Maze On Fire

Christopher Nguyen
Vatche Kafafian

February 19, 2021

1 Overview

In this writeup, we are observing two main sections. In the first part, we are analyzing search algorithms such as DFS, BFS, and A* through randomly generated mazes, produced by our algorithm in Python. In the second part, we are implementing strategies to solve a maze on fire while also creating a method of our own.

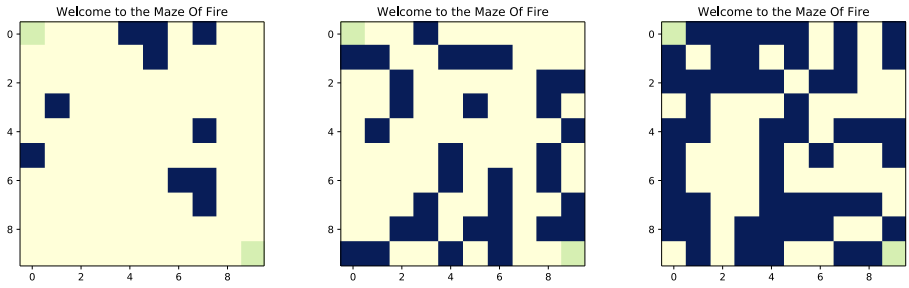
2 Section 1 - Search Algorithms

2.1 Generation of the Maze

The code below is what is used to generate our maze. We have imported matplotlib as well as random, so that we can randomly generate as well display our maze for this analysis.

Code:

```
import matplotlib.pyplot as plt
import random

dim = 10 #Dimensions of the maze
p = 0.2 #Probability of cell being occupied

#Generating Maze by inputting values randomly
def maze_generator(dim, p):
    maze = []
    for i in range(dim):
        maze.append([])
        for j in range(dim):
            value = random.random() <= p # can either be 0 or 1
            maze[i].append(value)
    maze[0][0] = 0.2 #Start Point
    maze[-1][-1] = 0.2 # End Point
    return maze

#Initializes the maze with randomized values
maze = maze_generator(dim,p)

#creating the size of the figure and each individual square in the grid
Grid = plt.figure(figsize=(dim,dim)).add_subplot(224)
Grid.set_title('Welcome to the Maze Of Fire')

#Display the data as an image
Image = plt.imshow(maze)

#Colormap of the generated grid ranging from yellow to blue
#Yellow equals free space (0)
#Dark Blue equals unavailable Space(1)
Image.set_cmap('YlGnBu')
```



a) p = 0.1 b) p = 0.3 c) p = 0.5

Figure 1: Three Example Mazes

The code above generates these mazes as shown in the following images. The initial and goal state are marked in light green, while the obstacles are highlighted in dark blue. In addition the yellow slots are the slots that are open or available.

In Figure 1, the images are randomly generated based on probability p. The image on the left has a p of 0.1. The image in the center has a p of 0.3. The image on the right has a p of 0.5. As shown in the figure, as the p increases the number of obstacles increase making the maze more difficult or impossible to solve as it is increasing.

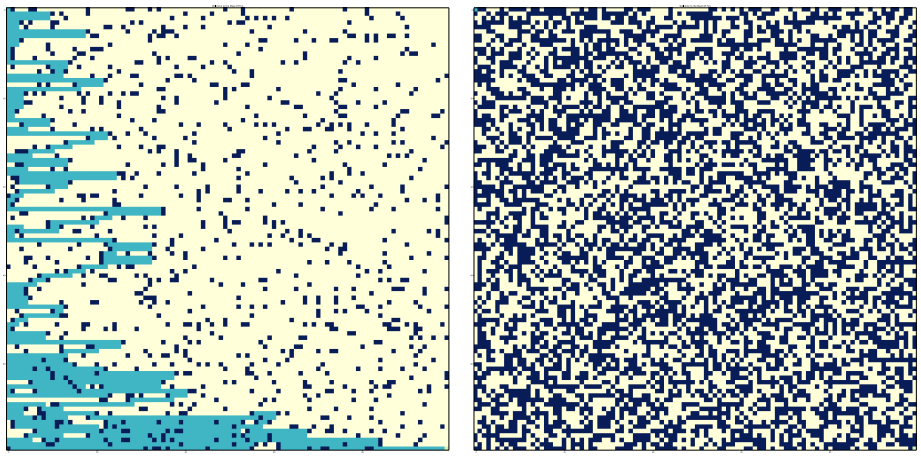
2.2 DFS Algorithm

In this subsection, we are going to evaluate our DFS algorithm, which is used to see if the goal state can be reached from the initial state in the maze. The code we used to implement our algorithm is shown below.

Code:

```
# Function that implements DFS
# Parameters and the starting and goal coordinates represented as tuples
def dfs_search(start, goal):
# initialize the stack
    fringe = deque()
    closed_set = []
 # add the starting coordinates to the stack
    fringe.append(start)

# traverse the plot until the fringe is empty
    while fringe:
```

```
# the current position is the position at the top of the stack
        current = fringe.pop()
# the goal coordinates have been found
        if current == goal:
            return True
        else:
# the current coordinates have not been visited
            if maze[current[0]][current[1]] != 0.5:
# all neighboring coordinates are added to the fringe
                generate_valid_children(current, fringe)
# visited coordinates are marked and placed in the closed set
                arr[current[0]][current[1]] = 0.5
                closed_set.append(current)

    # the goal cannot be reached
    return False

    ////////////////////////////////////////////////////////////////

    # Helper method used in DFS to add valid children to the fringe

    def generate_valid_children(current, fringe):
    #(north, south, east, west) {check the row above, check the row below, 0, 0}
    row_direction = [-1, 1, 0, 0]

    # {0, 0, Check the column to the right, check the column to the left}
    col_direction = [0, 0, 1, -1]

    # loop through every possible coordinate
    for i in range(0,4):
        row = current[0] + row_direction[i]
        col = current[1] + col_direction[i]

    # if the row coordinate is out of bounds, it cannot be added to the fringe
        if row in range(0,dim):

    # if the col coordinate is out of bounds, it cannot be added to the fringe
            if(col in range(0,dim)):

    # Check if the (row, col) coordinate is a obstacle
                if(arr[row][col] != 1):

    # Check if the coordinates are already visited
                    if(maze[row][col] != 0.5):
                        fringe.append((row,col))
```



a) p = 0.3 b) p = 0.5

Figure 2: Two Example DFS Mazes

The DFS code above, in coalition with the code for generating a maze, produces these two grids shown above. As you can see the maze with a probability of 0.3 has a path available, shown by using DFS. While the maze with a probability of 0.5 has no path available, which was checked by using DFS. In figure two, we also see that when the probability is increased the maze becomes more difficult or just impossible to solve. On the next page, we are going to display a plot that is derived by our analysis upon obstacle density p vs the probability that G can be reached from S.

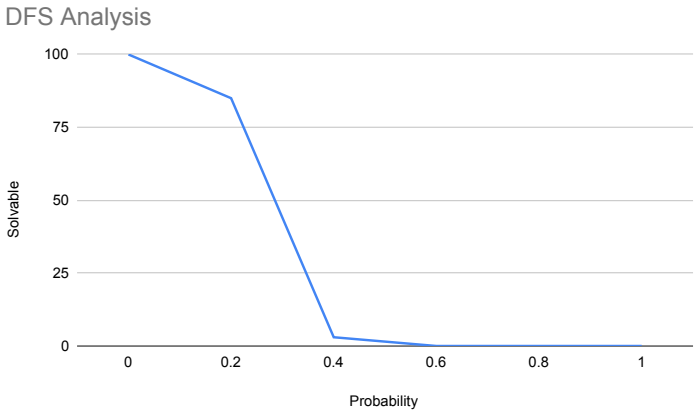


Figure 3: P VS Probability

This plot indicates the percentage of completed mazes within regards to the probability or density of the maze while using DFS. To generate this plot, we used a sample size of 100 randomly generated mazes for each density. The dimension that we chose to use was a 100 due to generating the most efficient and effective data. From this plot, we can conclude that the maze solvability depends on P. This is because as the probability increases the number of solvable mazes decreases until it eventually reaches 0 at a certain threshold, which in this graph can be seen to be p = 0.6 for a dim of 100.

In this situation it is better to use DFS than BFS because although BFS finds the most optimal path, DFS is faster. In this case our dimension was 100 so our target is not very close to the source, which means BFS would take a lot more time then DFS to compute making DFS more efficient. In the next subsection we will continue to analyze other search algorithms such as A* and BFS.

2.3 BFS and A* Analysis

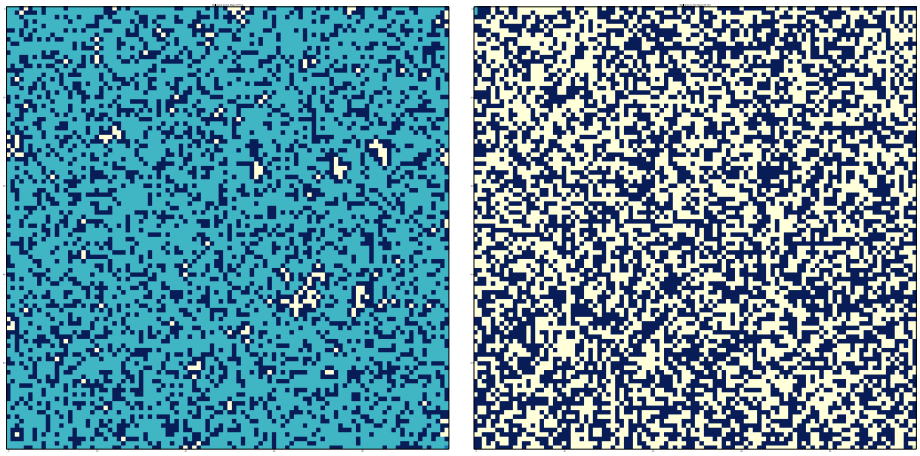
In this subsection, we are going to evaluate our BFS and A* algorithms, which is used to see if the goal state can be reached from the initial state in the maze. The code we used to implement both algorithms is shown below.

Code BFS:

```
#Function that implements BFS
# Parameters and the starting goals represented as tuples
def bfs_search(start, goal):
    # initialize the queue
    fringe = deque()
    closed_set = []

    #Add the starting coordinates to the Queue
    fringe.append(start)

    #Traverse the plot until the fringe is empty
    while fringe:
        current = fringe.popleft()

        # The goal coordinate has been found
        if current == goal:
            maze[-1][-1] = 0.7 # End Point
            maze[0][0] = 0.7 # Start Point
            print("Number of nodes visited: ", len(closed_set))
            return True
        else:

            #The coordinates have not been visited
            if maze[current[0]][current[1]] != 0.5:

                #All neighboring valid coordinates are added to the fringe genera

                #visited coordinates are marked and placed in closed set maze[cur
                  closed_set.append(current)

    print("Number of nodes visited: ", len(closed_set))
    #Return False if the goal cannot be reached
    return False

    ////////////////////////////////////////////////////////////////

    # Helper method used in BFS to add valid children to the fringe
    def generate_valid_children_B(current, fringe):
    # (north, south, east, west) {check the row above, check the row below, 0, 0
```

```
row_direction = [-1, 1, 0, 0]

# {0, 0, Check the column to the right, check the column to the left}
col_direction = [0, 0, 1, -1]

# loop through every possible coordinate
for i in range(0,4):
    row = current[0] + row_direction[i]
    col = current[1] + col_direction[i]

# if the row coordinate is out of bounds, it cannot be added to the fringe
    if row in range(0, dim):

# if the col coordinate is out of bounds, it cannot be added to the fringe
        if(col in range(0,dim)):

# Check if the (row, col) coordinate is a obstacle
            if(maze[row][col] != 1):

# Check if the coordinates are already visited
                if(maze[row][col] != 0.5):
                    fringe.append((row,col))
```

On the next page we will see what this code generates when implementing it with the algorithm for maze generation.



a) p = 0.3 b) p = 0.5

Figure 4: Two Example BFS Mazes

The BFS code above, in partnership with the code for generating a maze, produces these two grids shown above. As you can see the maze with a probability of 0.3 has a path available, shown by using BFS. While the maze with a probability of 0.5 has no path available, which was checked by using BFS. In the figure, we can also see that when the probability is increased the maze becomes more difficult or just impossible to solve. In our case it became impossible because there was no path available for this maze of p =0.5. Now we are going to display the algorithm for A* and show example graphs that were generated similar to DFS and BFS.

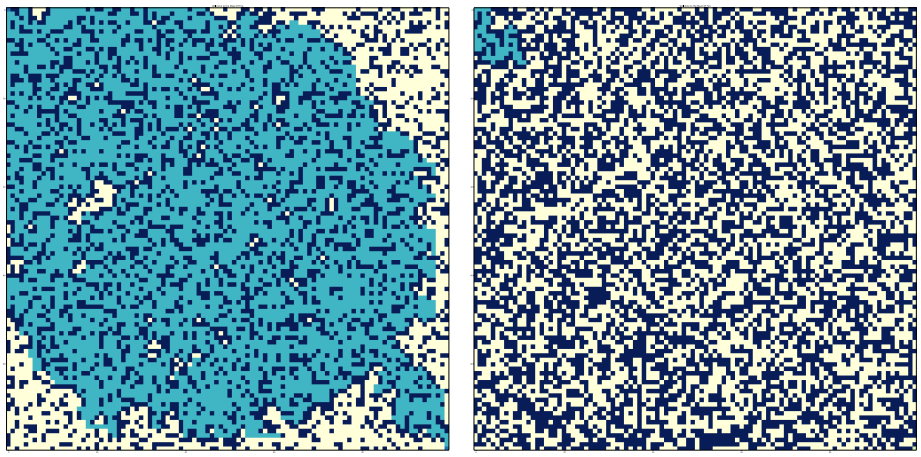
Code:

```
#Function that implements A*
def a_star(start, goal):
# generate priority queue for a star
    fringe = PriorityQueue()
    closed_set = []

# add  starting coordinates to queue with heuristics
    fringe.put(start)
    while not fringe.empty():
        current = fringe.get()
        if current[3] == goal[3]:
```

```
            maze[-1][-1] = 0.7
            maze[0][0] = 0.7 #Start Point
            print("Number of nodes visited: ", len(closed_set))
            return True
        else:
            if maze[current[3][0]][current[3][1]] != 0.5:

# all neighboring coordinate are added to fringe and heuristic is calculated for
                generate_valid_children_A(current, fringe, start, goal)
                maze[current[3][0]][current[3][1]] = 0.5
                closed_set.append(current[3])

    print("Number of nodes visited: ", len(closed_set))
    return False

////////////////////////////////////////////////////////////////

#Helper Method that generates valid children for A using euclidean distance
def generate_valid_children_A(current, fringe, start, goal):
 #(north, south, east, west) {check the row above, check the row below, 0, 0}
    row_direction = [-1, 1, 0, 0]

# {0, 0, Check the column to the right, check the column to the left}
    col_direction = [0, 0, 1, -1]
    current[2] = current[2] + 1

# loop through every possible coordinate
    for i in range(0,4):
        row = current[3][0] + row_direction[i]
        col = current[3][1] + col_direction[i]

# if the row coordinate is out of bounds, it cannot be added to the fringe
        if row in range(0, dim):

# if the col coordinate is out of bounds, it cannot be added to the fringe
            if(col in range(0,dim)):

# Check if the (row, col) coordinate is a obstacle
                if(maze[row][col] != 1):

# Check if the coordinates are already visited
                    if(maze[row][col] != 0.5):

# exact distance from current to start
                        g = current[2]
```

```
#estimated distance from current to goal
                        h = euclidean_distance_A(row, col, goal)
# the total heuristic
                        total = g + h
                        fringe.put([total, h, g, (row, col)])


////////////////////////////////////////////////////////////////////

#Helper Method that calculates the Euclidean Distance between two vectors
def euclidean_distance_A(row, col, coor):
    a = np.array([row, col])
    b = np.array([coor[3][0], coor[3][1]])

    return norm(a-b)
```

On the next page, we will show examples of A* being implemented on randomly generated graphs.



a) p = 0.3 b) p = 0.5

Figure 5: Two Example A* Mazes

The A* code above, implemented with the code for generating a maze, produces these two grids shown above. As you can see the maze with a probability of 0.3 has a path available, shown by using A*. While the maze with a probability of 0.5 has no path available, when using A*. In the figure, we can also see that when the probability is increased the maze becomes more difficult or just impossible to solve. In our case it became impossible because there was no path available for this maze of p = 0.5.

As you can see for the probability for 0.3 A* completes the maze with less nodes visited compared to the BFS algorithm above with the same p value. On the next page we will look at a plot that shows the average difference in nodes visited for both algorithms and observing why this is the case.

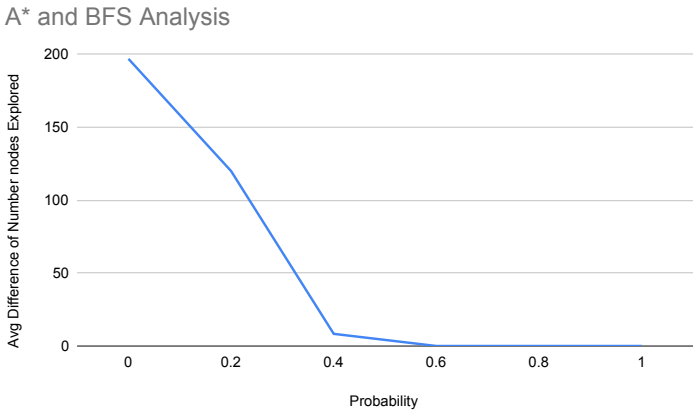


Figure 6: Avg difference in nodes visited by A* and BFS VS Probability

This plot indicates the avg difference between nodes visited between A* and BFS within regards to the probability or density of the maze. To generate this plot, we used a sample size of 100 randomly generated mazes for each density. The dimension that we chose to use was a 100 due to generating the largest possible grid, and the most efficient and effective data. From this plot, we can conclude that the A* search algorithm significantly visits less nodes than the BFS search algorithm. This is because of the use of heuristics. A* is a very similar algorithm to BFS, except that it uses a priority queue. A* takes account of the exact distance from the initial state to current state and also takes in a estimated distance from the current state to goal state to form a heuristic. Using the heuristic, we are able to generate the priority queue for the fringe. The difference shows how effective the extra book keeping is for the heuristic. We see that as the probability increases the difference decreases due to the increasing obstacles limiting the nodes the algorithms are able to visit until eventually the mazes become impossible to solve. In the case where the mazes are impossible to solve the difference is zero, because the algorithms cannot completely run and visit all the possible nodes making the difference in nodes visited negligible in this situation. However, the easier mazes show that the difference is quite significant and that A* visits a less amount of nodes than BFS, meaning that the heuristic is admissible in this situation.

In the final subsection, we are going to analyze all the algorithms and see the largest dimension we can solve at $p = 0.3$ in less than a minute.

2.4 Search Algorithm Analysis

In this subsection we are going to evaluate all the three search algorithms we covered so far. We are going to observe the largest dimension that the algorithm can solve a p = 0.3 in less than a minute.

When we ran each of the search algorithms multiple times we were able to obtain a estimate of the largest dimension that each of the methods can solve in less than a minute. For DFS, the algorithm was able to solve a maze of dim equal to 7300 in 58.55252766609192 seconds. Next, BFS was able to run through a maze of dim equal to 3400 in 58.851914167404175 seconds. Finally, A Star was able to produce a successful result with a maze of dim equal to 1300 in 57.15948009490967 seconds. This proves that even though BFS produces the most optimal route, DFS will obtain a path faster as the dimensions become larger. In addition, A Star's addition heuristics required additional overhead causing it to run slower than BFS, however, it visits less nodes when acquiring its solution. In the end, BFS finds the most optimal solution, while DFS achieves the solution the fastest in bigger dimensions.

3 Solving A Maze On Fire

In this second section, we are going to evaluate three different strategies to solve a maze. During the evaluation, we are going to analyze how the three strategies compare to each other in terms of average success versus the flammability of the fire in the maze q.

For the first strategy, the algorithm solves for the shortest path from the start point, the upper left, all the way to the end, the lower right. Once it finds the shortest path no matter where the fire spawns, the algorithm follows until it reaches the end of the maze or burns. This strategy does not modify its initial path in any way as the fire changes. In order to implement this strategy, we implemented the BFS function from the first section. We modified it so that it does not show every node that it has visited but the final optimal path that it generates. When the optimal path is returned we commence the strategy where we follow the path with no regard of the fire until we burn or reach the end.

Next, we have the second strategy. In the second strategy, at every iteration, the algorithm re-computes the shortest path from the current position to the end of the maze, based on the current state of the maze and the fire. Essentially, the algorithm follows this new path for one time step, then regenerates a new path. This strategy constantly re-adjusts its strategy based on the spread of the fire. If the agent gets trapped with no path to the goal and dies. To implement this strategy, we also used the BFS function from the first section. However, unlike the first strategy we regenerate a new optimal path for every single iteration and follow it for one time step. This strategy keeps modifying

its path at every time step until there is no path left to the goal or until it reaches the end.

Finally, the third strategy is one we came up with by implementing the first two strategies. However, unlike the first two strategies it accounts for the unknown future of the fire. First, we generate the original optimal path like we did in strategy one using BFS and we generate the initial fire position. Then, we account for the unknown future by taking the euclidean distance of the most recently generated fires to each node on the generated path and assume the worst case scenario, in which the fire will spread at every time step. We calculate the euclidean distance and round it down in order to estimate the fastest arrival for the fire so that we are prepared for all possible evolutions of the fire. In addition when we generate the euclidean distance, we only account for the most recently spread fire so that we can limit our computational resources and the fact that the children fire have the best probability of being the closest to each node in the current time step. For every time step, we then would decrease the euclidean distance of each node after the current node on the path by one, in the case that the fire spreads each iteration and since we no longer need to account for past iterations. After decreasing the euclidean distance, we would continue the same process on the same path until we read a node on the path with a euclidean distance that is less than or equal to 0, which tells us that there is a possibility that the fire has already reached that position assuming the worst case scenario. If this is the case, we would then implement a similar approach to strategy two, where we use BFS again to regenerate a new optimal to the end and we regenerate the new euclidean distance for each node on the new path and repeat the process until there is no path left, hit a fire, or reach the end of the maze. We call this strategy H.I.F.I.V.E., which is an acronym for High Intensity Fire Interpretation Vector Evader. We came up with this name because the strategy implements vectors to store the euclidean distance of the fires, which is then used to interpret the intensity of the fire and tries to evade it.

After implementing all the strategies, we came up with a environment to analyze the average success rate versus the flammability q. We set the probability of the obstacles p to equal 0.3. Then we chose to generate a grid with a dim = 10, due to it being the largest common dim among each strategy and due to it being the most efficient way to generate the grids for each strategy. For each value of q, we generated a sample size of 100 and took the average of each for every strategy and plot it. The plot showing the comparisons between each strategy is shown below on the next page.

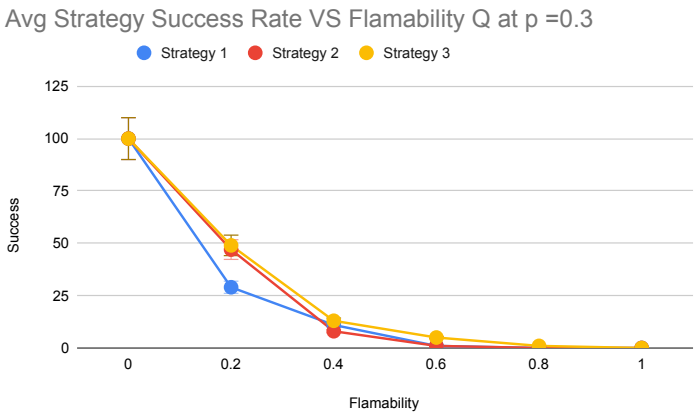


Figure 7: Avg success rate versus flammability q

As displayed by the plot, when the flammability is low, we can see the largest difference in the performance in the strategies. Strategy three and strategy 2 exceed strategy 1 by a large margin due to their ability of taking account of the fire and changing up their route, while strategy one tries to brute force and dies due to the fire most of the time. Then comparing strategy 2 and 3 when the q is low, we can see that strategy only exceeds strategy 2 by a small amount. Strategy 3 outperforms strategy 2 due to one main differences. The main difference is that strategy 3 does not need to recompute a new path at every time step saving much more time and processing. This is because strategy 3 takes account of the future of the fire so that it can continue on the most optimal path for as long as possible, then recomputes only when it must saving computational resources. This allows strategy 3 to improve time costs compared to strategy two and improve survivability. Then the graph shows that when q is increased the margin of difference between all strategies decrease. This is due to the increased rate at which the fire spreads at each time step. Since the rate of fire is increased, strategy 2 and strategy 3 cannot compute as many alternative paths, in which they ultimately get trapped and die. This lowers their success rate to be on par with strategy one, until eventually the flammability is to high for any strategy to reach the end. Overall, strategy three was able to exceed strategy two and strategy one in success as shown in the graph and was able to reduce the time costs of strategy two by only regenerating paths when necessary.

We are able to expand on strategy three, in the case that we have unlimited computational resources at our disposal. With unlimited computational resources we can improve our strategy by taking a better account of the future by calculating the exact distance from the closest fire node instead of the euclidean distance from the most recently generated fire nodes. By using BFS a myriad of times from each node to each generated fire, we can find the closest

fire node to each node in the path and return the exact distance. This helps us improve the exact moment the algorithm regenerates a new path so that instead of only regenerating for the fastest time when the fire can reach the current node, we can account for a more exact time when the fire will arrive. By using unlimited computational resources to compute the exact distance of the closest fire from each node, we can improve our algorithm.

Finally, for the case in which we would only have only ten seconds to compute each move rather than being given additional computational resources, we would change our strategy to a more greedy approach for faster decision making. Rather than trying to compute for the probability when the fire would arrive, we would implement an approach similar to A* except with a different heuristic. The heuristic would take into account two main conditions for its cost function, which are the distance from the closest fire and the distance from the destination. We would wanna choose the next time step in which we would maximize the distance from the closest fire and minimize the distance from the destination. Although, this algorithm may generate decisions and reach the destination faster, it has a fatal flaw. Since this algorithm chooses its next time greedily, it has a likely chance to run into a dead end and causing its own demise. This concludes our analysis on the strategies solving the maze on fire, our code for each strategy will be submitted separately along with the entirety of the code for the first section of the report.