CS440 Project 1: Maze on fire

Mustafa Sadiq (ms3035) and Parth Patel (psp116)

February 2021

Introduction

Implementation and analysis of BFS, DFS and A* on mazes built on a 2-dimension array with a block probability of p. Added functionality and analysis when maze goes on fire with flammability q.

Libraries required: *random*, *numpy*, *deque* from *collections*, *heapq*, *sqrt* from *math*, *matplotlib.pyplot* and *time*.

Problem 1

- **Maze generation**

  To generate the skeleton of the maze, we used the package *numpy* and specifically the function *empty*. The function $make_maze(dim, p)$ handles the generation of mazes. The parameters dim and p represent the dimension and the obstacle density respectively. Using the *random* package, we generate a number between (0, 1) for each cell and if that randomly generated number is less than or equal to p, we declare the cell as blocked, otherwise, we declare it as open.

- **Representation**

  ASCII was used to represent the maze.

  'S' = Start (visualized)

  'G' = Goal (visualized)

  'X' = Blocked

  ' ' = Open

  '*' = Path (visualized)

  'F' = Fire

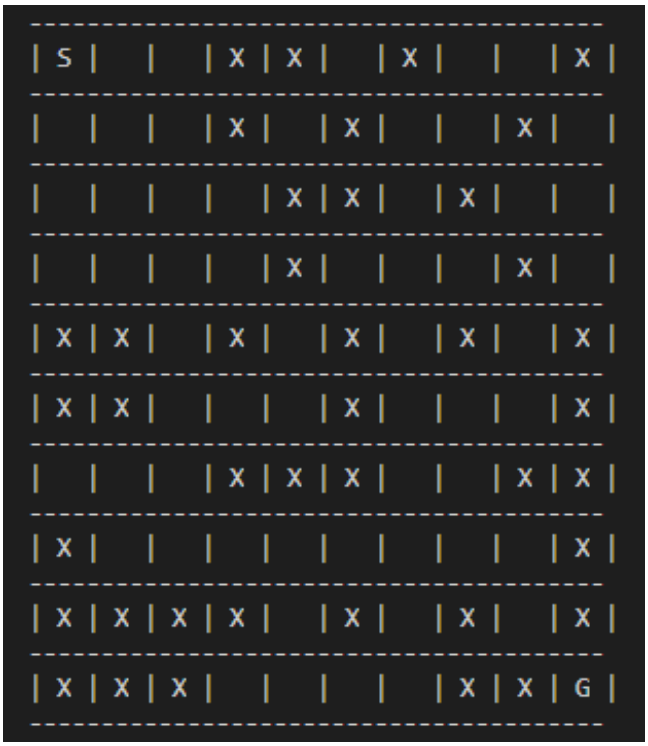

Figure 1: maze representation

Problem 2

- **DFS algorithm**

  The DFS algorithm we used was very similar to the outline provided to us during lecture. Our fringe was implemented using *stack* and to take a track of the parent of each child we used a *dictionary* which in essence represents a tree structure where: $tree[child] = parent$.

  The function $dfs(maze, start, goal)$ loops until there is nothing left in the fringe and the first element to be added in the fringe is *start*. Inside the loop, an element (call it current) is popped off the fringe, checked whether it is the *goal* we are looking for and if not then with the help of $get\_nonfire\_neighbours(maze, current)$ the children of current are determined. The way $get\_nonfire\_neighbours(maze, current)$ is that it looks up, down, left and right to the current and if open cells are found, it appends them to a list to be returned at the end of it. We utilize *tree* which is a dictionary to make sure that cells that have already been popped off before do not get added again, and *tree* also stores the parent of each cell.

  Implementation can be found in *maze.py*

- **Why is DFS a better choice than BFS here?**

  DFS is a better choice here since we are interested in finding if a path exists and DFS will find find a solution in less time than BFS which strives to find the shortest path. We can also see from Problem 4 that DFS can find a solution to a much higher dimension maze than BFS in under a minute.

  For the above variables set at:

```
dim = 100
density = numpy.linspace(0,1,100)
#(0, 1) with intervals of 0.01
runs per density = 100
```

  Implementation can be found in *maze.py* **We find the following results in Figure 1. plotted using** *pyplot*

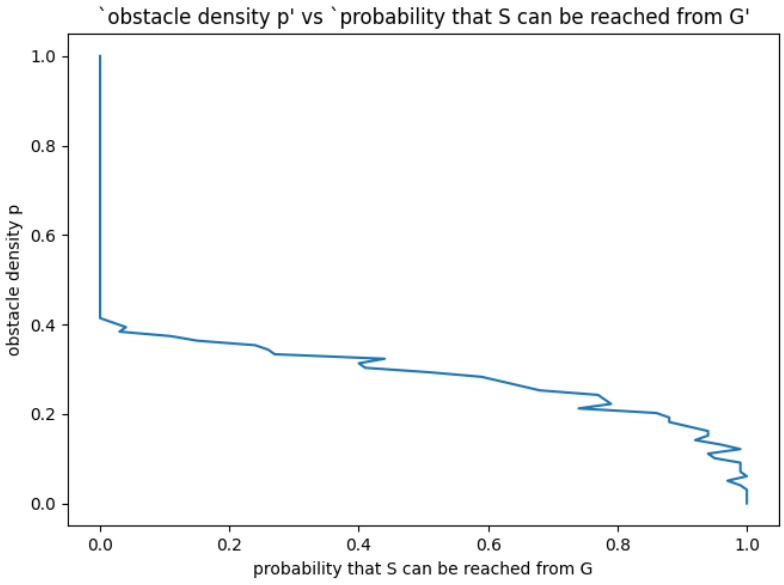

Figure 2: Problem 2

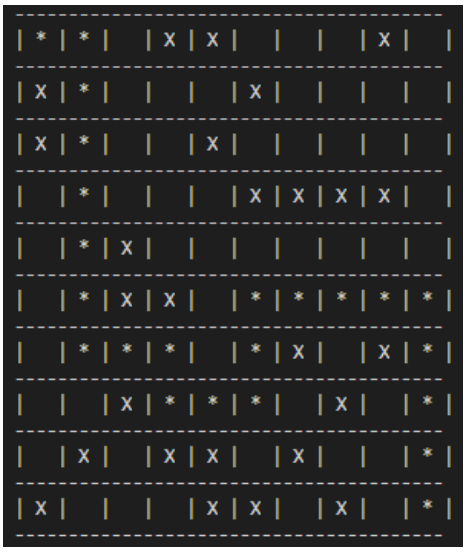

Figure 3: DFS

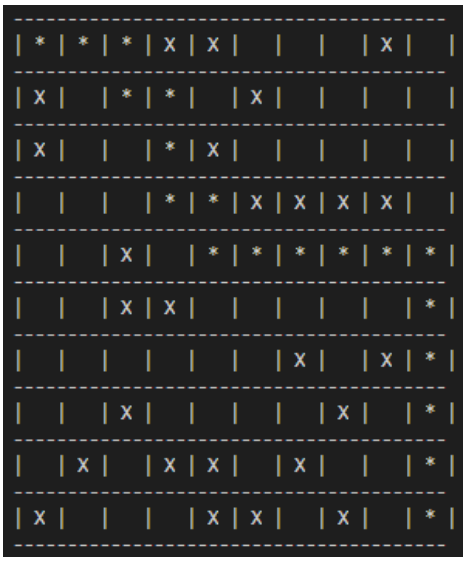

Figure 4: BFS and A*

Problem 3

- **BFS and A* implementation**

  The implementation for both of these algorithms is identical to the ones found in lecture. For BFS, we use *deque* from package *collections* which

is essentially an efficient implementation of queue structure.

Moreover, the only difference between the DFS and BFS algorithm is that the former uses a *stack* for the fringe while the latter uses a *queue* For A*, we use *heapq* which allows for an efficient implementation of a priority queue. The priority of a cell is determined by the addition of the euclidian distance from the cell to the goal and the level of that cell. We store the path the same way we did for DFS.

- **If there is no path from S to G, what should this difference be?**

  the difference should be zero since both the algorithms are exhaustive which means they will traverse through every node available.

  For the following variables set at:

```
dim = 100
density = numpy.linspace(0,1,100)
#(0, 1) with intervals of 0.01
runs per density = 100
```

  We find the following results in Figure 2. plotted using *pyplot* **Implementation can be found in** *maze.py*

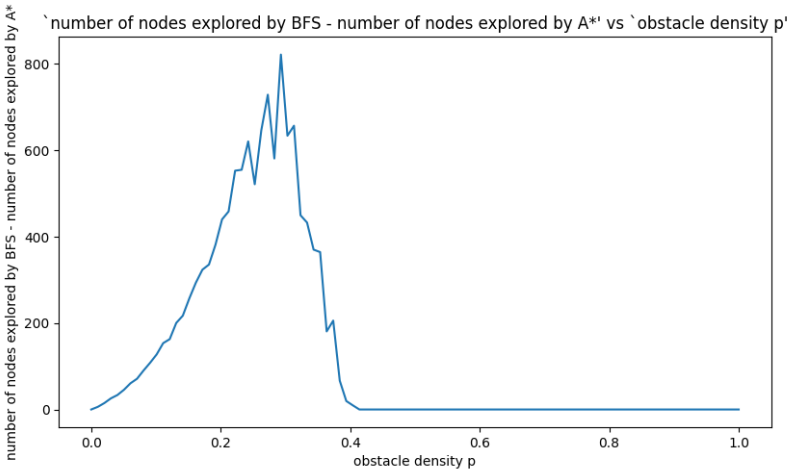

Figure 5: Problem 3

Problem 4

Using the following algorithm:

```
for 5 runs:
```

```
while time < 60:
    find largest dimension for a solved maze
```

The following largest possible dimensions were found to be solved at $p = 0.3$ under 60 seconds:

$$DFS = 10,700$$
$$BFS = 3,000$$
$$A* = 2,600$$

Implementation can be found in *maze.py*

Dynamic Fire Maze (Strategy One and Two)

- **Strategy One**

  The strategy one algorithm has been developed just like the way it has been described. The strategy kicks off by randomly generating coordinates of an open cell to set fire using *randrange* and then using our bfs algorithm, we find the shortest path from start to goal. We let the agent move and the fire spread alternatively. The fire spreads using $advance\_fire\_one\_step(maze, q)$ where the probability that a cell will catch on fire is equal to $1 - (1 - q)^k$ where k = number of neighbors that are on fire.

```
dim = 100
p = 0.3
q = numpy.linspace(0,1,100)
runs per density = 100
```

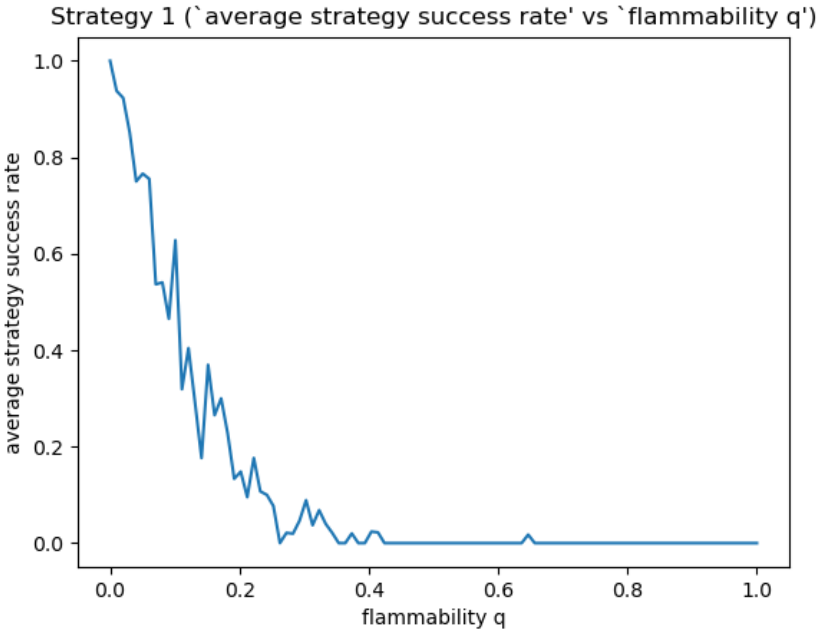

Figure 6: Strategy one

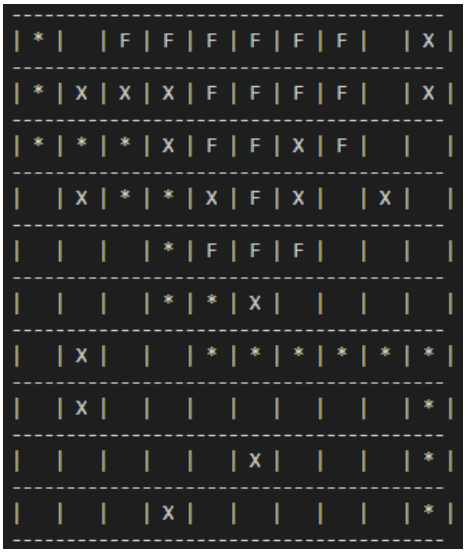

Figure 7: Fire maze traversal example

- **Strategy Two**

  This strategy is similar to strategy one but instead of computing the path just once, we compute a new path every time the path moves. Theoretically and practically, strategy two does better than strategy one.

```
dim = 100
p = 0.3
q = numpy.linspace(0,1,100)
runs per density = 100
```

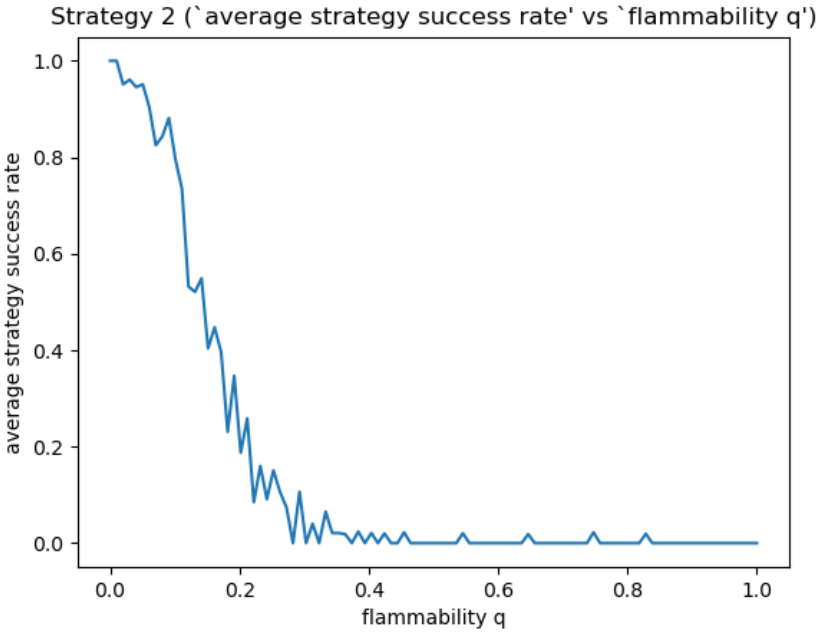

Figure 8: Strategy two

Problem 5

- **Strategy three**

  our initial philosophy was to come up with a heuristic that would help us traverse the maze in k step intervals. The first heuristic we came up with was as follows:

  $euclid\_distance(maze, current, goal) - find\_nearest\_fire(maze, current)$.

  We implemented the function $find\_nearest\_fire$ by utilizing bfs and this heuristic was processed using a_star. We typically used a path for $dim/4$

steps before discarding. The idea that drives this approach is that the closer fire is to a cell, the higher the probability that it will get flamed in the future.

Moreover, by adding the euclid distance from the cell to the goal, we are trying to find a balance between reaching the goal and evading the fire. This implementation was able to edge out startegy 2 for the most part. We called this algorithm: CALM (CALculated Maze). This is how it perfomed:

```
dim = 30
p = 0.3
q = numpy.linspace(0,1,100)
runs per density = 50
```

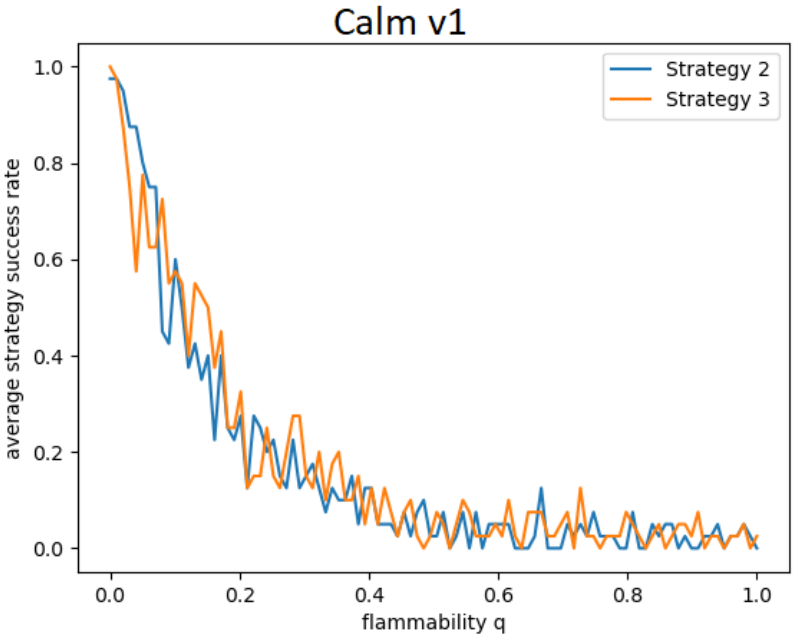

Figure 9: CALM comparision

The second implementation of CALM was devised by emphasizing on accounting for future states. We changed the heuristic to this:

$euclid\_distance(maze, current, goal) - find\_nearest\_fire(maze, current) - find\_nearest\_fire(future\_maze, current)$.

For this, we passed a k steps future maze (which essentially was a simulation of how the fire would look in k steps) to a_star to use in the modified

heuristic. The idea behind this implementation was to make cells that are either on fire or closer to fire in k steps even more unlikely to be chosen during path discovery. We observed a marginal improvement in CALM:

```
dim = 30
p = 0.3
q = numpy.linspace(0,1,100)
runs per density = 50
```

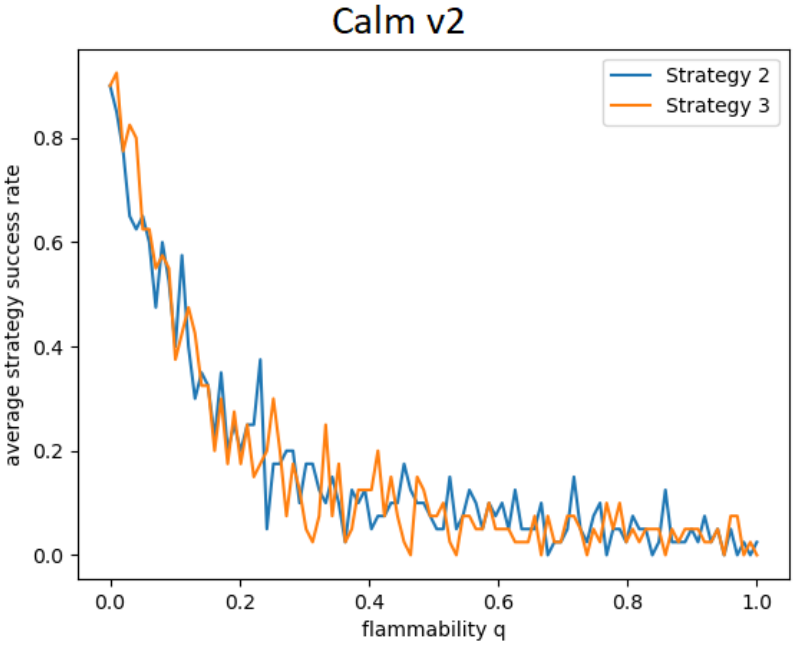

Figure 10: CALM v2 comparision

The one that performed well we are calling SKIM (SKip Maze) as following:

```
X = distance threshold (take one step at a time)
Y = how many steps to take when distance > X
path = bfs(maze, start, goal)
for step in path:
    if distance from nearest fire < X:
        take one step and calculate path again
    else:
        take path[:Y+1] steps and calculate path again
```

To find out the distance to the nearest fire we devised the an algorithm similar to BFS, looking for any neighbour which is 'F' and return it:

```
def find_nearest_fire(maze, start):
    returns nearest fire position to 'start'
```

This strategy is similar to Strategy 2 but instead of calculating a new path every time we find a new path K steps after. (e.g. every 7 steps for dim = 50 unless we are 14 away from fire). We can say it is a sweet spot between Strategy 1 and 2 where it is neither to liberal nor to conservative. To find out the performance compared to Strategy 2 we made 10 runs of dim=25 mazes where X=7 and Y=4 and found the following results (mazes with no path to goal and no path to initial fire were discarded):

```
dim = 30
p = 0.3
q = numpy.linspace(0,1,100)
runs per density = 50
```

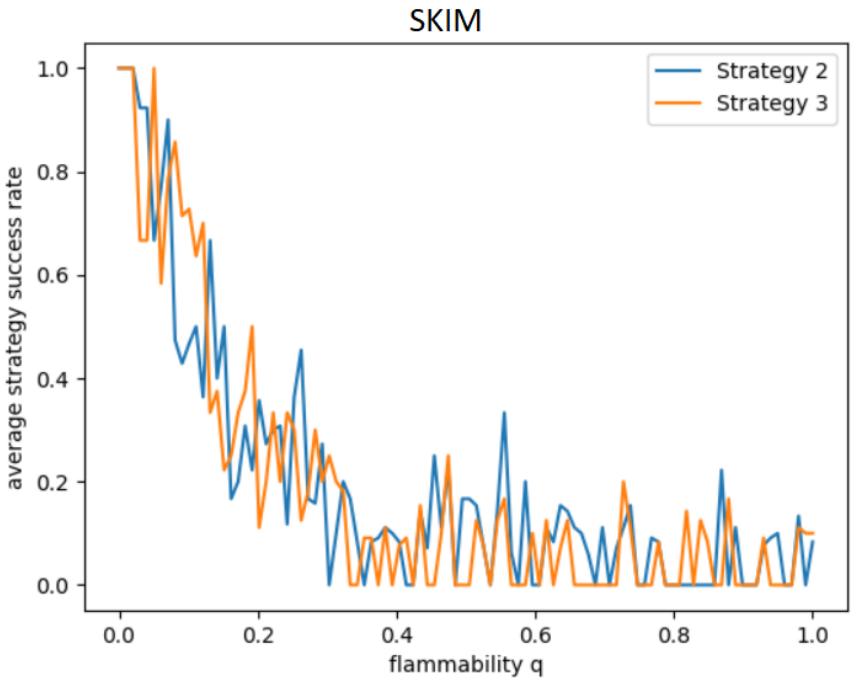

Figure 11: Skim

Problem 6

- **Strategies compared**

  For the following variables set at:

```
dim = 25
p = 0.3
q = numpy.linspace(0,1,100)
runs per density = 20
away from fire = 7
steps = 4
Implementation can be found in $maze.py$
```

We find the following results plotted using *pyplot*

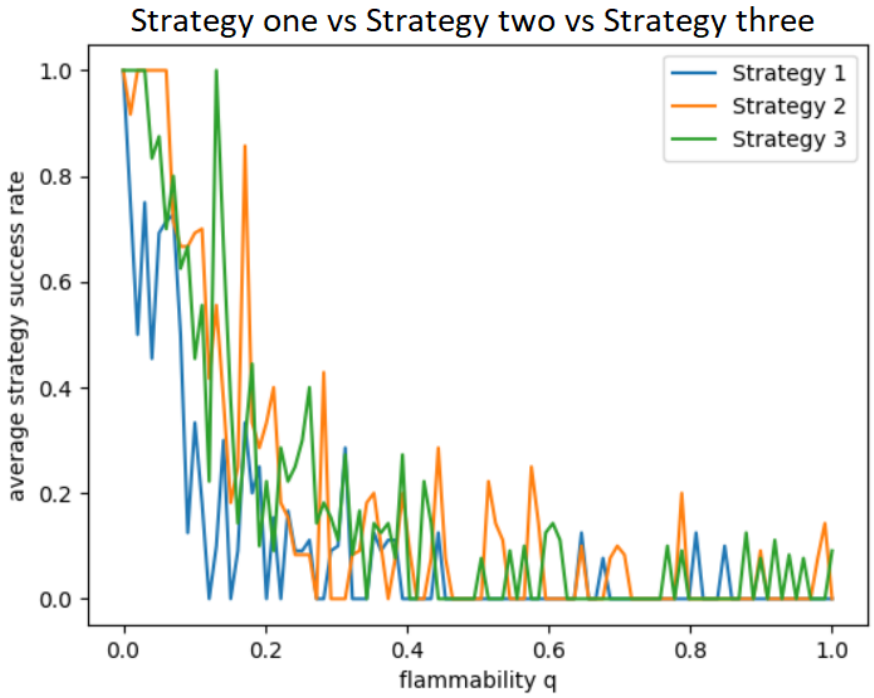

Figure 12: Strategy 1 vs Strategy 2 vs Strategy 3

- **Where do the strategies perform the same? Where do the perform differently? Why?**

  Although it is not as visible in this graph, the strategies perform the same at the start and the end of the graph. They perform same around the start because the q is low which means the fire does not spread that easily and there is a higher chance of success; where else for the end, it behaves similarly because it is typically harder to escape the fire and that comes with a lower chance of success. The part where it really shows how each strategy performed is in the middle where exists a good balance of q and chance of success

Problem 7

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

  Having unlimited computational resources would allow us to run an $infinite$ amount of simulations on how the fire would progress from a given cell and using statistical methods such as sampling, our algorithm will have a precise image of the fire in given k steps. Subsequently, we can use this image to further improve our heuristic from CALMv2, particularly, the part which determines the closest fire cell for a given cell in k-step future maze.

  Alternatively, using a computationally heavy approach, we can calculate path success from every neighbour. The neighbour with the most success will be the one we end up going on. We devise the following algorithm:

```
at current step:
    for every neighbour:
        run 100 paths from this neighbour and find success
    use the most successful neighbour as next step
```

  This way at every step we will pick the neighbour as next step in the path which has the most expected success.

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 a modified strategy 3 where instead of using a_star, we could use a heuristically driven dfs where the order in which the children are added to the stack depend on the heuristic. The heuristic will remain the same i.e. $distance\_from\_goal - distance\_from\_closest\_fire\_cell$. This idea has been derived from the philosophy of local search where at each point, we are chasing a local maximum/minimum.

Addendum

DFS, BFS algorithm and graphing was done by Parth Patel
A*, Strategy one algorithm and graphing was done by Mustafa Sadiq
Strategy two and Strategy three implementations were done by both group members
I have read and abided by the rules laid out in the assignment prompt, I have not used anyone else's work for my project, my work is only mine and my group's.
Signed by: Mustafa Sadiq and Parth Patel