Assignment I

Intro to Artificial Intelligence

CS 16:198:520

Kaustubh N Jadhav
knj25

Pranav Shivkumar
ps1029

Sanyam Jain
sj770

I. The Maze is on Fire

Introduction to the Problem

The basic idea of this problem is that an agent wants to move from one end of a burning maze to the destination. The maze is a square grid of cells where each cell is either open or blocked and the agent can navigate through the maze using the open cells. To solve this problem, we used Python as the primary programming language.

The first step in tackling this problem was to create the maze. To accomplish this, we first generated a 2D array in Python. The basic idea was to randomly assign each index of the array with a value of 0 or 1, corresponding to whether the cell in the maze is open or blocked and converted the array into a graph by using a dictionary. The dictionary was initialized by assigning the node coordinates in the maze as the keys and the values for these keys were a list of the neighbors of that node. The resulting figure generated is the maze, shown in the four figures below:

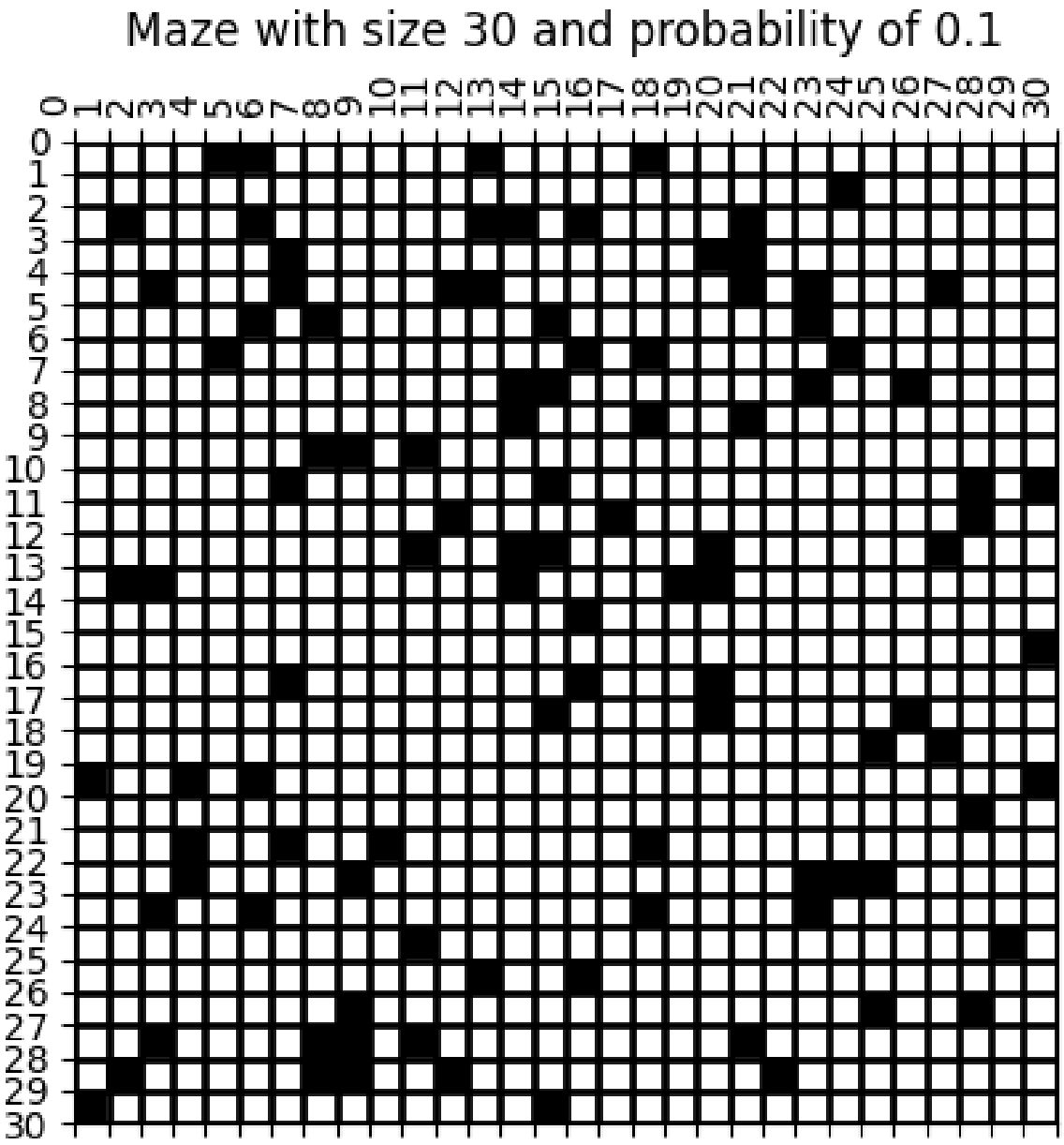


Figure 1: 30x30 Maze with Probability 0.1

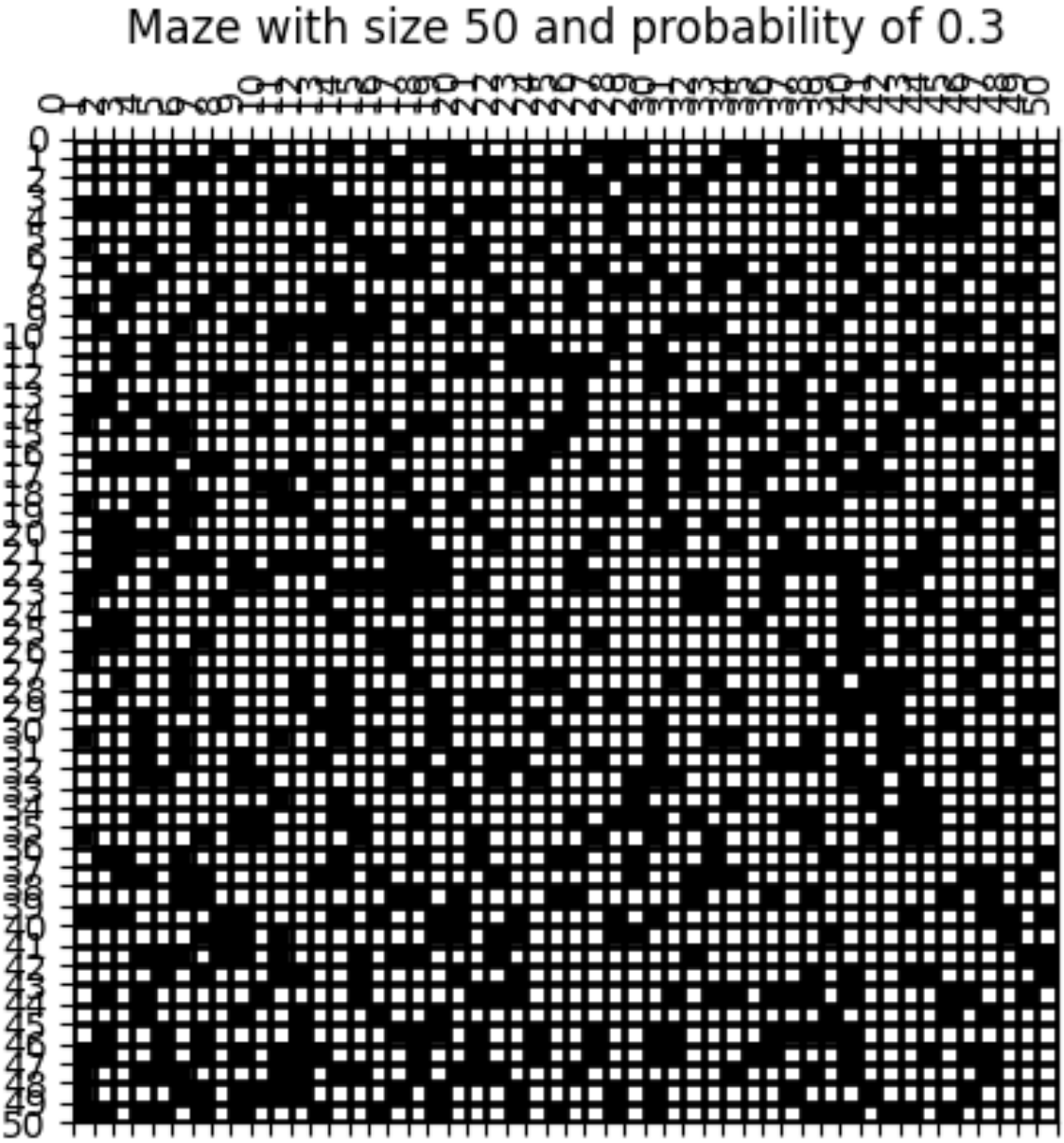


Figure 2: 50x50 Maze with Probability 0.3

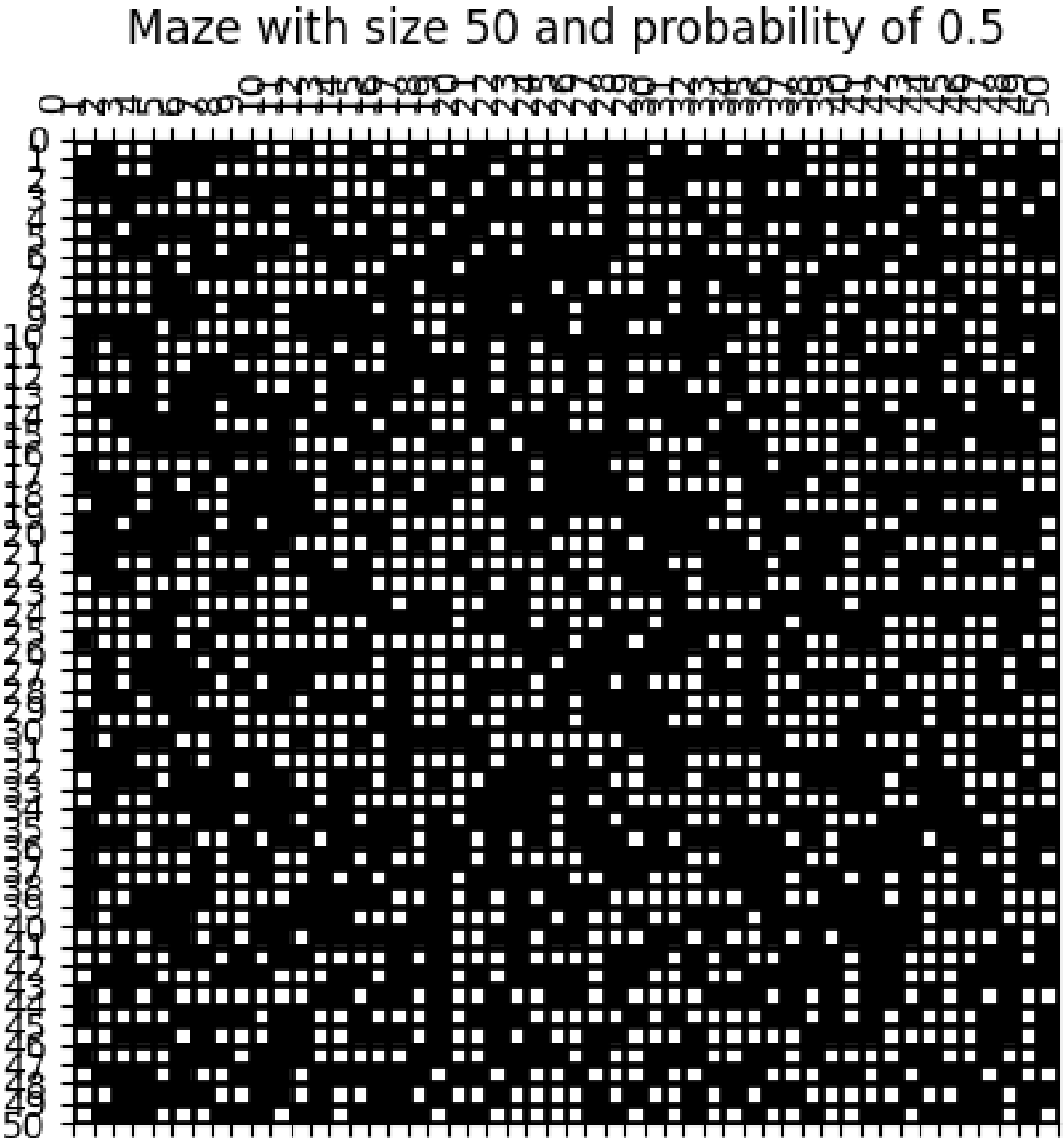


Figure 3: 50x50 Maze with Probability 0.5

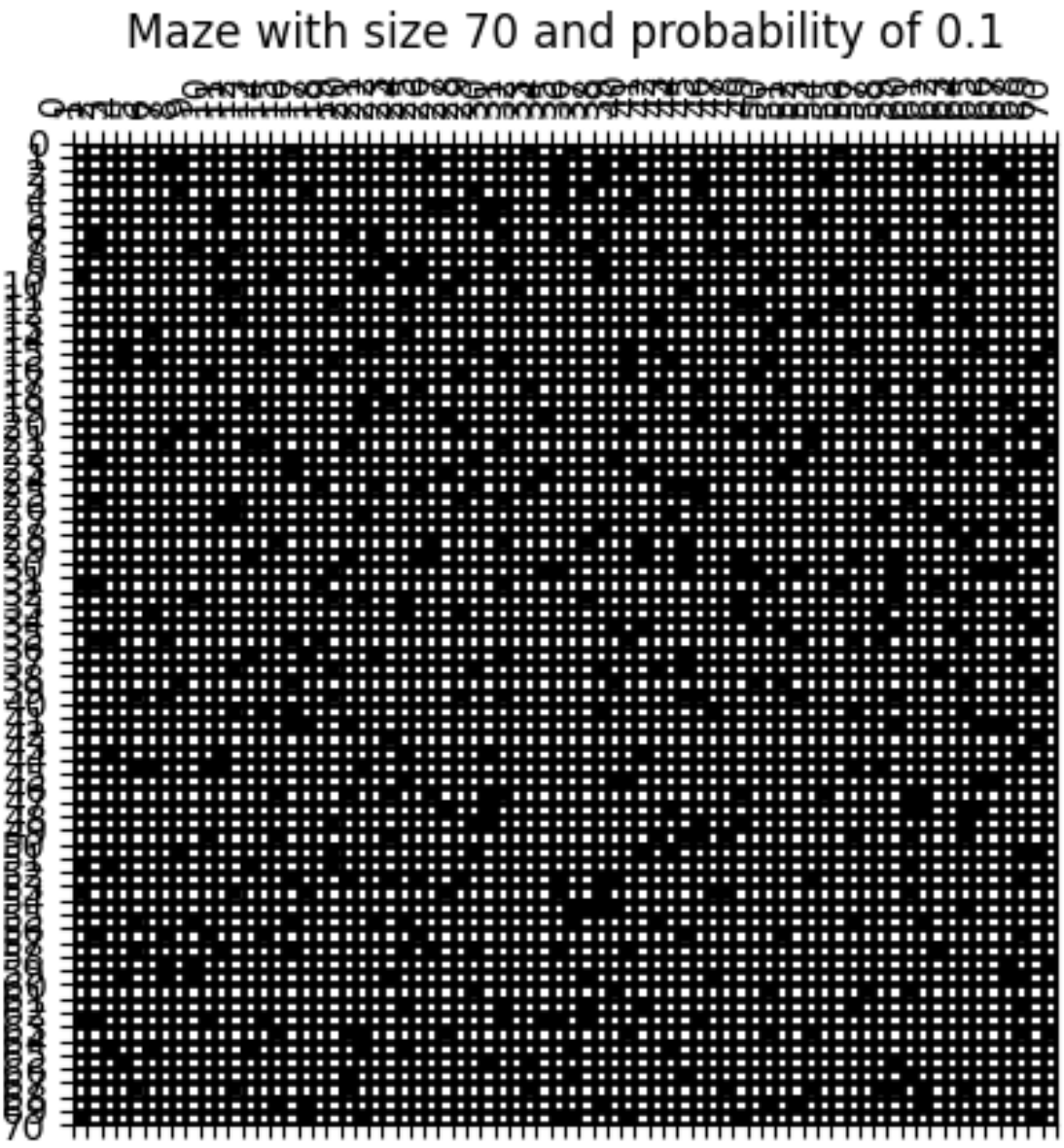


Figure 4: 70x70 Maze with Probability 0.1

Implementation of Searching Algorithms

In order for the agent to reach the goal/destination, there has to be a path that he could follow. For this, we had to code five algorithms to determine the path for the agent to follow. Those are:

1. Depth First Search (DFS)
2. Breadth First Search (BFS)
3. Iterative Deepening Depth First Search (IDDFS)
4. Dijkstra's Algorithm
5. Bidirectional Depth First Search (BDDFS)

Based on the results and the time taken by each, we determined the best algorithm to use.

- DFS
  In our implementation, we used a stack to keep track of the nodes visited. We run a loop until the stack is empty and pop out/remove a node from the stack in Last In First Out (LIFO) order. We then get the neighbors for this node and iterate through them, checking if the nodes have been visited or not. If they haven't, we add these neighbors to the stack and continue. If the neighbor is the destination, we return the agent's path back to the main function. If there is no path, we return "F" back otherwise, we return "S".
- BFS
  In our implementation of BFS, the working principle is the same as DFS; the only difference is that the data structure used to store the nodes is a queue instead of a stack. This means that the nodes are removed from the queue in First In First Out (FIFO) order rather than LIFO order. If there is no path, we return "F" back to the function, and if there is a path, we return "S".
- IDDFS
  For our approach, we start at the root node of the graph and explore as far as possible along each path by marking the node as visited and moving to the adjacent node, before backtracking. Once we backtrack, we check for other unmarked nodes and once visited, mark them and continue the process.
- Dijkstra's Algorithm
  The idea behind our implementation of Dijkstra's algorithm is that we mark each node as having a distance infinity from the source node. We traverse along the adjacent neighbors and update the distances from the source node, with the help of a set (visited), to keep track of the visited nodes.
- Bidirectional BFS
  The idea behind bidirectional BFS is that we implemented BFS from the source in the forward direction and from the destination in the backward direction. To accomplish this, we used two queues, the front queue and the back queue. We try to alternate between these queues; in each iteration, we choose the smaller queue for the next iteration which effectively helps in alternating between the two queues only when the swapping between the two queues is profitable. We incrementally approach each other until an intersection point is reached. Once this happens, we return the path back to the main function. If there is no path, we return "F" back to the function, and if there is a path, we return "S".

Selecting the Size of the Maze and Search Algorithm

The next step to tackle the problem was to determine the appropriate size of the maze. To find this, we created mazes of sizes 20 to 100 for different probabilities.

We solved the mazes using the search algorithms discussed above (except ID-DFS as we were getting repeated warnings for high number of recursion for large mazes), and plotted graphs for the number of successes (how many times the agent is able to reach the destination), time taken in microseconds to solve the mazes as well as the length of path followed by the algorithms to reach the goal. Based on theses factors the optimal size (that is large enough but solvable multiple times) for the maze is decided. These graphs are shown below.

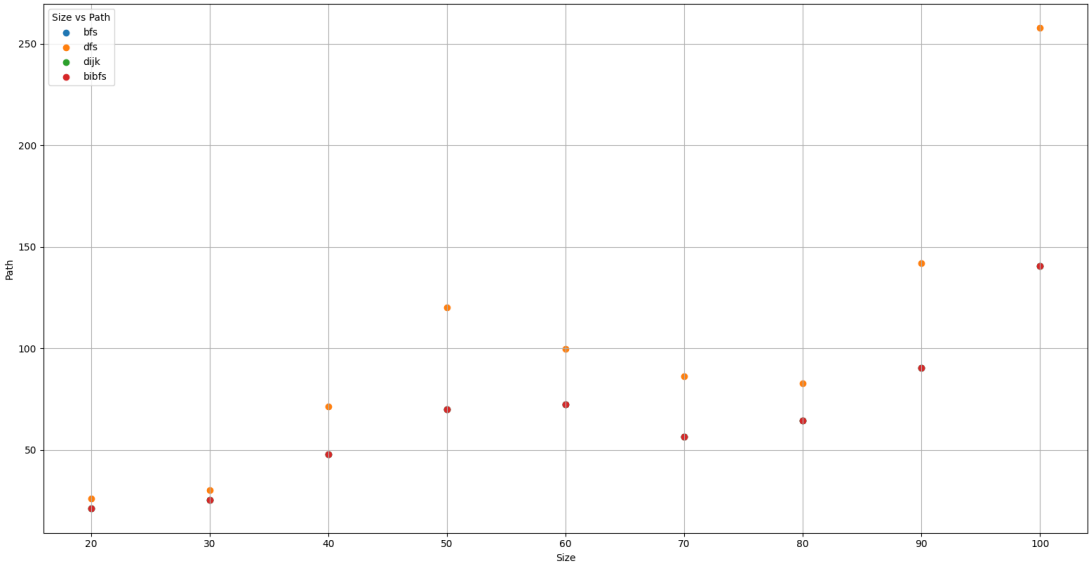


Figure 5: Size vs Path

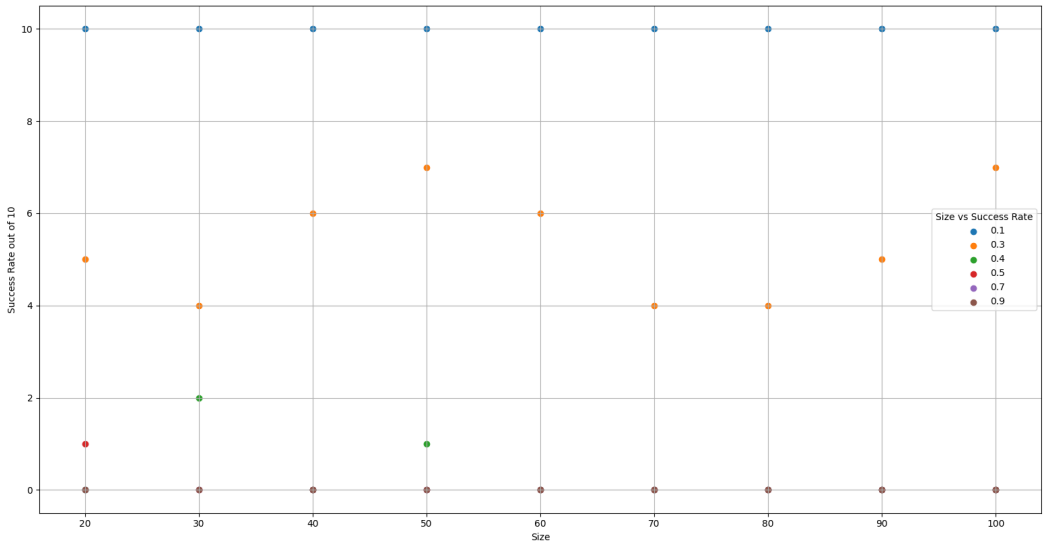


Figure 6: Size vs Success Count

Based on these graphs, we found that a maze of size 70 was the optimal size - it wasn't too difficult to solve, nor was it too easy to solve. The sample maze is shown in the figure.Based on the number of success counts, time taken and path length by each algorithm for given probability (0.3), we decided to go with bidirectional BFS algorithm for solving maze on the fire.

Note - During our analysis we found that probability given to us 0.3, is optimal as for probabilities above that maze is too hard to solve and probabilities below that maze is too easy.

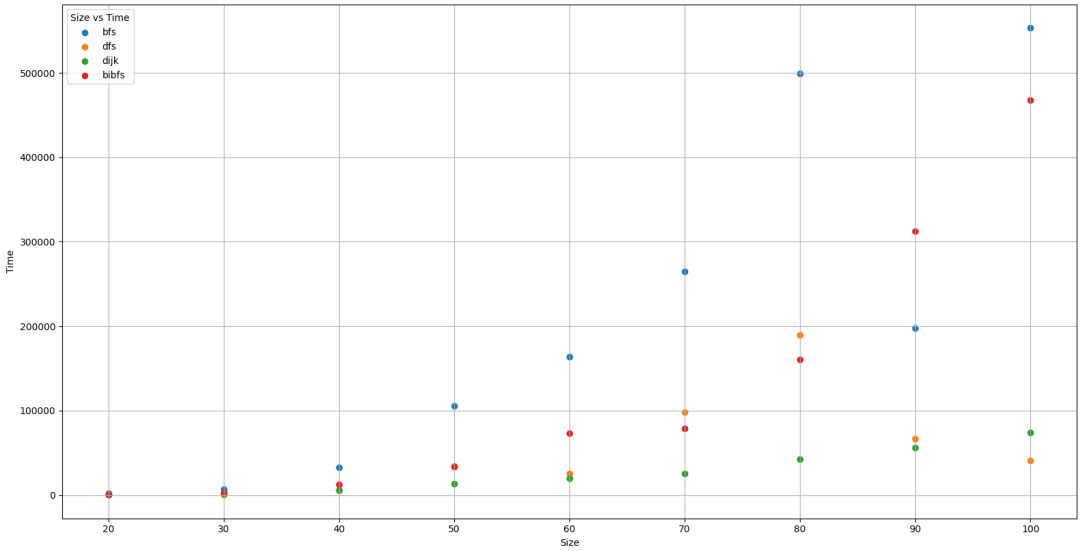


Figure 7: Size vs Time

Fire in the Hole

The next stage of emulating this problem was to burn the maze down. For this, we first randomly chose a node of the graph to be on fire, using the function *lettherebefire()*. This function Iterates through graph keys and generates a random number using numpy's random function. If the random number index the nodes on the graph, thus if node provided by this random number is not start or end and there exists a path between this node and start point, this node is selected as first node on fire.From this node, we simulate the fire spreading with the help of the function *spreadfire()*, which takes the node on fire as the argument. We iterate through the nodes of the graph. Provided the node is not the source or destination, we initialize a counter (onfire) to count the number of neighbour nodes on fire any node. If any of the neighbors are on fire, we increment the counter. If the counter is greater than 0, we use probabilistic logic $(1-((1-q)^k))$ to set that on the fire is on fire. Once this is done all nodes that caught the fire are added to the list.

This procedure is done for each step agent takes and list of node on fire is used to detect if user burned or not.

Strategies to Solve the Problem

The baseline strategies provided to us are described below:

- Strategy 1 (*Foolhardy*): In this strategy, we first determine the shortest path the agent to follow starting from the source to the destination, irrespective of where the fire is. This would result in either the agent dying due to the fire, or he successfully reaches the destination.
- Strategy 2 (*Intelligent but Cheat*): In this strategy agent recomputes the shortest path from his current position to the destination every time, thus, the agent changes path based on the fire. While this would help the agent reach his destination with a higher chance of surviving, there is still a chance of him dying either by getting trapped in the maze on fire with no path available or getting burn as even new step catches the fire. This method seems cheating as agent can only feel the fire it is closer to it note right from the starting.

  We were also asked to come up with our own strategy, which is described below:

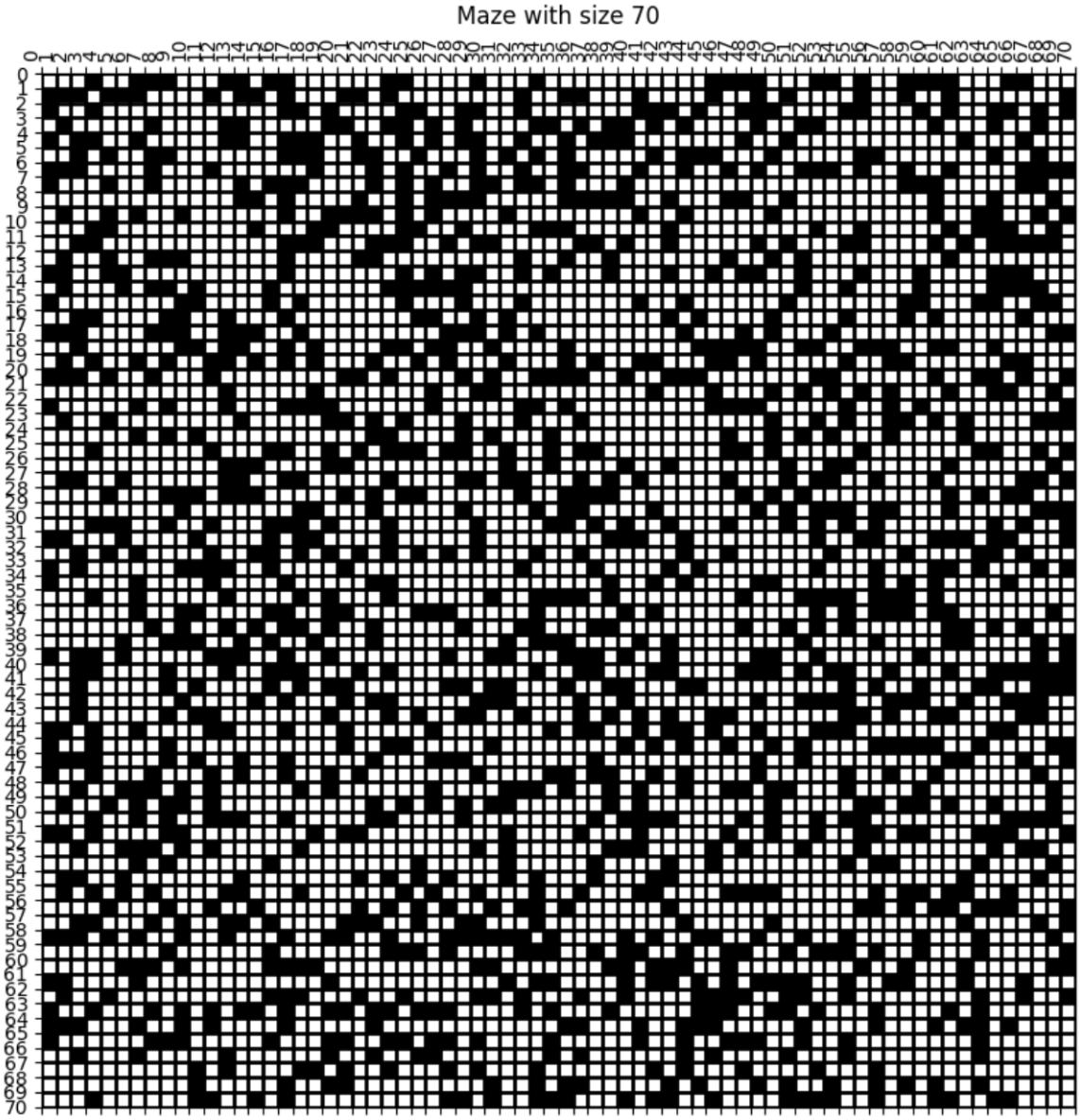
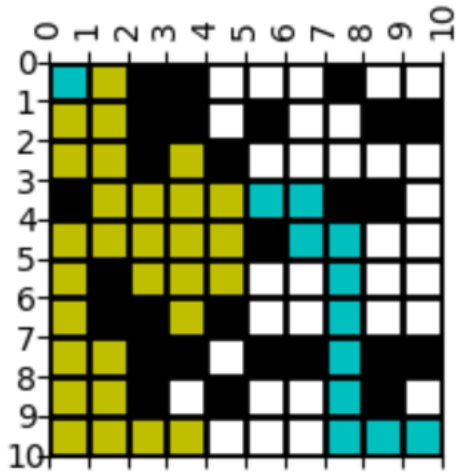


Figure 8: Sample Maze - Size 70

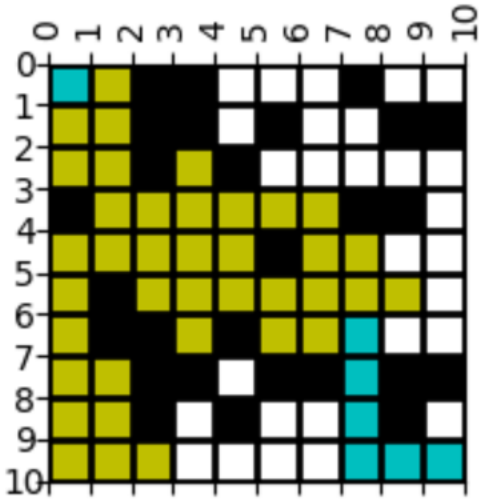
- Strategy 3 (*Realistically Intelligent*): In this strategy, we decided to compute the shortest path from the source to the destination and check if the node on path or neighbors(depending on the decided maxDepth) of that node are burning. If they are, we recompute path from the previous step to the goal. Since there are blockages and we have little room to move it make happen that agent may have to take same step as given by the path calculated initially. In this method also agent can die either by burning or by being trapped

Sample solutions are shown on next page for different sizes and flammability.

Size :: 10 , Flamability:: 0.3 , Solution:: SOLUTION 1



Size :: 10 , Flamability:: 0.3 , Solution:: SOLUTION 2



Size :: 10 , Flamability:: 0.3 , Solution:: SOLUTION 3

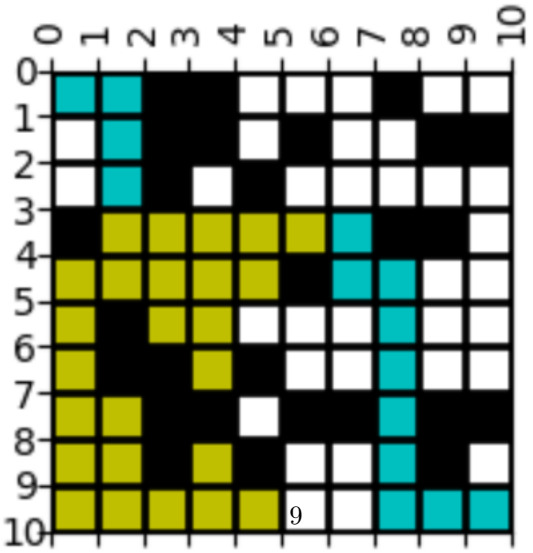
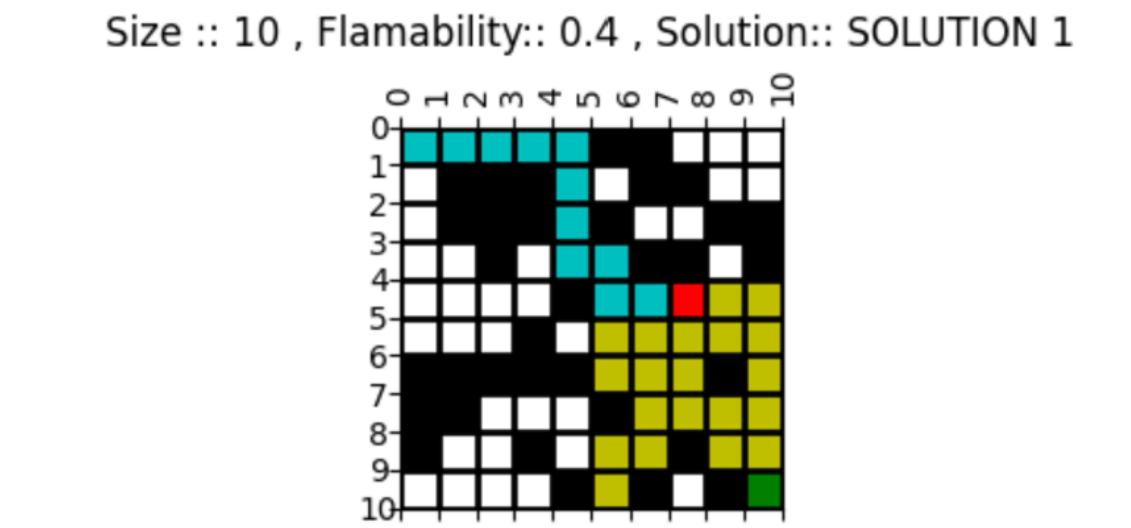
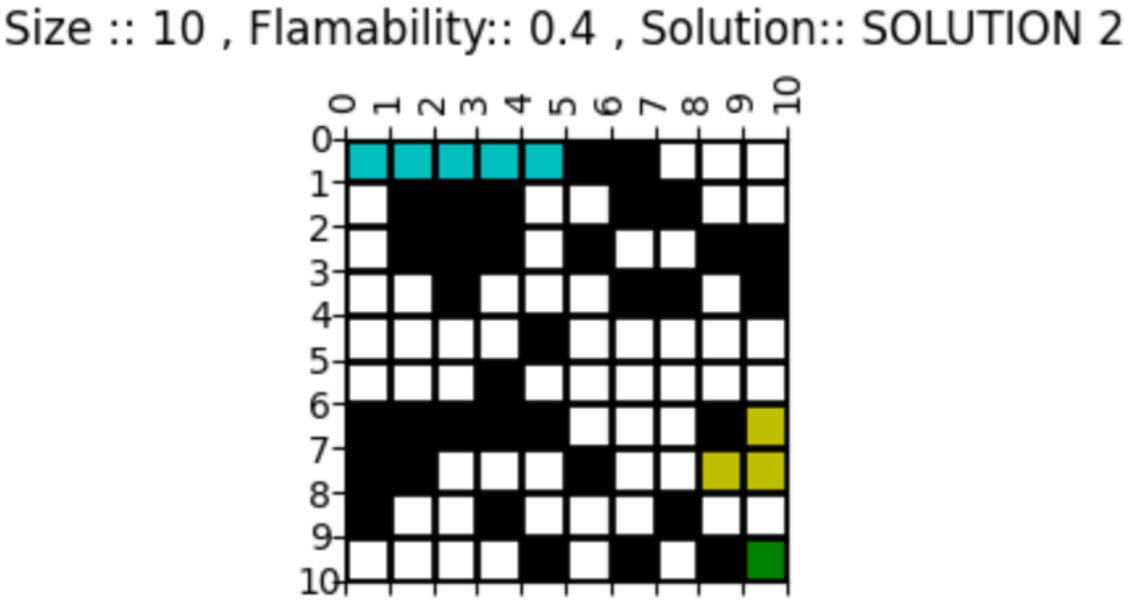


Figure 9: Success





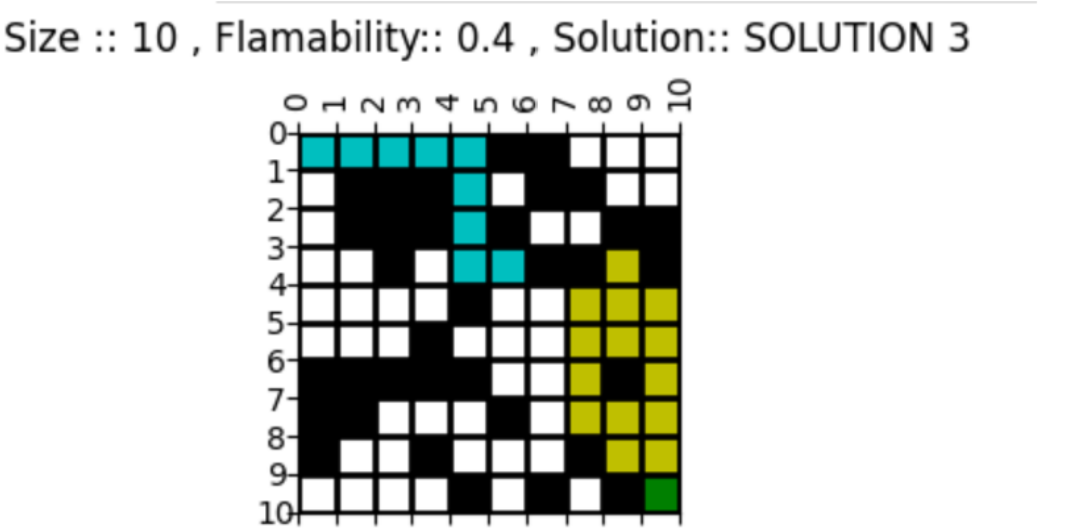


Figure 10: Burned or Trapped

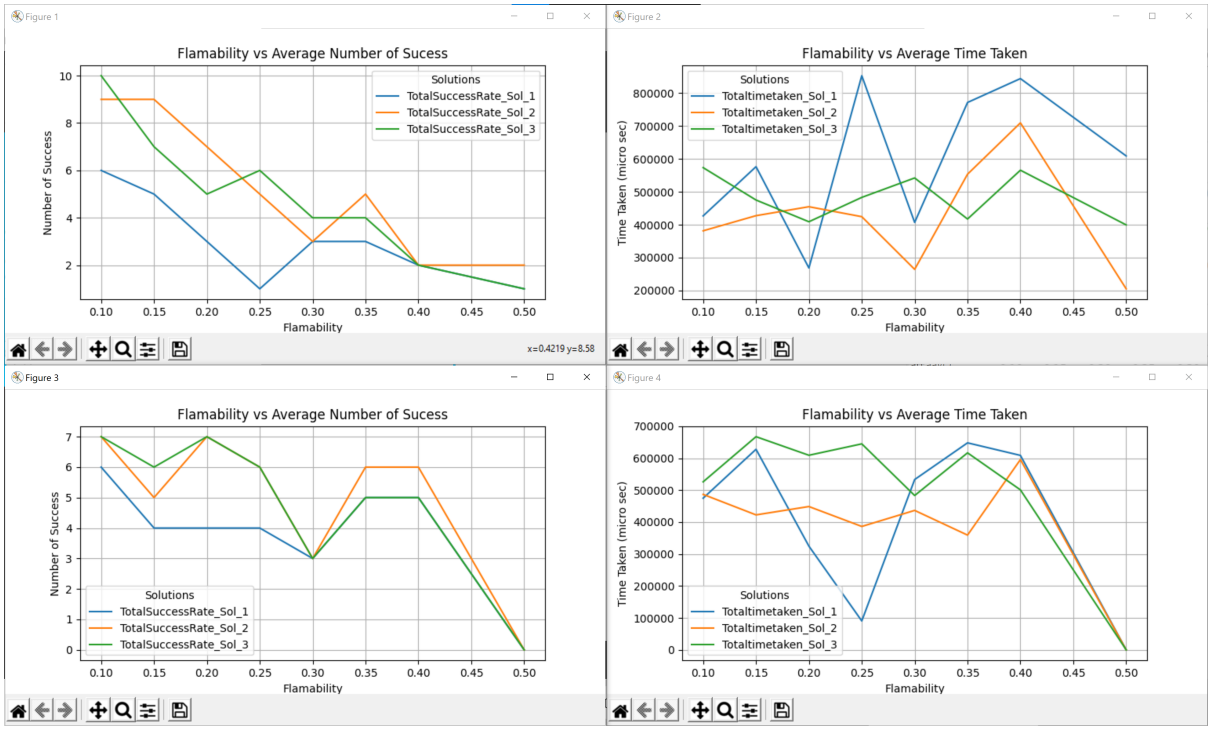


Figure 11: Maze Size 70

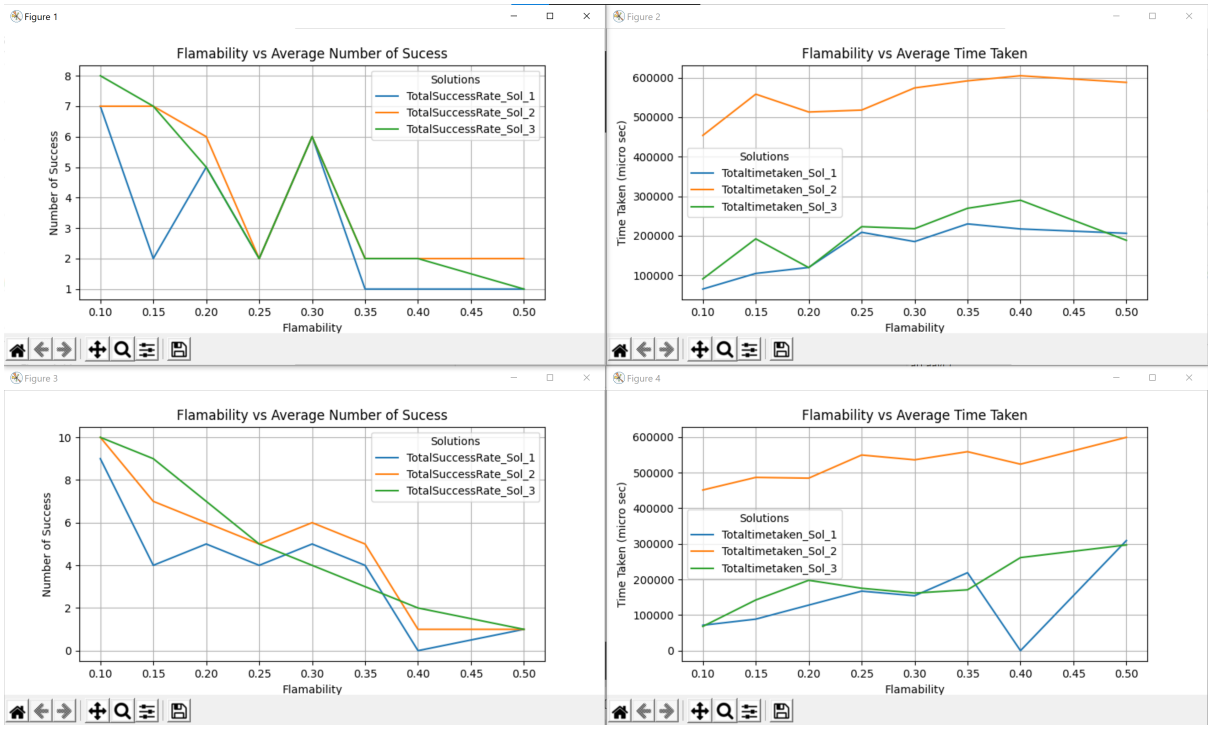


Figure 12: Maze Size 50

Conclusion

Figures 11, 12 and 13 are graphs for strategies 1, 2 and 3 for maze sizes 70, 50 and 30 respectively. Initially we planned to run only solutions for a maze of size 1x1 but we were not the getting expected result i.e. time taken by strategy 3 should be less than taken by strategy 2 so we ran solutions for the other two sizes also. We also ran two iterations for setup which include 10 iterations of maze solving for each solution. All the solution strategies are given the same playing ground i.e same fire starting point, same maze the only difference was in fire spreading which is driven by probabilistic logic.

For Size '70' as expected strategy 1 was the worst performance, Strategy 3 was on par with strategy 2. But strategy 2 was the clear winner as it took lesser time than other two strategy.
For Size '50', as expected again, Strategy 1 was the worst performance. Strategy 3 performed on par with strategy 2, if not better. But strategy 3 was the clear winner as it takes less time than other two strategy.

We can conclude that for large mazes, strategy 2 will be preferred and for smaller maze,s strategy 3 will be preferred.

Strategy 3 took more time for large mazes as it was first checking If the neighbouring nodes are on fire using depth limited DFS algorithm and if any node found to be on fire, it recomputes the path. For large mazes, it may be the case that the algorithm had to look deeper (max depth) before it encounters node of fire and then it recomputes the path too. These both action if performed together must be increasing the time taken.

II. Maze Thinning

Introduction to the Problem

For this part of the project, the maze was no longer on fire. Instead, we utilize the A* algorithm with heuristic like the Manhattan distance, Euclidean distance, solving thinned maze and relaxed movement.

For thinned maze heuristic, we solved a thin maze to decide priorities for nodes by taking start point as concerned node and end point as goal.

For relaxed movement heuristic, we added diagonal movement for agent and the path length from a node to destination was used as heuristic.

Procedure

First we implemented thinning logic which thins the maze i.e remove the given fraction of obstacles from the maze. Figure 13 shows the effect of thinning.
To figure out the best size of the maze for the problem, we solved mazes of size 30 to 70 with blocking probability of p = 0.3 using the A*-Manhattan algorithm for 100 iterations. Therefore, we concluded that we can reasonably solve a maze of size '50' repetitively in a reasonable amount of time.

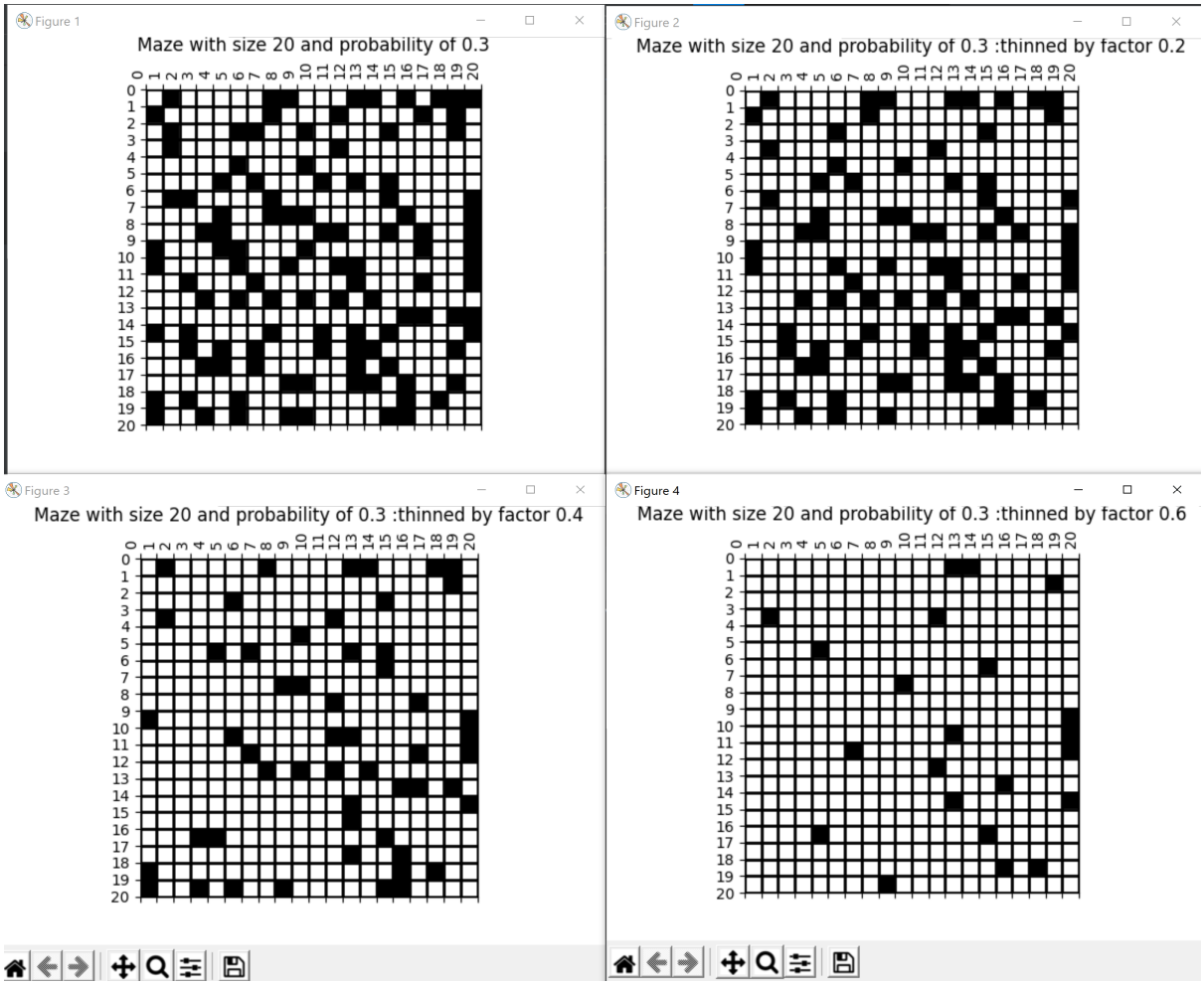


Figure 13: Effect of Thinning the Maze of size 50x50 and p=0.3

The first step in this problem was to create an A*-Manhattan solving algorithm. The basic idea behind this algorithm is that it picks a node from the graph based on a value f(n), equal to the sum of two parameters g(n) and h(n), where g(n) is the cost of moving from the source to the current node (n) following the path generated and h(n) is the priority decided on the basis of Manhattan distance from the current node to the destination for A* Manhattan and Euclidean distance from the current node to the destination for A* Euclidean. The algorithm chooses the node with the lowest value of f(n) and move forward to find path to destination from selected node.

For A* Thinning we solved the thinned maze with for nodes with goal as destination to decide priority for heuristic. Thus for each node a thinned maze is solved and path length is used as priority.
We constructed one more heuristic strategy which involves allowing diagonal movement for the user to calculate priority. Thus for each node a graph with diagonal nodes is solved and path length is used as priority.

To draw conclusion, we ran 100 iterations of A* thinning for each thinning factor. Along with this we also included A* Manhattan, A* Euclidean and A* diagonal(Our relaxation based strategy) in these iterations (although they didn't have any dependency on thinning factor) to do comparison.

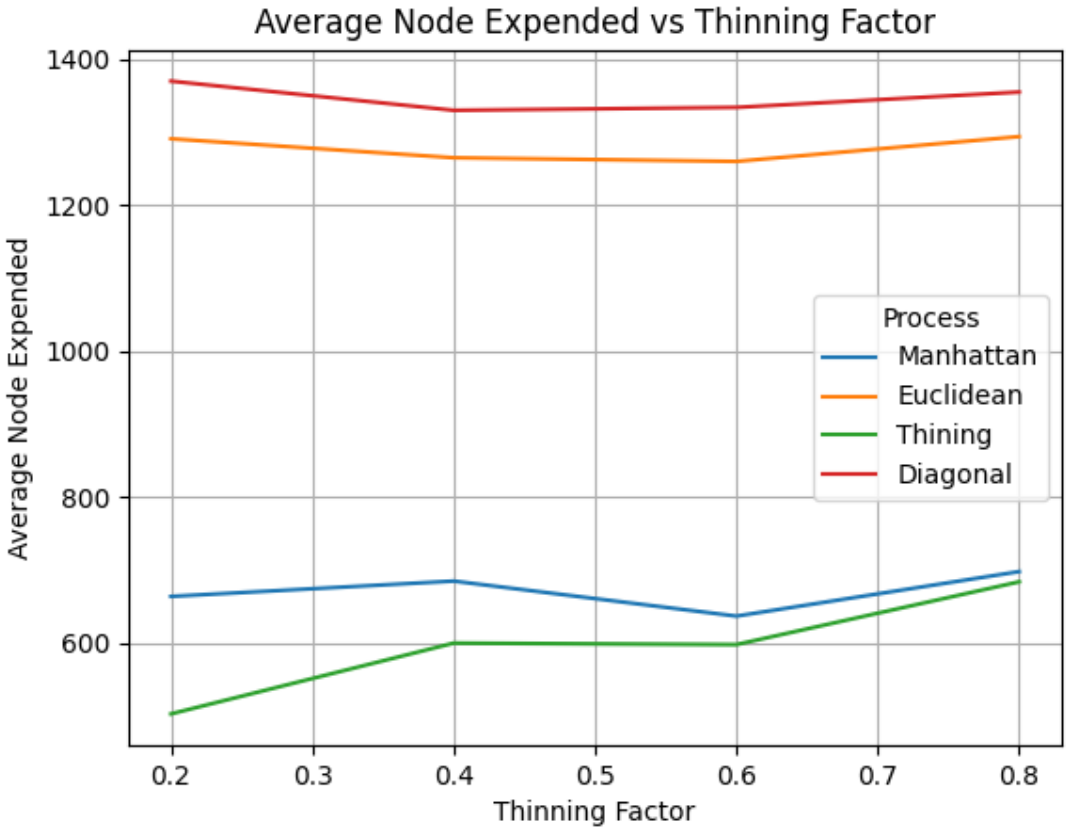


Figure 14: Avg number of nodes expanded vs Thinning Factor

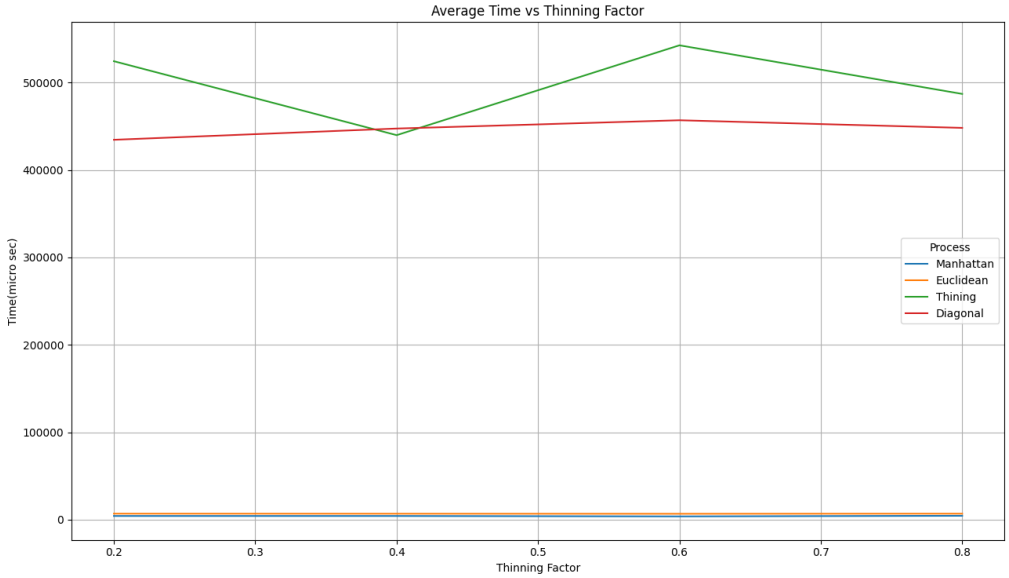


Figure 15: Average Time

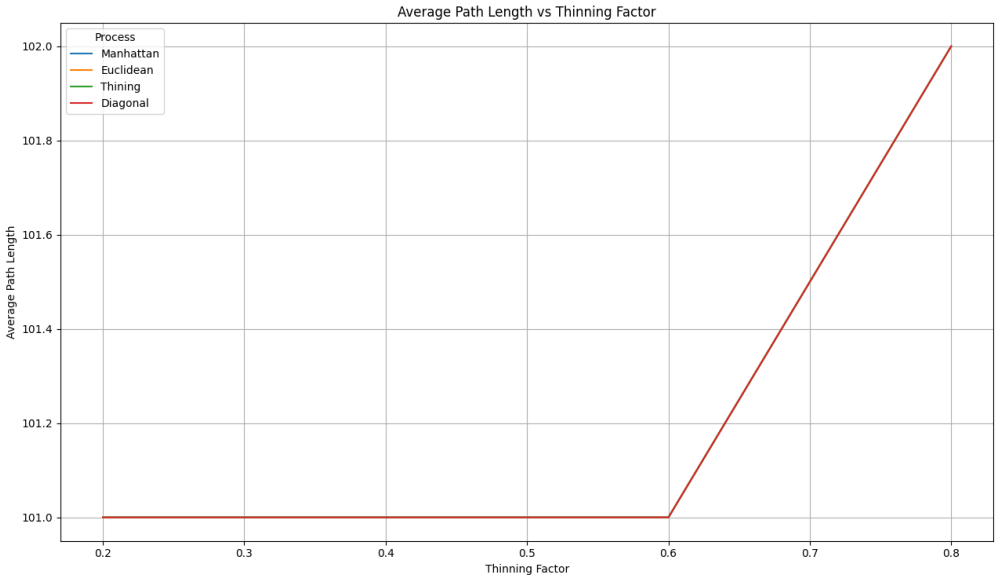


Figure 16: Average Path Length

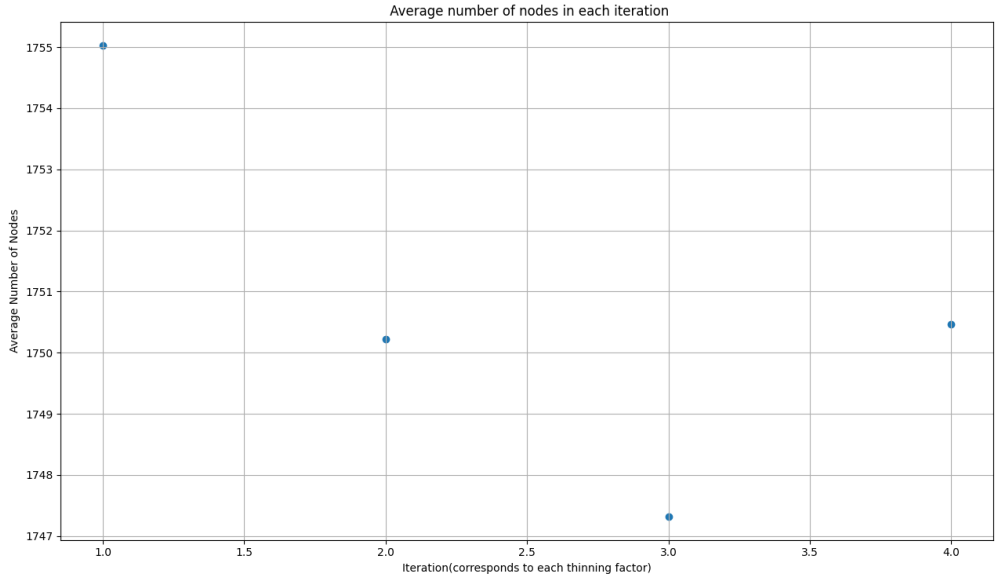


Figure 17: Average Total Nodes

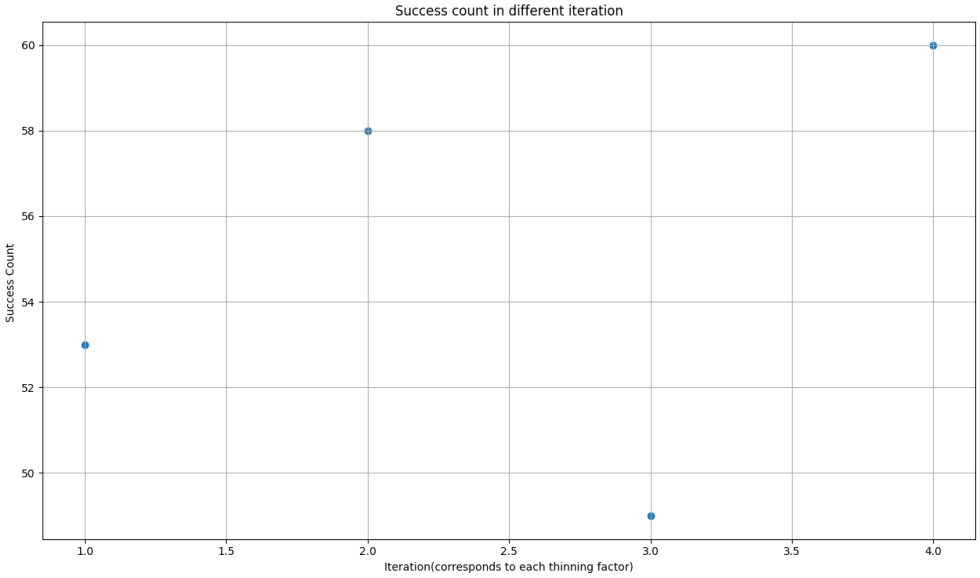


Figure 18: Success Counts

Conclusion

The graphs obtained in figures 14 to 16 use the other algorithms like Manhattan, Euclidean distance and diagonal distance as a means of comparing the performance of the thinning algorithm. For the analysis, we ran 100 iterations for each thinning factor.

The observations and inferences from these graphs are listed below:

- From Figure 18, we can see the number of success counts for each thinning factor (iteration) which is 50 on average.
- From Figure 17, we see the average number of total nodes for each iteration which was 1752 on average.
- We can also see in Figure 16 that for all the heuristics used in the A* algorithm, the average path length is the same for each thinning factor.
- Comparing these algorithms based on the number of nodes expanded and the average time taken, we observe that the Manhattan and thinning heuristics expand the least number of nodes.
- Additionally, these heuristics perform better than the diagonal heuristic constructed by us and the average time taken to solve the mazes is the least in the case of both these heuristics.
- Both A*-Thinning and A*-Diagonal take a longer time than both these heuristics; however, the A*-thinning algorithm takes slightly longer compared to the A*-Diagonal algorithm.

Answers

Are there anywhere you see a savings over the straight A*-Manhattan at this obstacle density?
A*-Thinning reduces the number of nodes that are expanded to search the path.

What additional time costs do you incur?
Additional cost for both A*-Thinning and A*-Diagonal is computing the path for each node. This increases the total time taken to solve the maze.

Do you think these approach is viable?.
Simple heuristic strategies like A* Manhattan and A* Euclidean will be preferred over traditional search algorithms as they reduce the number of nodes and time taken is also small. Heuristic strategies A* Manhattan though reduce no. of node searched but more time. Heuristic strategies suggested by us neither reduce number of nodes nor takes less time, though it takes less time than A* Thinning.

To conclude for very large mazes advance heuristic approaches will be very useful as they will save reduce the number of nodes that expended and for large mazes the time can be comparable to basic heuristics. Heuristic suggested by us is not viable as number of nodes expanded are high and for large mazes it will only increase.

How does the representation of the maze factor in to this (sparse vs not sparse, etc)?.
Maze factor is directly related to nodes expended as a dense maze will inherently have more nodes to expend and multiple possible path. Thus, a reasonably dense maze will be take less time to solve if solvable.

Can choice of strategy be dynamic?.
Yes, we can have dynamic choice of heuristic where we start with with complex heuristics like A* Star Thinning and as we get reasonably closer to destination (we can set a threshold for that). We switch to to basic heuristics like A* Manhattan. Using this we can have benefits of both strategy i.e reduced number of nodes with A* thinning and reduce time using A* Manhattan at later stages.

Code Overview

The Python files that we used for this project are listed below:

1. **createmaze.py** This file consists of functions that generates the 2D maze and the 2D thinned maze, by removing p blocked cells from the maze, which are then converted into their respective graphs.
2. **algorithm.py** This file includes the implementations of our search algorithms, namely BFS, DFS, Dijkstra's algorithm, Bidirectional BFS and Iterative Deepening DFS.
3. **solutions.py** This file contains the initialization of where the fire begins from and its spreading, along with the implementations of the strategies used to solve *The Maze is on Fire*.
4. **visualization.py** This file contains the code to display the mazes and the graphs for both the problems.
5. **analysis.py** This file contains the results obtained by running the strategies in problem 1 which is returned as a dictionary.
6. **maze_thinning.py** This file consists of the implementation of the priority queue used for the A* algorithm, along with the implementations of the Manhattan distance, Euclidean distance, diagonal distance, thinning algorithm and generates the results.

   GitHub Repository:Intro to AI - Project 1

Responsibilities

For this project, all the team members shared equal responsibilities, but each one of us took lead in the below mentioned parts of the project:

Kaustubh N Jadhav (knj25) - Algorithms and Readability of Code
Pranav Shivkumar (ps1029) - Report, Data Visualization and Representation
Sanyam Jain (sj770) - Coding