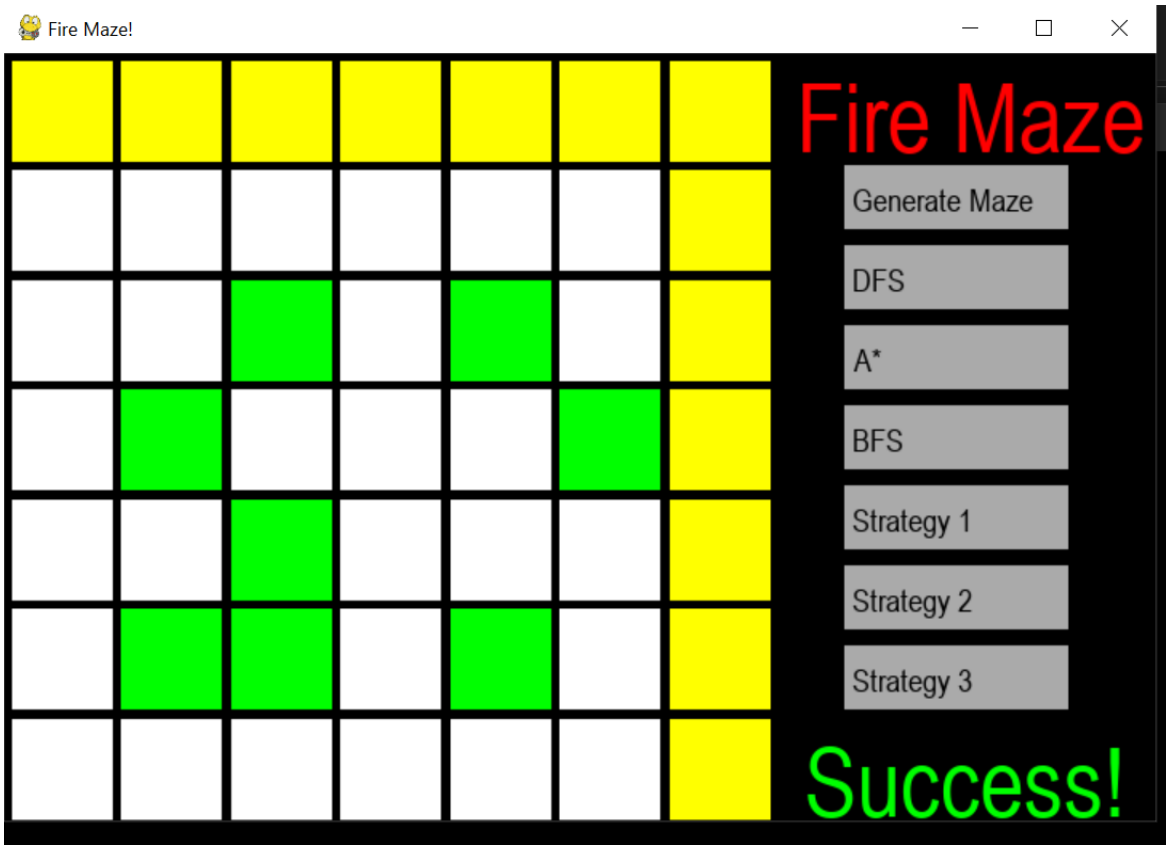

Figure 5: BFS Sample Output

Algorithm 4: a*(*maze*,*startPoint*,*endPoint*)

```
open, close = []    // Create open and closed fringe
open.append(startPoint)    // Add startPoint to visited
while True:
    temp = lowF(maze, open, startPoint, endPoint)    // Find node in open with lowest fcost
    current = open[temp]    // Push and pop from open to close based on cost
    close.append(current)
    open.pop[temp]
    if current.x == endPoint.x and current.y == endPoint.y:    // end reached, return the path
        return close
    kids = generateValidChildren(current, maze, open, close)    // collect kids of current that are not in open or close
    open.extend(kids)
    if len(open) == 0:    //No more nodes, failed, return the path so far
        return close
return close
```

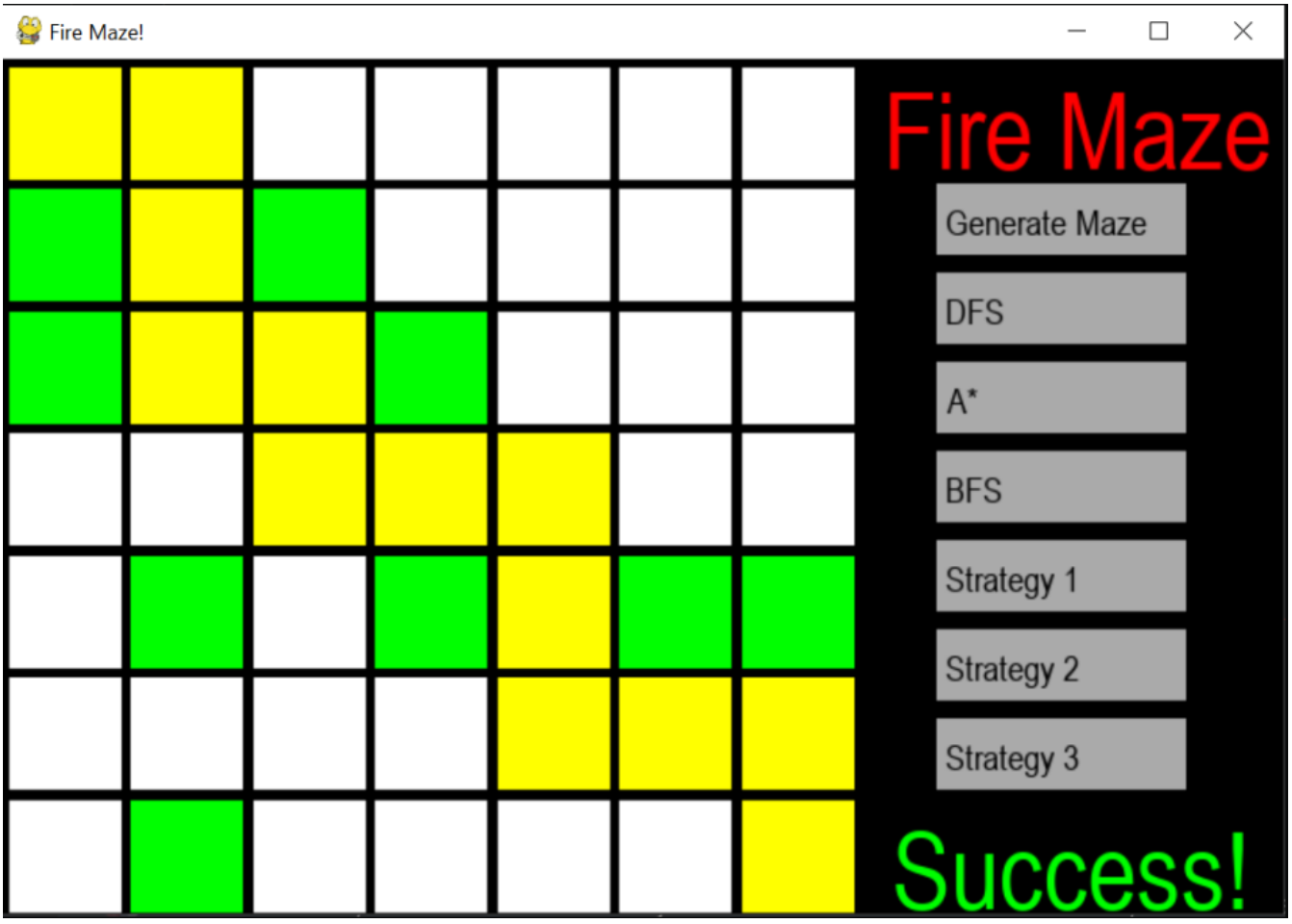

Figure 6: A* Sample Output

Since both of these algorithms promise optimality, just how much more efficient is a* in terms of space? Below is a plot of the average number of nodes explored by BFS and A* vs Obstacle Density p. Because of how long BFS takes, the graph is based on 20x20 mazes instead of the previous 50x50. It can be seen that BFS explores significantly more cells than A*. As the object density increases, the number of nodes explored by BFS decreases(as the number of nodes it can even reach decreases) and the number of nodes explored by A* increases as there are more obstacles in it's path. It should be noted from the previous DFS sucess graph that

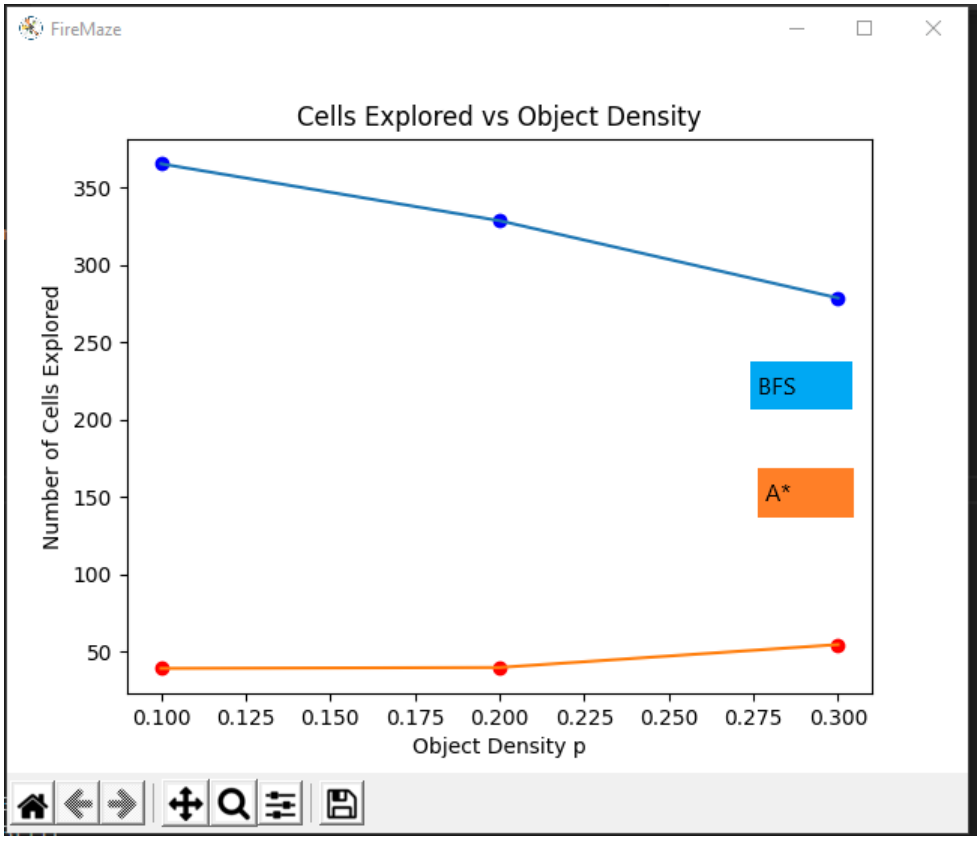

Figure 7: Obstacle Density vs Number Of Nodes Visited

at an object density of 0.4 and above, the chances of success is quite low. For that reason, on the graph, we only included data up until 0.3. For all of those data points, when there is no path from S to G, BFS and A* should visit the same number of nodes, that is every node it can reach before returning the fact that there is no path.

5 Problem 4: The Limits of DFS, BFS and A*

From the previous graph, we now know that A* will visited exponentially less nodes when compared to BFS. How about DFS? When it comes to how the dfs function take, there is a giant standard deviation as it visiting very few nodes and visiting almost every single node are both possibilities. One thing to note is that regardless of the dimensions, successes are possible. For instance if all children are occupied from the start, then the all three algorithms will return almost instantly. Because of this, we measured the limits based off consistency. When testing the limits of our DFS algorithm we found that in a 200x200 maze, it was about to successfully perform the search within a minute about 75 percent of the time. Since it's able to complete

it fairly consistently in that time, we will say that 200 is the largest dimension we can solve at p=0.3. Naturally we are ignoring all cases when failures occur early.

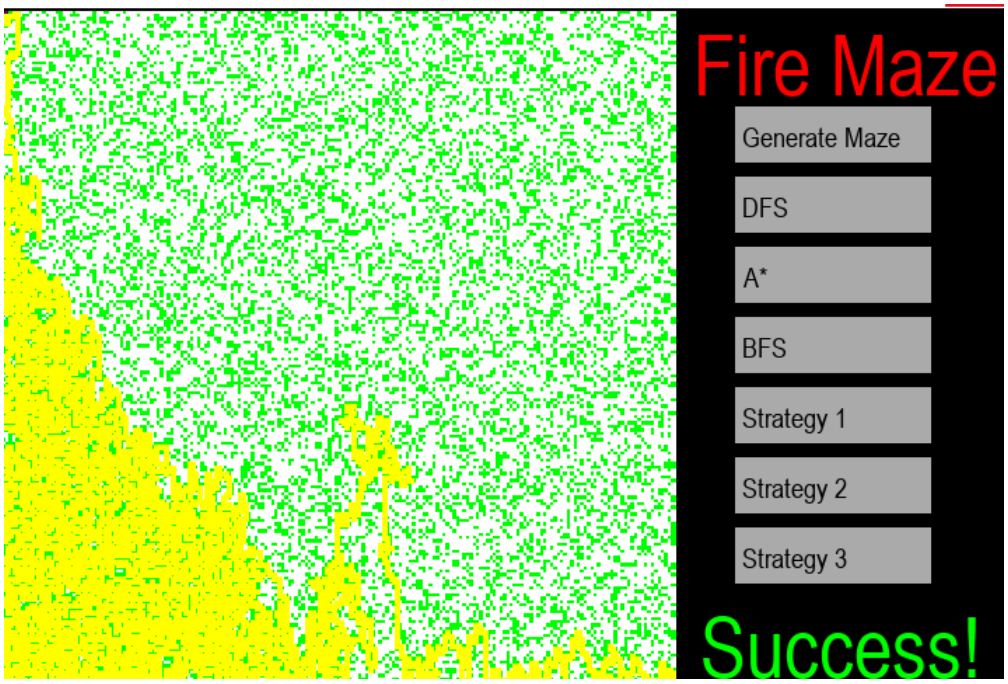

Figure 8: Example of DFS 200x200 Maze Success

BFS as expected has a much bigger fringe and thus takes a lot longer. We found that our BFS was only able to solve a 55x55 maze somewhat consistently, about 70 percent of the time. This is a big drop compared to DFS and so we can say that the largest dimension we can solve in BFS is about 55.

With A*, we found that our results were quite similar to DFS in terms of the largest dimension. Our A* algorithm was able to complete a 200x200 maze about 60 percent of the time, which is a bit lower than DFS. With this it is fair to say that our A* algorithm's largest dimension is 200, or a little less if we are more strict on the success rate. *Note: runs were all done on the GUI which generates a path so outside influenced the result

6 Problem 5: Strategy 3

As a strategy for solving these mazes, strategy 2 is pretty solid. Every time the fire changes, it recomputes the shortest path, and proceeds towards it. What it lacks is the ability to account for future fire states. Now this is tricky since there is no guarantee in terms of what the fire state will look like. Our strategy will be similar to strategy 2 but we added another protocol. When the fire is nearby steps away, the agent will become cautious. The strategy is built on the assumption that the agent can somewhat sense the fire when it's near. Perhaps he can feel the heat, or maybe he can see it from the cell he's standing in. When the agent chooses the most optimal route from the fringe, if he realizes that it's 1-2 squares away from the flame, he

will skip it in the fringe and choose the next most optimal cell in the fringe (the best heuristic). In other words, the agent will be prioritizing survivability and will attempt another route when the currently most optimal one is 1-2 squares away from a fire. (The agent will not try to beat the fire but will instead avoid it). This safety net is how the agent accounts for future fire states. One could say the agent will to some extent maintain his distance from the fire. We dedicate this protocol to one of our favorite dogs, so called it the Locally Unpleasant Navigation Avoidance Protocol. L.U.N.A. Protocol for Short.

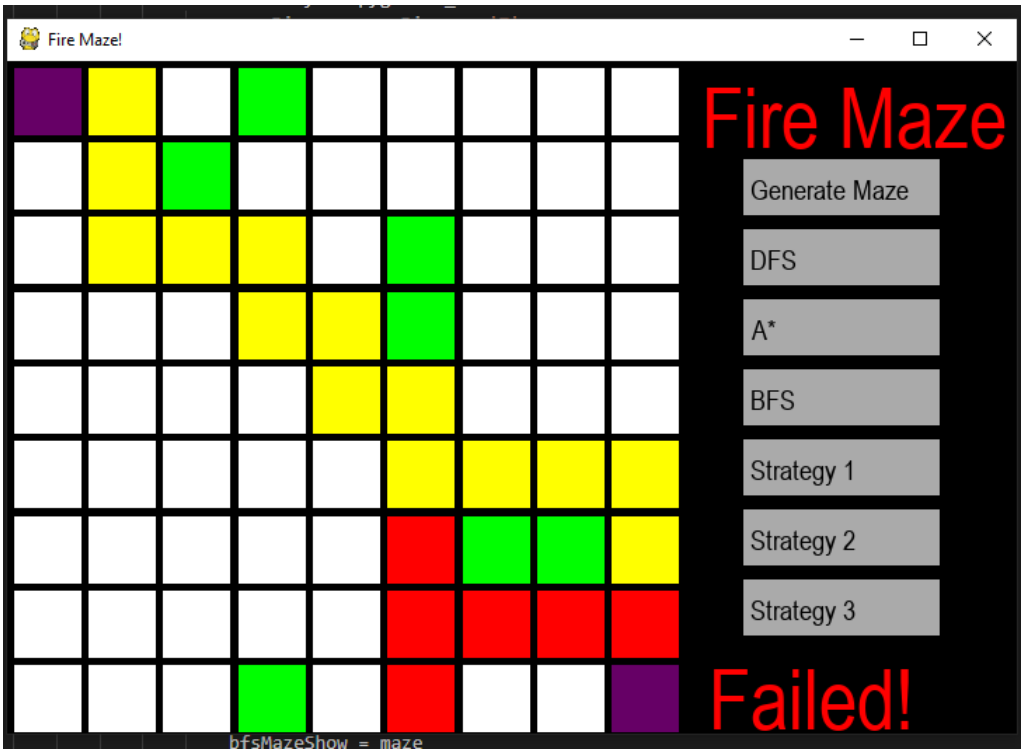

Figure 9: Sample Output of Strategy 3

7 Problem 6: Comparison of Strategies

The given strategies can all be found in the firemaze.py file. Note that from figure 4, we can see that at p = 0.3, about 40 percent of mazes are impossible mazes to solve, thus significantly hurting the strategy's successes. Below are graphs of of the average success rate vs flammability q at p = 0.3. Because of this, the differences in success rates are only slight as shown in the below graphs. Data is taken from 500 simple trials.

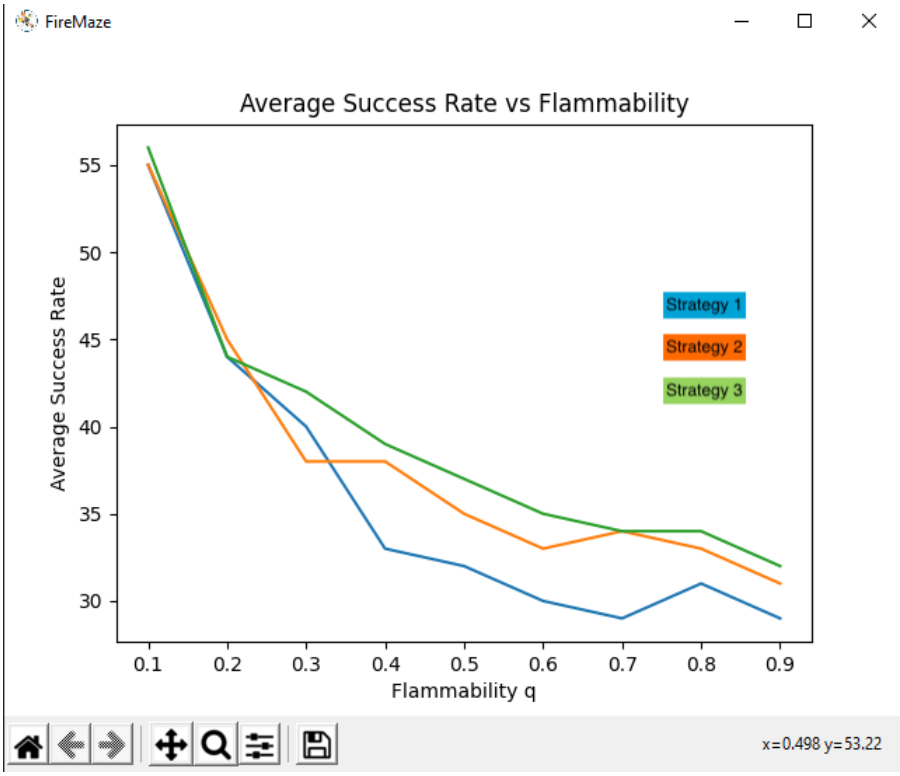

Figure 10: Success vs Flammability

It can be seen that strategy 1 and 2 have nearly the same success rate as there are only very few edge cases where strategy 2 is better than strategy 1. It takes into account the current state of the fire so it will not ram into the flames. However, especially at a higher flammability, this advantage is not as beneficial because the corrections made are not able to keep up with the higher flammability.

Using the graph we can see that all the strategies look similar near the beginning of the graph and diversify as flammability increases. This is because strategy 2 and 3 change more based on the amount of fire there is to work around. Since strategy 3 works with fire more exhaustively than strategy 2, it will show more difference as the amount of flammability increases.

Our strategy 3 is more effective at avoiding the fire than strategy 2 in that it can take alternate routes. In theory, strategy 2 could possibly take other routes as well, but it is a lot less effective as it is based strictly on the current state of the fire. In the end, their success rates are quite similar as many of the cases of failure are unavoidable. An example of this is where strategy 3 heads towards a dead end. If this were to happen, strategy 3's ability to predict where the fire will spread can not save the agent if the fire blocks the path before the agent can take sufficient action. In short, there is more accountability for future states, and the agent will be "more cautious" in terms of our strategy 3. The only downfall of strategy 3 when compared to strategy 2 and strategy 1 is strategy 3 takes a lot of time. In a real fire, it will become increasingly hard to calculate the future path of time fire with such limited time. Essentially strategy 3 would only work in real life if you could calculate the fires path faster than the spread of the fire at all times.

8 Problem 7: Unlimited Resources

If we had unlimited computational resources, we could easily power up our strategy 3. Unlimited Computational resources means we can easily look down the entire tree. Since we are placing emphasis on survivability, we can have the agent start from paths furthest away from the fire and choose the shortest path among those faraway paths. This is assuming the agent has some heuristic of where the fire begins, so a power up from our strategy 3. In other words, the agent will be choosing the path that would ideally allow him to walk around the fire while still being the most optimal of the possible paths.. Naturally, it's very possible this isn't necessary. Since we have unlimited computational resources, the agent will start off computing the shortest path normally, and if he finds that there is no way for the fire to reach him based on distance, the agent would just take that route.

We can actually add on to this by taking hypothetical worst case scenarios. We can have the agent assume the fire is guaranteed to spread in each cell and proceed. Should that fail, he can run simulations of 0.9 flames and proceed. Naturally, there are limits to this as we don't know what the maze looks like in terms of the occupied cells and this is assuming we have a heuristic of where the fire is. These are all means of gathering information with the computational strength. At it's core, what the agent is doing is finding the path that could lead to the end outside of the fires area of potential takeovers. Our strategy 3 would be powered up by including all of these factors.

9 Problem 8: Limited Resources: 10 Seconds

On the opposite end of the spectrum, if we only had 10 seconds to make a move, then we are in fact limited. We could follow the same thought process as strategy 3 where we predict whether the fire's progression would intersect with the future path. The only difference we need to make to limit the time that us used to predict the fires path to a little less than 10 seconds. This will make sure that each step takes 10 seconds or less.

The question then becomes, when we only have 10 seconds, is it 10 seconds of thinking, or 10 seconds to execute the part of the algorithm. If it's the former, we will use the entire time, as much as we can, to factor in future fire states, and increase the "cautiousness" of the agent. If it's the latter, we would need to test and utilize the time so there's just enough time to finish the move at 10 seconds.

10 Additional notes

All code is written by Jerry Yang and Sanjay Koduri.

How to run Code:
Run startMaze and when the GUI appears, type the size you want for the grid. Then click generate maze and then select what pathfinding algorithm you want to run.

Contributions-
Code:
GenMaze- Jerry
Testing and Diagnostics Code - Sanjay

DFS-Jerry
BFS-Sanjay
A*-Sanjay
Strategy 1- Jerry
Strategy 2- Jerry
Strategy 3- Both
Pygames UI- Both

Write up:
In general the write up was done together in latex with Overleaf's feature of sharing and editing at the same time. Graphs and pseudo code were generally done by Jerry and fixed up by Sanjay. Data was all gathered together.