AI Project 1: Maze, DFS, BFS, and A*

Daniel Ying dty16 440:06, Kyle VanWageninge kjv48 440:05

May 24, 2021

Submitter:

Honor Code:

I abide to the rules laid in the Project 1: Maze on Fire description and I have not used anyone else's work for the project, and my work is only my own and my group's.

I acknowledge and accept the Honor Code and rules of Project 1.

Signed: Daniel Ying dty16, Kyle VanWageninge kjv48

Workload:

Daniel Ying: Wrote, compiled, and organized of the report. Coded the report in latex. Translated the coding into Pseudocode.

Kyle VanWageninge: Did a majority of the coding. Implementing the 3 search algorithms, extra GUI implementation as well as code for the 3 strategies.

Together: Discussed how to implement the overall project. Discussed and brainstormed how to implement strategy 3. Both implemented code to find results for the report problems. Answered report questions together and analysis.

Problem 1

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

Answer:

Algorithm 1 Generate Maze(rows, columns, p)

```
Create 2D array named Maze[rows][columns]
for i = 0, ..., rows do
    for j = 0, ..., columns do
        Create grid in Maze[i][j]
        if i = 0 and j = 0 then
            Make grid's color green as START
        end if
        if i = rows − 1 and j = columns − 1 then
            Make grid's color red as GOAL
        else
            Create Source as random integer from 1 to 100
            Create Probability as p · 100
            if Source is less than Probability then
                Set grid color as black
            else
                Set grid color as white
            end if
        end if
    end for
end for
Return Maze
```

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'.

Answer:

DFS would be the better algorithm than BFS, because:

1) The START node is located at Maze[0][0], while the END node is located at Maze[row-1][column-1], which is the farthest node from the START node. Thus in this case, we would choose DFS rather than BFS, since DFS would reach the farthest point faster than BFS.

2) Because our Maze is finite, meaning that it is too "deep" and would end eventually, DFS would be a better choice than BFS.

3) Since set in a Maze where there are a relative large number of possible moves for each grid, it would be more efficient to go towards a direction from a grid rather than explore all paths for the grid. And if that direction leads to the GOAL, then the Maze would end. Thus, DFS would be a better choice than BFS.

Analysis:

The results from our DFS search algorithm generates a relatively high (above 0.95 success rate) success rate while p is less than or equal to 0.2 (As seen in Figure 1, Graph for Problem 2). but as p increases in density, DFS's success rate decreases. And once p reaches or exceeds density 0.4, the DFS search algorithm success rate drops close to or to 0. We observe this low success rate is caused by the walls (obstacles) blocking all possible paths for our robot to start from START and reach the end, GOAL.

Graph:

Dim: 1000x1000. Each p was ran 100 times using given dim. From p = 0 to 1 at increase of 0.1 each step

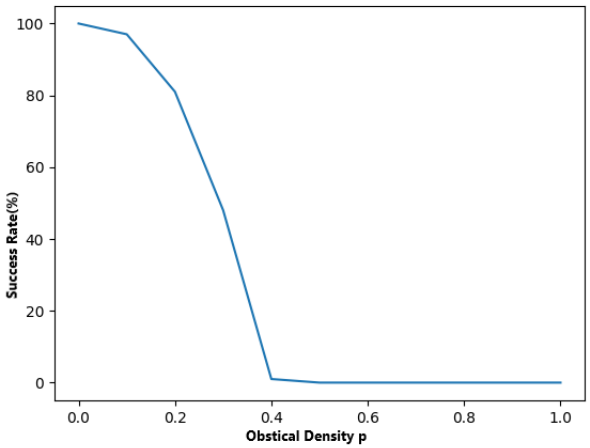

Figure 1: Graph for Problem 2.

Algorithm 2 DFS Runner(START, GOAL)

```
Create arrays named Fringe and Visited
Create Start as START in Maze
Set Start's PARENT as itself
Append Start to Fringe
while Fringe is not empty do
    Create Current as popped LAST NODE in Fringe
    Append Current to Visited
    if Current is GOAL then
        Return Visited
    end if
    if Grid above Current is True then
        Create node named Next as grid above Current
        Set Next's PARENT as Current
        Append Next to Fringe
    end if
    if Grid left of Current is True then
        Create node named Next as grid left of Current
        Set Next's PARENT as Current
        Append Next to Fringe
    end if
    if Grid below Current is True then
        Create node named Next as grid below Current
        Set Next's PARENT as Current
        Append Next to Fringe
    end if
    if Grid right of Current is True then
        Create node named Next as grid right of Current
        Set Next's PARENT as Current
        Append Next to Fringe
    end if
end while
if Fringe is empty and GOAL not reached then
    Return an empty array
end if
```

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 should this difference be?

Answer:

If no path exists that allows the robot to start from S and reach G, then the average number of nodes explored (vs obstacle density p) by BFS and A* would be the same. Looking at Figure 2, graph for problem 3, after the obstacle density reaches 0.4 and exceeds 0.4, the A* and BFS yields identical results. Thus, there is no difference between the two algorithms when no path exists from S to G.

Analysis:

For our search algorithms BFS and A*, they both yield similar success rates as DFS, which all yielded a decreased success rate once p exceeds 0.2 and drops to close to 0 success rate once p exceeds 0.4. The reasoning for this phenomenon in BFS and A* is also quite similar to that of DFS, the increase of obstacles which are the walls, blocked the robots path from START to GOAL, thus explaining why the success rate has dropped so low.

Interestingly, we noticed a difference between the three search algorithms when we look at the number of nodes the algorithms have visited. Looking at Figure 2 (Graph for Problem 3), the algorithm BFS have a higher visited nodes than A* when the obstacle density p exceeds 0.2, and this difference increases as p reaches the value 0.3. We believe the cause of the difference between the BFS and A* is caused by the nature of A*. A* explores less nodes because of the usage of heuristics instead of a regular queue like BFS. Thus A* is more efficient in visiting less nodes (less memory used) when it comes to finding a path form START to GOAL. It is also noted, once p exceeds 0.3 and closes in on 0.4, the difference in nodes visited of BFS and A* decreases. This is caused by the increase of obstacles blocking the possible paths the robot could take. Thus, leading to the decrease in nodes visited between the two algorithms, until finally, BFS and A* become identical in nodes visited once p exceeds 0.4.

Of the three algorithms (DFS, BFS, and A*), DFS is more efficient in time when finding a path from START to GOAL since it explores one node branch at a time, although the path found would not always be the optimal path. To ensure we find the optimal path, we would be using BFS, but at the cost of using more time due to the possible large size of the fringe. While for A* algorithm, it is more complicated due to the calculations on each node to get the current cost + Euclidean Distance metric. Though it is more complicated, the calculation allows A* to use less memory than the DFS and BFS.

Graph:

Completed at dim: 300x300

Ran each p 20 times using given dim

From p = 0 to p = 1 at increase of .1 each step

Ran both BFS and A* on the same maze through each run

Algorithm 3 BFS Runner(START, GOAL)

```
Create arrays named Fringe and Visited
Create Start as START in Maze
Set Start's PARENT as itself
Append Start to Fringe
while Fringe is not empty do
    Create Current as popped FIRST NODE in Fringe
    Append Current to Visited
    if Current is GOAL then
        Return Visited
    end if
    if Grid above Current is True then
        Create node named Next as grid above Current
        Set Next's PARENT as Current
        Append Next to Fringe
    end if
    if Grid left of Current is True then
        Create node named Next as grid left of Current
        Set Next's PARENT as Current
        Append Next to Fringe
    end if
    if Grid below Current is True then
        Create node named Next as grid below Current
        Set Next's PARENT as Current
        Append Next to Fringe
    end if
    if Grid right of Current is True then
        Create node named Next as grid right of Current
        Set Next's PARENT as Current
        Append Next to Fringe
    end if
end while
if Fringe is empty and GOAL not reached then
    Return an empty array
end if
```

Algorithm 4 A* Runner(START, GOAL)

Create arrays named Fringe and Visited, Create Start as START in Maze, Create Distance as (1/2)· ((Node[row]-GOAL[row])^2+ (Node[column]-GOAL[column])^2)
Set Start's PARENT as itself, Set Start's distance value as Distance, then Append Start to Fringe
while Fringe is not empty **do**
    Create Node as first node in Fringe, Append Node to Visited
    Create Current as Node
    if Current is GOAL **then** Return Visited
    end if
    if Grid above Current is True **then**
        Create node named Next as grid above Current
        Set Next's PARENT as Current
        Create Cost as number of parents from Current
        Create NewDistance as (1/2)· ((Current[row]-GOAL[row])^2+ (Current[column]-GOAL[column])^2)
        Create TotalDistance as Cost + NewDistance
        Set Next's distance value as TotalDistance, Append Next to Fringe
    end if
    if Grid left of Current is True **then**
        Create node named Next as grid left of Current
        Set Next's PARENT as Current
        Create Cost as number of parents from Current
        Create NewDistance as (1/2)· ((Current[row]-GOAL[row])^2+ (Current[column]-GOAL[column])^2)
        Create TotalDistance as Cost + NewDistance
        Set Next's distance value as TotalDistance, Append Next to Fringe
    end if
    if Grid below Current is True **then**
        Create node named Next as grid below Current
        Set Next's PARENT as Current
        Create Cost as number of parents from Current
        Create NewDistance as (1/2)· ((Current[row]-GOAL[row])^2+ (Current[column]-GOAL[column])^2)
        Create TotalDistance as Cost + NewDistance
        Set Next's distance value as TotalDistance, Append Next to Fringe
    end if
    if Grid right of Current is True **then**
        Create node named Next as grid right of Current
        Set Next's PARENT as Current
        Create Cost as number of parents from Current
        Create NewDistance as (1/2)· ((Current[row]-GOAL[row])^2+ (Current[column]-GOAL[column])^2)
        Create TotalDistance as Cost + NewDistance
        Set Next's distance value as TotalDistance, Append Next to Fringe
    end if
end while
if Fringe is empty and GOAL not reached **then** Return an empty array
end if

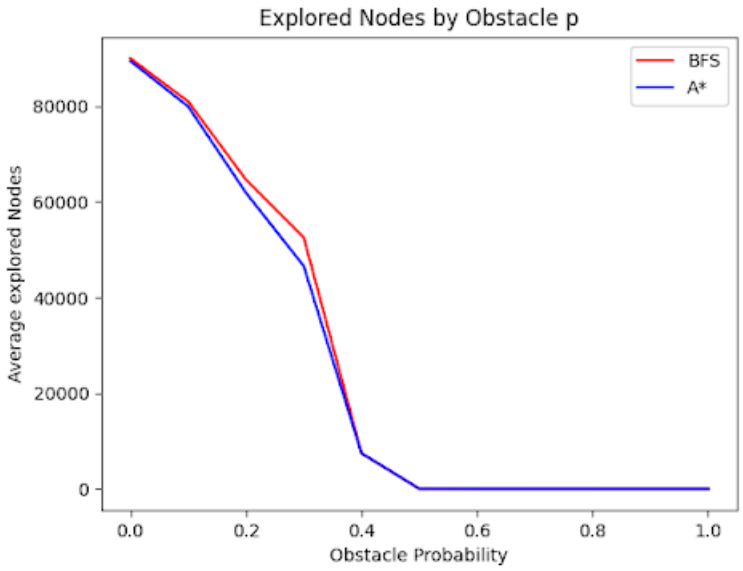

Figure 2: Graph for Problem 3.

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?

Answer:

The largest dimension we can solve using DFS in less than a minute was about 5,850 when p = 0.3. While under the condition of p = 0.3 and within one minute, BFS can go up to 5350 and A* can only go up to 4500.

As stated before in problem 3, DFS and BFS are very similar. The reason why DFS has a slightly large dimension than BFS is because DFS runs the search algorithm by node branch, while BFS searches for the optimal path that is affected by the size of the fringe. A* has a much smaller dimension than DFS and BFS, this is caused by the nature of A*, which applies the calculation of cost + Euclidean Distance metric. Though A* has a smaller dimension (only 4500), A* uses less memory than DFS and BFS.

Theoretically, the expected A* search algorithm should have produced a higher dimension due to its implementation by calculation of cost and Euclidean Distance Metric, making it more efficient than DFS and BFS. But, due to our computation of A*, the actual A* had a lower dimension than the other two search algorithms.

DFS: 5,850; BFS: 5,350; A*: 4500

Problem 5

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

Answer:

Our strategy 3 is called Danger Aversion Movement (DAM). This strategy takes into account the possible location where the current fire would be in the next step as if q = 1. We would then recompute the future fire at each step in the maze and like that for strategy 2, create a new path where it is 'least danger.'

Thus to do this, a secondary integer array was created. Then while iterating through the original maze, when ever a fire is seen, the value 4 would be set into the variable 'danger maze' to represent the fire. If the grid is a wall then the value would be 5, otherwise it would be 0. After the fire is located, we would then add 'danger levels' to each grid where the fire could possibly go to. The value that was set on this square is q. By doing so, we would add multiple q's to one grid. Thus, for example, if there are 3 fires staring down at a grid, then 3 q's would be set to the grid after running, thus the grid would have a higher 'danger level' of being covered by the fire in the next step. The search algorithm would then take in a 'danger array' and find the shortest possible path least likely to have grids burning in the next step.

This strategy would allow our robot to see the fire's potential position in the next step. Thus, the robot would also prioritize grids with lower 'danger levels' over grids higher 'danger levels' (meaning more likely to burn) in the future steps, allowing the robot to avoid burning and move around the fire more efficiently than strat2, leading to higher success. Our search algorithm will also simulate a priority queue, thus it acts like A* but with 'danger levels' instead of distance and give the optimal solution at each step.

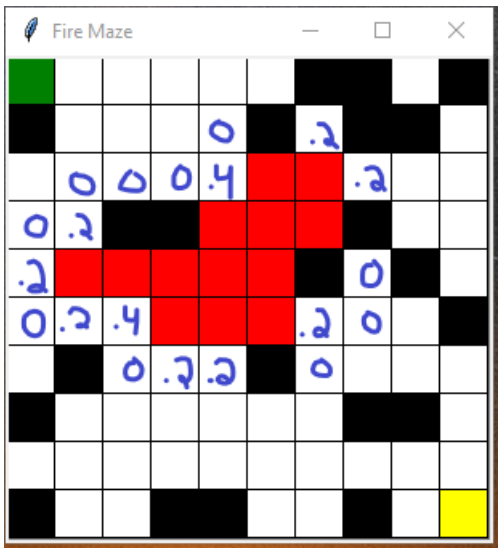

Figure 3: Example for q=0.2 after one iteration on current fire (in red). The search algorithm would then pick the smallest values to reach the GOAL and recompute each new fire at each step. For girds without values, they are 0's because they are not in danger of being touched by the fire in the next step, making them safe.

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?

Answer:

Strat 1:

Dim:150, p:0.3, q: 0.1 to 1 with 0.1 increase

Total of 20 unique mazes tested, running each maze 10 times. Each maze had to have a reachable goal, and the fire had to be in a spot where the robot could reach it, otherwise the maze would be thrown out and a new maze would be created.

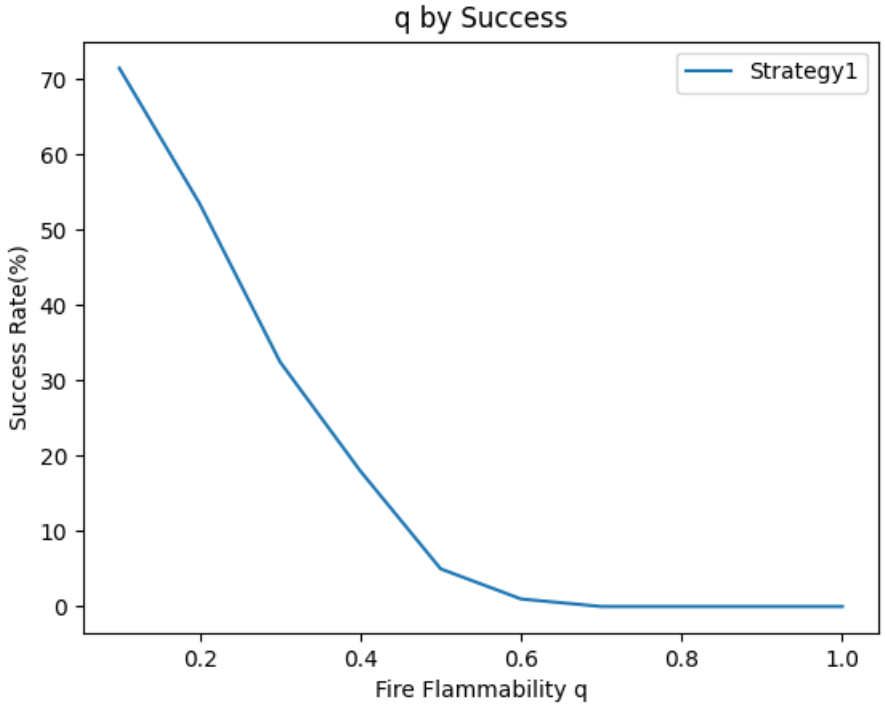

Figure 4: Strat 1 graph. Values:[71.5, 53.5, 32.5, 18.0, 5.0, 1.0, 0.0, 0.0, 0.0, 0.0]

Strat 2:

Dim:150, p:0.3, q:0.1 to 1 with 0.1 increase

Total of 20 unique mazes tested, running each maze 10 times. Each maze had to have a reachable goal, and the fire had to be in a spot where the robot could reach it, otherwise the maze would be thrown out and a new maze would be created.

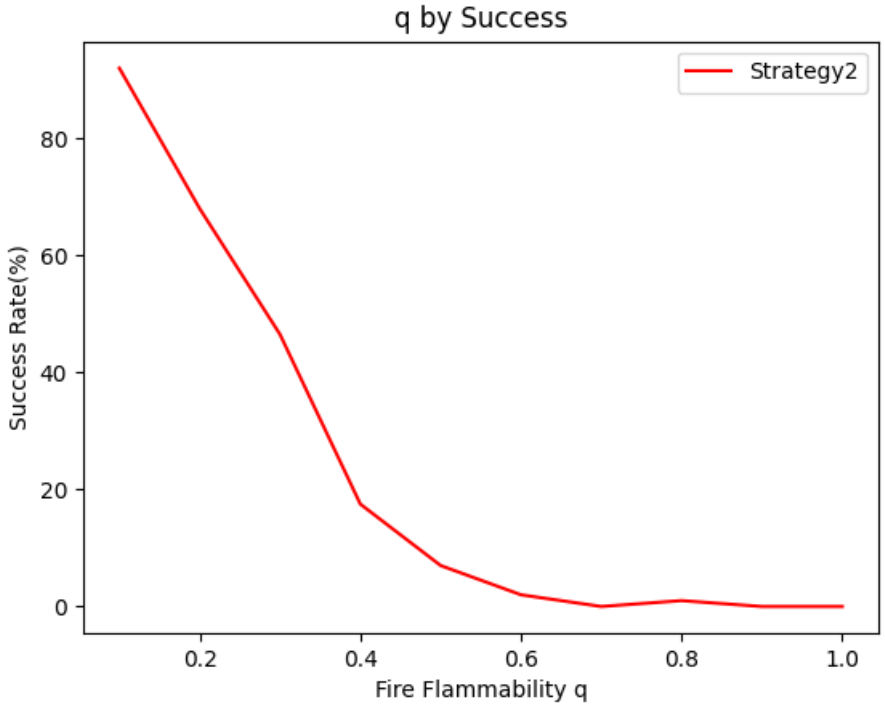

Figure 5: Strat 2 graph. Values:[92, 68, 46.5, 17.5, 7.0, 2.0, 0.0, 1.0, 0.0, 0.0]

Strat 3:

Dim:150, p:0.3, q:0.1 to 1 with 0.1 increase

Total of 20 unique mazes tested, running each maze 10 times. Each maze had to have a reachable goal, and the fire had to be in a spot where the robot could reach it, otherwise the maze would be thrown out and a new maze would be created.

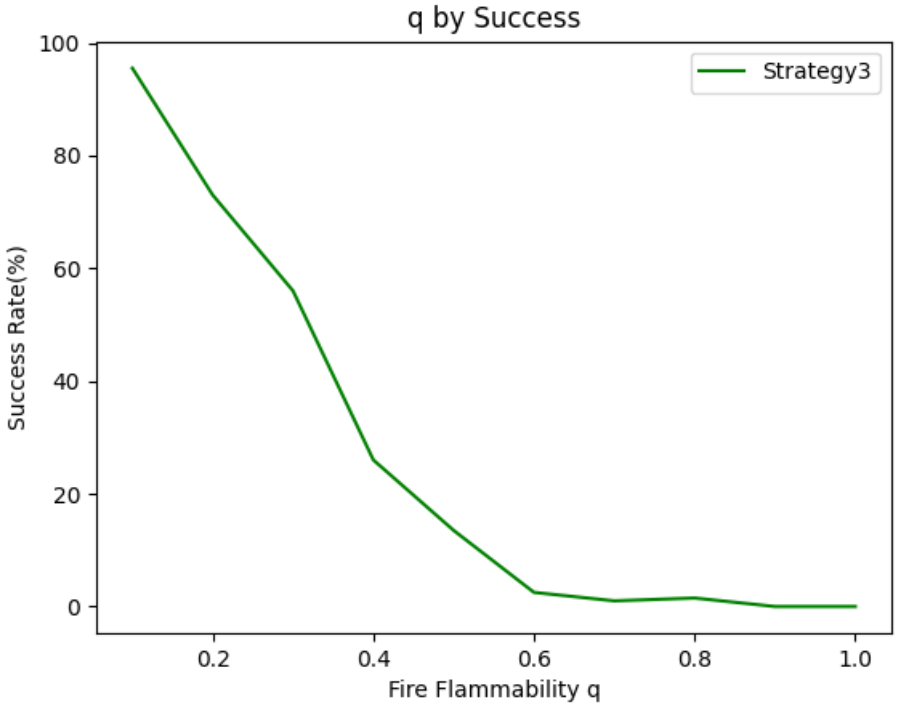

Figure 6: Strat 3 graph. Values:[95.5, 73, 56, 26, 13.5, 2.5, 1, 1.5, 0, 0]

Similarities:

when q reaches 0.6, it was difficult for all three strategies to complete the maze. The reason was either the robot was burned by the fire or the only way to the goal would be blocked by the fire.

Differences:

The time each strategy took was different. From shortest time to longest time: Strat1 < Strat2< Strat3. The safety between the three strategies are also different. From least safe to safest: Strat1 < Strat2 < Strat3.

Strat1 could run the fastest while Strat2 and Strat3 took more time to compute. When the dimension goes over 200, the time Strat2 and Strat3 took was well over a minutes. Strat3 would usually take 10 to 20 seconds longer than Strat2. Strat3 was the safest among the three strategies, Strat2 being the second safest, while Strat1 was the least safe, with the lowest success rate among the three strategies.

Why:

Strat1 only takes a single path and runs with it, the only computation that heads to the grids where the fire could spread to. If the fire does hit a grid in the robot's path and the robot has yet pass that grid, Strat1 would fail. This strategy is very high risk, and high reward (least time spent).

Strat2 would recompute a new path after each step, thus computing not only where the fire could spread to but also running a new path search for fire's each step forward. If the dimensions of the maze were large, then Strat2 would lead to thousands of new starting points, thus time consuming, but safer than Strat1.

Strat3 does the computations as Strat2 (recomputing at each step) while adding computations of an extra secondary array that holds 'danger levels' and runs a new search algorithm to find a path through the 'danger.' This strategy takes a bit more time than Strat2, but it is safer than Strat2 and yields a higher success rate.

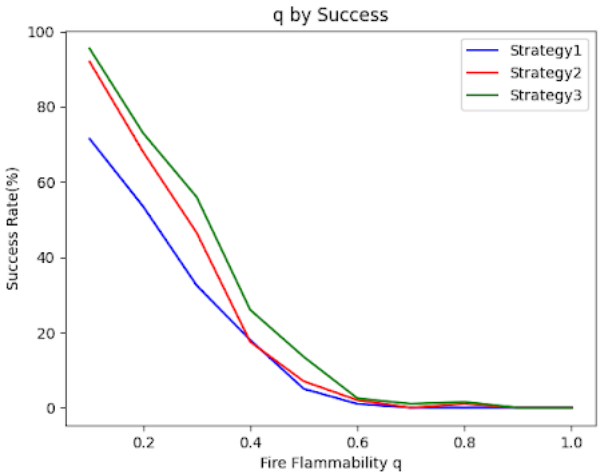

Figure 7: Comparison Between Strat1, Strat2, and Strat3.

Problem 7

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

Answer:

If we had unlimited computational resources at our disposal, we would improve and update Strat3 to simulate 'danger levels' at a more complex level rather than just simulating one step away. Originally, Strat3 takes the distance from each grid to the closest grid on fire and define the danger level as 'danger level' = 1 - ('distance to fire' / 'farthest distance possible') and run this for each step the fire takes in the maze.

The search algorithm would be mostly the same. The robot would still take the least dangerous grid and find the optimal path to reach the goal while walking all the way around the fire. Due to the nature of the strategy, any dimension that is over 40 would take Strat3 more than a minute to compute a path to the goal. So the newly updated Strat3 would not produce the most optimal path to the goal, but instead, it would give the safest possible path in every step the fire takes, leading to a higher success rate at each q.

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.

Answer:

Given the condition that we can only take 10 seconds between moves in the Fire Maze, we would change our strategy to find the nearest wall and use it as a barrier between the robot and the spreading fire. The computation for this strategy would also include choosing the side of the wall that is on the opposite side of the fire. Thus, because we only have 10 seconds to compute the path from S to G and the walls in the maze would not change their position when the run begins, the robot can produce a higher success rate by using the wall to protect itself from the fire and to force the fire to 'walk' around the wall instead of spread towards the robot (giving the robot time to escape from the spreading fire).