# CS 520 Introduction to Artificial Intelligence
**Professor Wes Cowan**
**Assignment 1: Maze Runner**

**Group Members:**
Keya Desai (197001879)
Prakruti Joshi (197002286)
Rushabh Bid (196005915)

## Part 1: Environment and Algorithms

For checking the correctness of our code, we tested our search algorithms on a maze with:
Dimension of matrix (dim) = 20
Probability of cell of the matrix being blocked (p) = 0.2
With start = (0,0) and goal = (dim-1, dim-1)

Representation of maze - We have represented the maze as a numpy 2D array of dimension (dim, dim). The blocked cell takes the value of 0, open-cell takes the value of 1 and start and goal as 0.5.

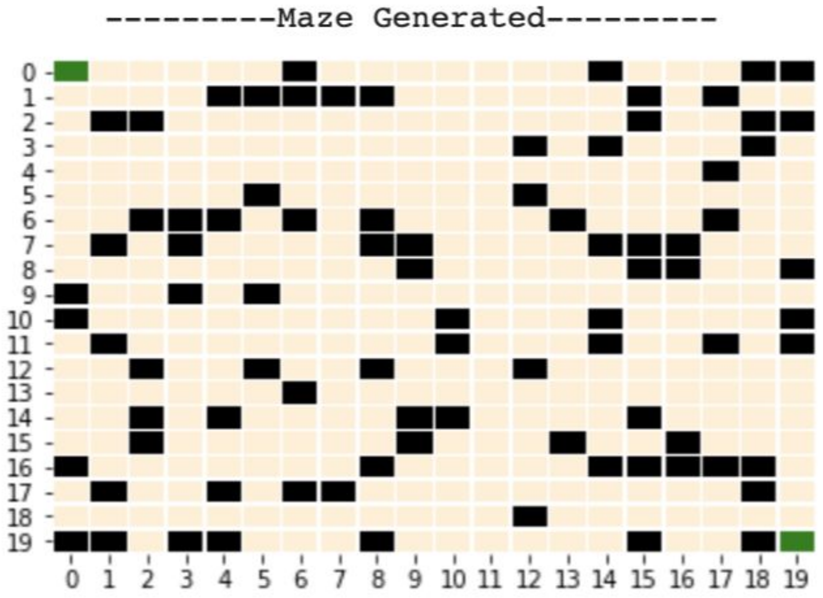


**Fig 1: Sample maze generated with dim = 20, p = 0.2**

On the maze generated, we implemented the following search algorithms to reach from start to goal.

1. Depth-First Search
2. Breadth-First Search
3. A*: where the heuristic is to estimate the distance remaining via the Euclidean Distance
4. A*: where the heuristic is to estimate the distance remaining via the Manhattan Distance
5. Bi-Directional Breadth-First Search

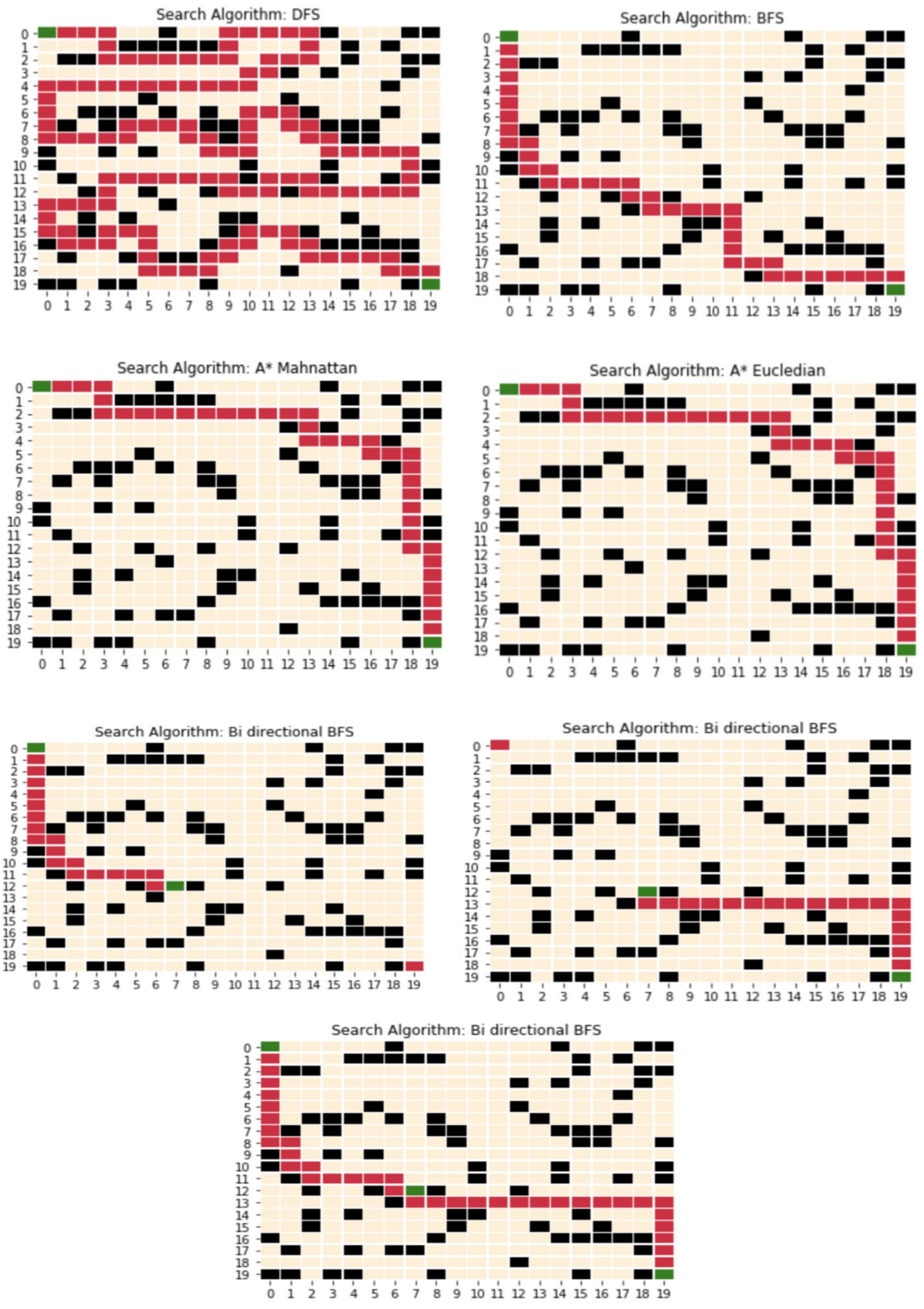
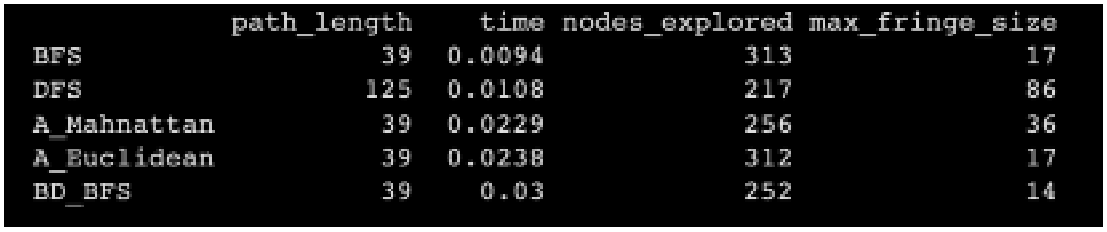


**Fig 2. Result of search algorithms for the random maze generated with dim=20, p=0.2**
**In Bi-Directional BFS, the intersection node is shown is green. The path from start to the intersection node and goal to intersection node is found using two independent BFS's.**

The algorithm returns failure if there is no path found from start to goal. If a valid path is found, the search algorithms returns success along with

- A list containing the coordinates of the path taken (This list is used to retrace the entire path)
- Count of nodes explored
- Max fringe size
- Visited nodes

We store this information in a data frame in order to easily compare the performance of the search algorithms.



```
             path_length    time nodes_explored max_fringe_size
BFS                   39  0.0094            313              17
DFS                  125  0.0108            217              86
A_Mahnattan           39  0.0229            256              36
A_Euclidean           39  0.0238            312              17
BD_BFS                39    0.03            252              14
```

**Fig 3. Data Frame storing the parameters required to analyze the performance of the search algorithms**

One direct conclusion that we can draw even at dim = 20 is that DFS does not always take the optimal/shortest path. This is because once the DFS finds a path. In order to draw other meaningful conclusions, we need to increase the dimension of the maze. This is the next step.

## Part 2 Analysis and Comparison

*1. Find a map size (dim) that is large enough to produce maps that require some work to solve, but small enough that you can run each algorithm multiple times for a range of possible p values. How did you pick a dim?*

In order to find the optimal map size(dim), the following parameters are taken into account:
(A) Execution time to find the path length - Time taken by the algorithm is important for us to design functionalities that require the algorithms to be run multiple times.
(B) No. of nodes explored in the entire maze in order to find the shortest path - More the number of nodes that an algorithm explores, more is the work required for that maze to be solved.

We need an algorithm that explores relatively fewer nodes and executed in shorter time so that the function can efficiently compute the path even for higher dimensions. Out of all the algorithms we implemented, Improved DFS runs in minimum time, and hence we have chosen Improved DFS for running this part.

Improved DFS is run to obtain both the parameters for a maze generated over a range of p values and dimensions.

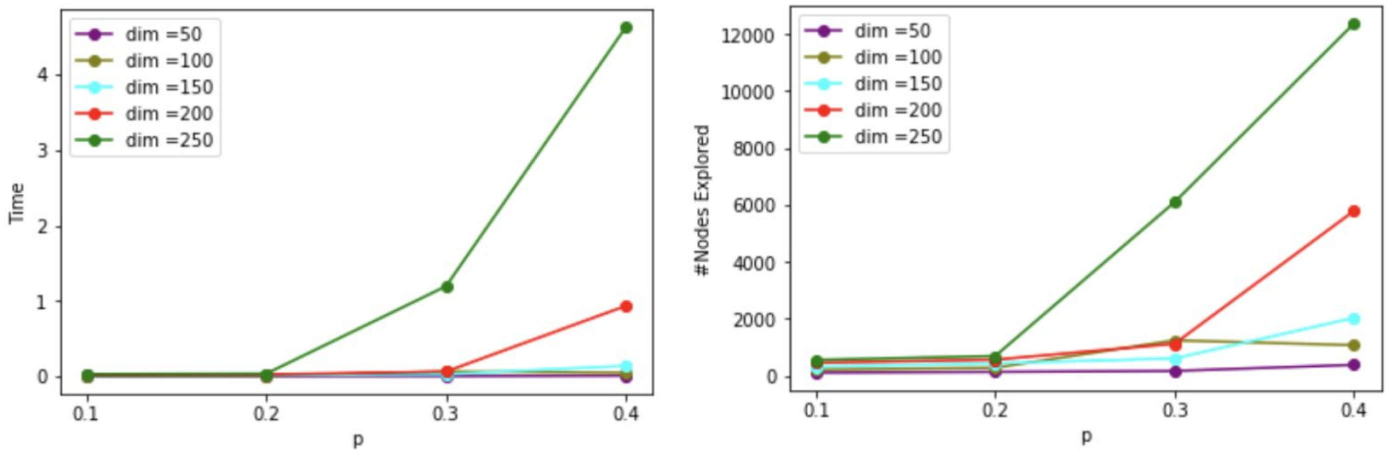


**Fig 4. Comparing Execution time and # nodes explored over a range of p values for different dimensions.**

From the figure, we can observe that as the p value increases, the time and number of nodes explored for each dimension increases. This increase becomes more significant with each p value at higher dimension. Since we have not decided upon a value of p, we decide to choose a dimension which does not have significant variability across a range of p values. Next we fix the p value at 0.2 and analyse further.

In order to take the variability of the generated maze into account, for a single dimension, the parameters are calculated for n_trials. If an unsolvable maze is generated, it is discarded.

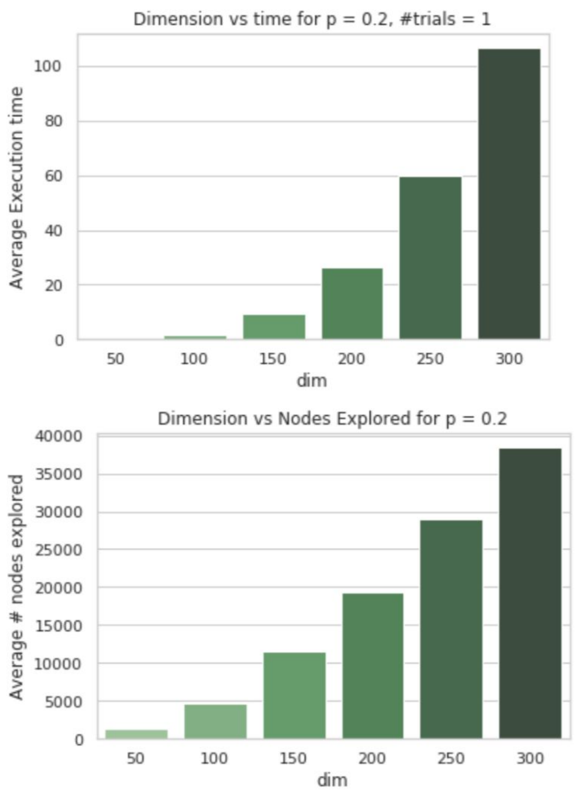


**Fig 5. Determining optimal dim by comparing the execution time and # nodes explored for each dim**

From the plot, in Fig 4, the time and the number of nodes explored is monotonically increasing as the dimension of the maze increases. For the purpose of our application, we need to select a dimension that takes less time but more work. As a middle ground,  we have selected a value that lies somewhere in the middle of the plot. Taking dim = 150

*<u>2. For p = 0.2, generate a solvable map, and show the paths returned for each algorithm. Do the results make sense? ASCII printouts are fine, but good visualizations are a bonus.</u>*

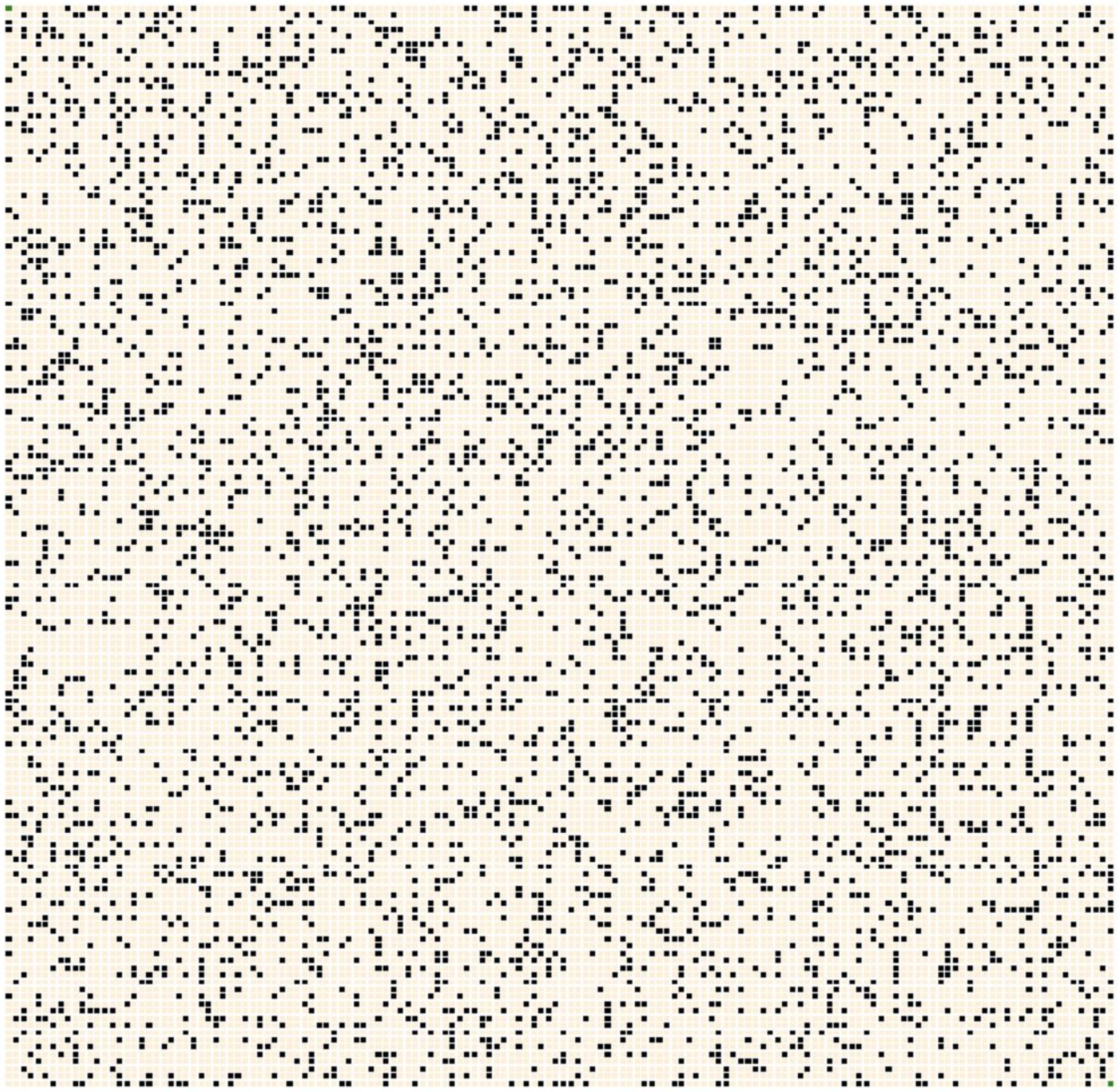
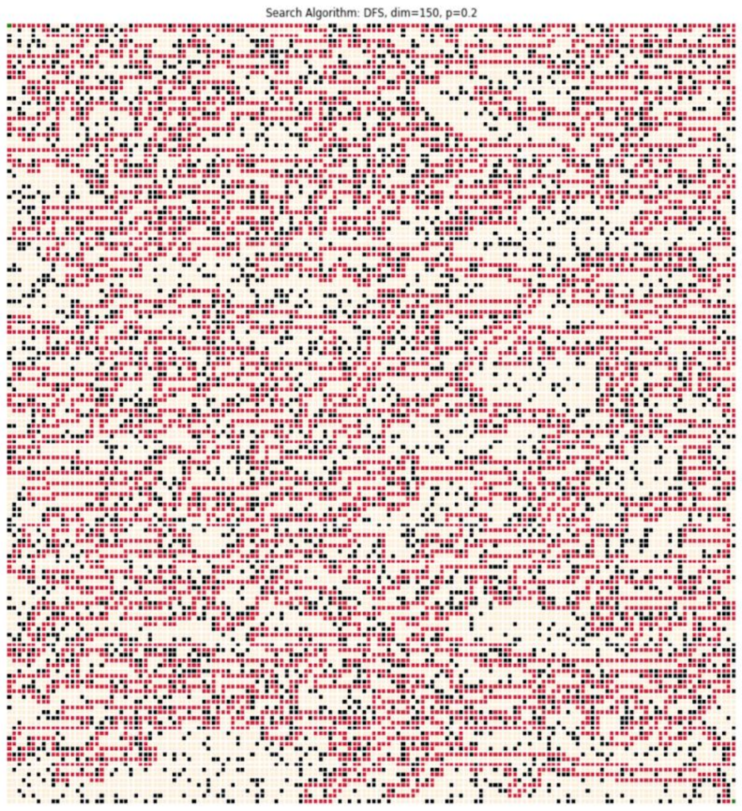
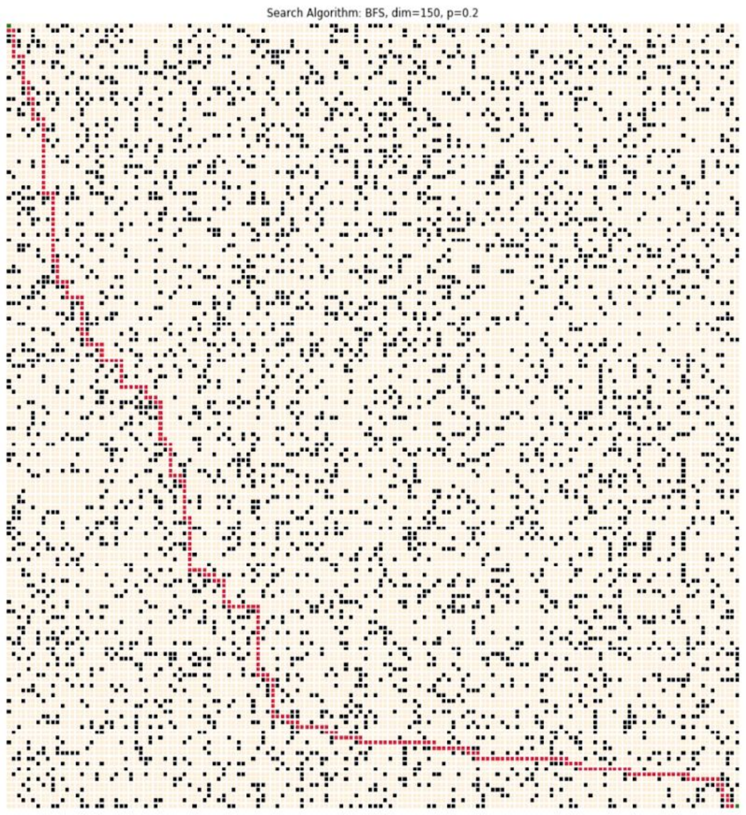


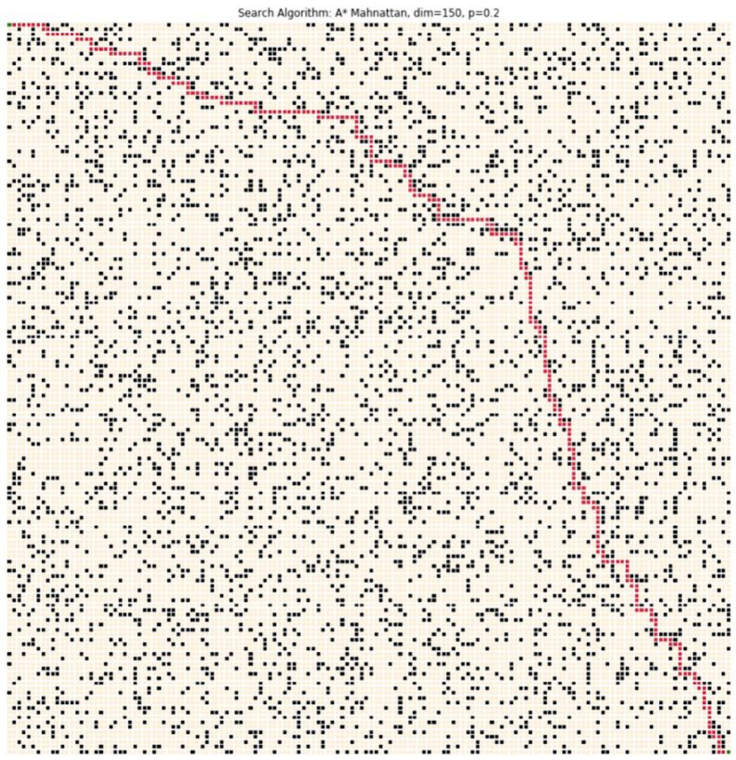
**Fig 6. Generated Maze for dim = 150, p = 0.2**



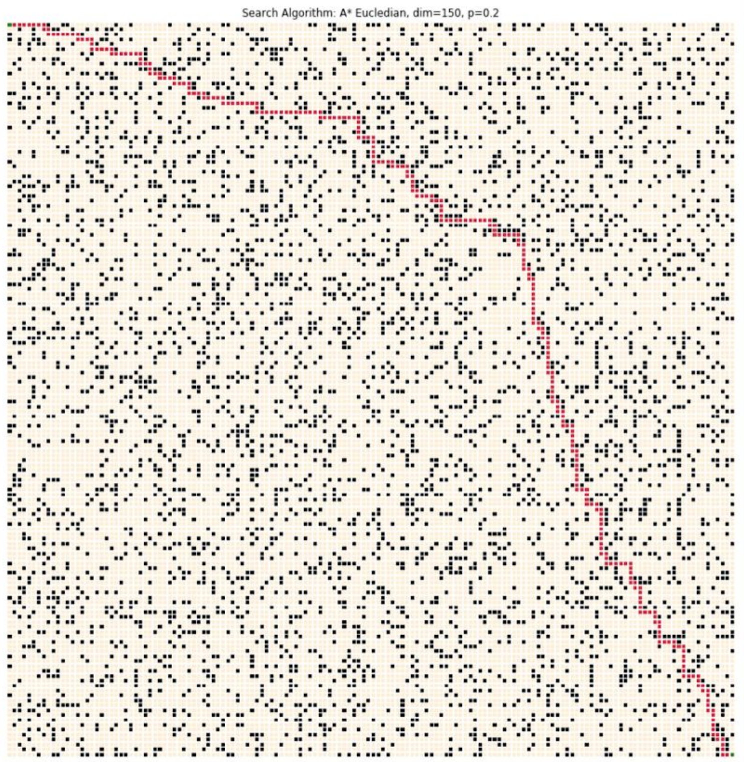
(a) DFS



(b) BFS



(c) A* - Manhattan



(d) A* - Euclidean

**Fig 7. Paths from start to goal returned by (a) DFS (b) BFS (c) A* - Manhattan (d) A*- Euclidean for the maze in Fig 5.**

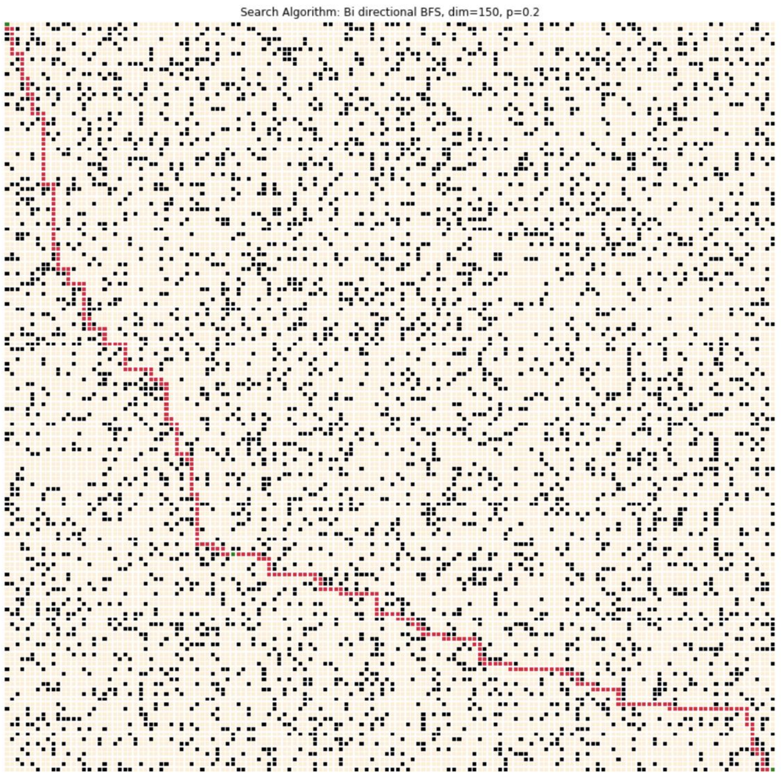
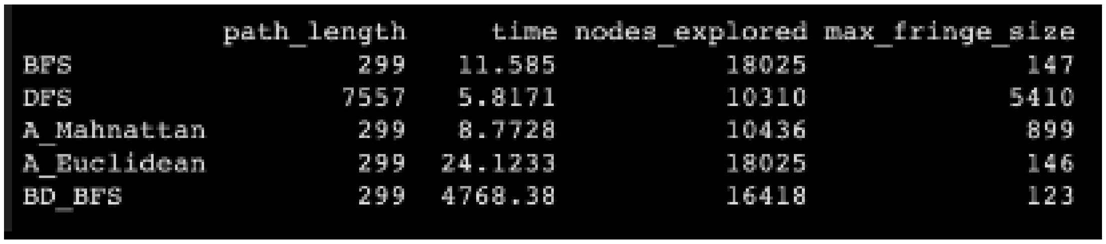


**Fig 8. Paths from start to goal returned by bidirectional BFS**



```
               path_length      time nodes_explored max_fringe_size
BFS                    299    11.585          18025             147
DFS                   7557    5.8171          10310            5410
A_Mahnattan            299    8.7728          10436             899
A_Euclidean            299   24.1233          18025             146
BD_BFS                 299   4768.38          16418             123
```

**Fig 9. Performance comparison**

*3. Given dim, how does maze-solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple mazes and checking them for solvability. What is the best algorithm to use here? Plot density vs solvability, and try to identify as accurately as you can the threshold p0 where for p < p0, most mazes are solvable, but p > p0, most mazes are not solvable.*

To check the dependency of maze solvability on p, we plot for a range of p values, the probability that maze will be solvable. For a particular 'p' value, we run the code 'n' times and see the number of times the randomly generated maze is solvable. The best algorithm, in this case, will be the one that takes the shortest time regardless of the path length. Here the goal is only to check only the solvability of the maze. Improved DFS takes the least time out of all the search algorithms and hence Improved DFS algorithms is chosen.

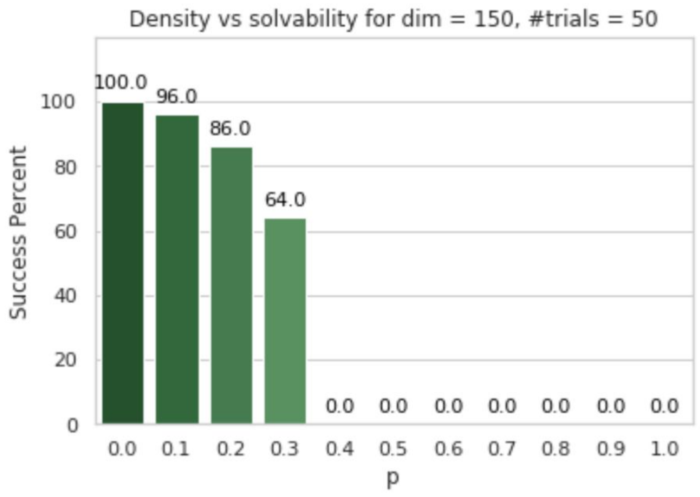


**Fig 10. Density vs Solvability**

From the plot, we can observe that the solvability becomes 0 at p = 0.4. For p <= 0.2, Maze solvability is more than 80%. For p = 0.3, it falls a little to 64% but still, more than 50% of the mazes are solvable. Hence, we can choose p_0 = 0.3.

*4. For p in [0; p0] as above, estimate the average or expected the length of the shortest path from start to goal. You may discard unsolvable maps. Plot density vs expected shortest path length. What algorithm is most useful here?*

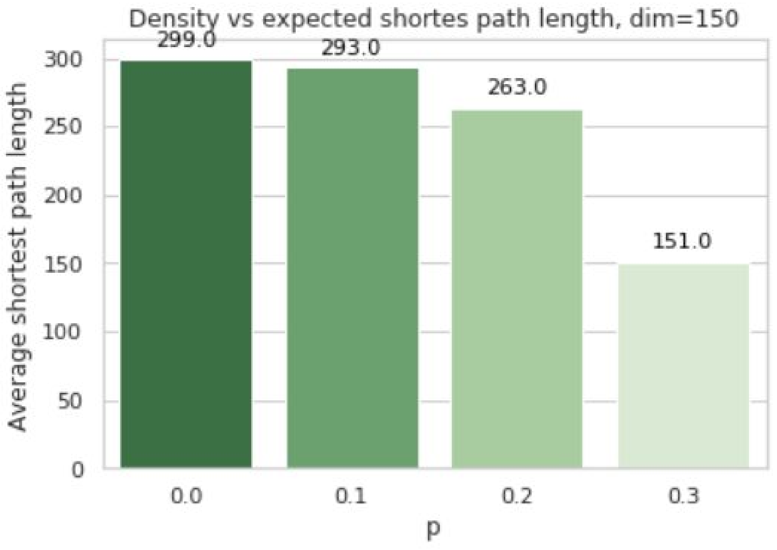


**Fig 11. Density vs Average Shortest Path**

BFS and A_star always returns the shortest path length. Both the algorithms can be used here but since A* is faster and requires less computation, here A* has been used.

*5. Is one heuristic uniformly better than the other for running A? How can they be compared? Plot the relevant data and justify your conclusions.*

Heuristics are the predictions of the path length from the current node to the end node. The better the heuristic, the algorithm will explore lesser number of nodes in reaching the goal. The two heuristics that we have considered in this assignment are Euclidean distance and Manhattan distance. Manhattan distance is the distance between two points measured along axes at right angles. In our environment where the motion of the path is restricted to vertical and horizontal movements, Manhattan should be a better heuristic than Euclidean distance. In environments where diagonal motion was also allowed, [ for example if the agent could move from (0,0) to (1,1) ], Euclidean distance would have been a better heuristic than Manhattan.

The performance of both heuristics, is compared on the parameters of time taken by the algorithm, the number of nodes explored and maximum fringe size.

| | **Execution time (in s)** | **# Nodes explored** | **Maximum Fringe size** |
|---|---|---|---|
| **A* - Euclidean** | 23.9 | 17,954 | 142 |
| **A* - Manhattan** | 8.45 | 10,290 | 879 |

**Table 1: Average of 5 trials**

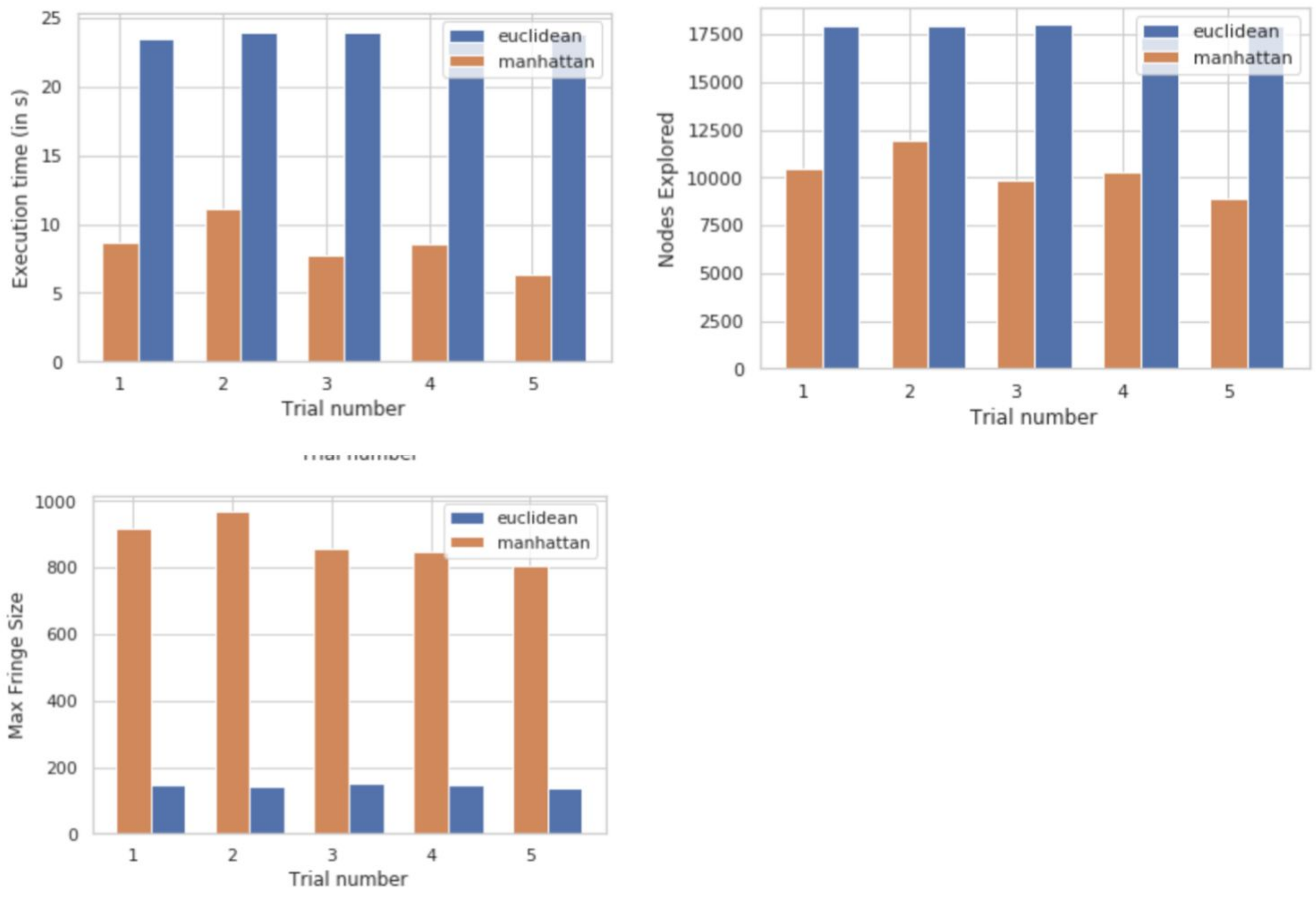
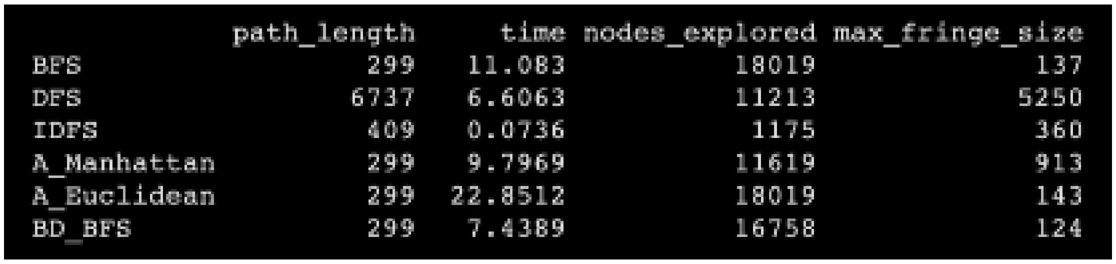


**Fig 12. Comparison between the performance of heuristics of Manhattan and Euclidean for a maze generated using dim = 150, p = 0.2.**

As seen in the figure, Manhattan heuristic takes significantly less time and explores lesser number of nodes to find the shortest path from start to goal than Euclidean distance. But the use of euclidean distance solves the maze with a lesser fringe size than Manhattan distance. The explanation is the same for both these arguments. Euclidean distance can be thought of as a relaxation that you can fly over the maze, while Manhattan distance is a relaxation than you can walk through obstacles. Manhattan distance is a closer estimate and thus allows comparatively more pruning of the sample space. If more nodes are pruned, it is obvious that A* based on Manhattan distance will explore lesser nodes. Along the same lines, since Euclidean distance allows lesser pruning and is a more optimistic heuristic, it demands more nodes on different branches be explored. Since, it keeps switching between branches, the fringe size is reduced as nodes are popped out of the fringe at a similar rate as they are pushed in. The difference between the nodes explored is much more significant than the fringe size difference between manhattan and euclidean distance, and using the manhattan heuristic takes lesser time. Hence, even though the manhattan heuristic is not *uniformly* better than euclidean, but overall it is the better choice.

6. Do these algorithms behave as they should?



```
             path_length     time  nodes_explored  max_fringe_size
BFS                  299   11.083           18019              137
DFS                 6737   6.6063           11213             5250
IDFS                 409   0.0736            1175              360
A_Manhattan          299   9.7969           11619              913
A_Euclidean          299  22.8512           18019              143
BD_BFS               299   7.4389           16758              124
```

**Fig 13. Comparison**

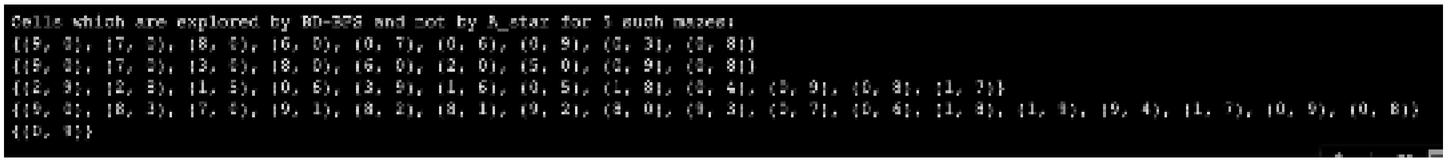
1. Since DFS does not explore the entire search space, the path returned is not the most optimal. It returns a path longer than any other search algorithm. But since it does not explore the entire maze, it is faster. However, DFS adds the neighbouring nodes at each step in the fringe. Since it takes a much longer route to reach the goal, the number of neighbouring nodes added in the fringe becomes larger. Thus, the fringe size of DFS is largest.
2. Out of BFS and A*, A* explores lesser nodes as it using heuristics to explore the nodes. It does not expand the nodes which has higher heuristic. As a result, the time taken by A* is less too.
3. As discussed in 5., the overall performance of A* Manhattan is better that A*Euclidean.
4. BD-BFS has two BFS's running in two portions of the maze. Since, BFS always returns the shortest path, the path length of each half is also halved. This reduces the time complexity by an exponential factor of half and is much faster than normal BFS.

7. For DFS, can you improve the performance of the algorithm by choosing what order to load the neighboring rooms into the fringe? What neighbors are `worth' looking at before others? Be thorough and justify yourself.

Rather than choosing a random or typical order to load the neighboring rooms into the fringe, we can figure out some parameter based on which some neighbors might be worth looking into before the others. For instance, to move from the top-left corner of the maze to the bottom-right, the rooms below and to the right of the current room might be more desirable choices in a scenario in which there is no reason to believe they don't lead to the goal. Manhattan distance as a heuristic for the neighboring rooms can be an even better option. Manhattan distance estimates the distance to the goal considering a relaxation that you can move through the obstacles. This allows us to choose between the bottom and right rooms as without this heuristic, both of them are equally desirable. Using this method improves the performance of the algorithm drastically as the prior version of DFS randomly roams around the maze without any informed decisions based on where it wants to reach. Even though the improved DFS algorithm might not return an optimal path or minimal fringe size, in terms of space and time complexity, it is drastically better than all other search algorithms that we have discussed.

8. On the same map, are there ever nodes that BD-DFS expands that A doesn't? Why or why not? Give an example, and justify it.

- A* uses the concept of heuristics to find the optimal solution to a problem. The functioning of A* depends on the goodness of the heuristic. The closer the estimate to the actual cost, the better it is, as it helps prune more nodes. So, A* will never explore nodes that will lead to a higher cost based on the estimate.
- Bi-directional BFS starts a breadth first search from the start as well as the goal. At each expansion both the fringes are compared to check for intersecting nodes. This algorithm does not check for any estimates and explores all possible nodes in both directions until an intersection is found. This is likely to explore more nodes than actually required.
- Evidently, Bi-directional BFS explores more nodes than A* as it does not check for any estimates. Additionally, the nodes closer to the goal will not be explored by A* if they are reachable from the start through a longer path. However, Bi-directional BFS will start the reverse search from the goal and explore them at a very early stage. The same argument can be made for nodes away from the goal as well.



```
Cells which are explored by BD-BFS and not by A_star for 5 such mazes:
{(9, 8), (7, 9), (8, 6), (6, 0), (0, 7), (0, 6), (0, 9), (0, 3), (0, 8)}
{(9, 8), (7, 9), (3, 6), (8, 0), (6, 0), (2, 0), (5, 0), (0, 9), (0, 8)}
{(2, 9), (2, 8), (1, 5), (0, 6), (3, 9), (1, 6), (0, 5), (1, 8), (0, 4), (0, 9), (0, 8), (1, 7)}
{(9, 8), (8, 3), (7, 6), (9, 1), (8, 2), (8, 1), (9, 2), (8, 0), (9, 3), (0, 7), (0, 6), (1, 8), (1, 9), (9, 4), (1, 7), (0, 9), (0, 8)}
{(0, 9)}
```

**Fig 14. Check for nodes explored by BD_BFS but not A_star for a maze of dim =10**

*Bonus: How does the threshold probability p0 depend on dim? Be as precise as you can.*

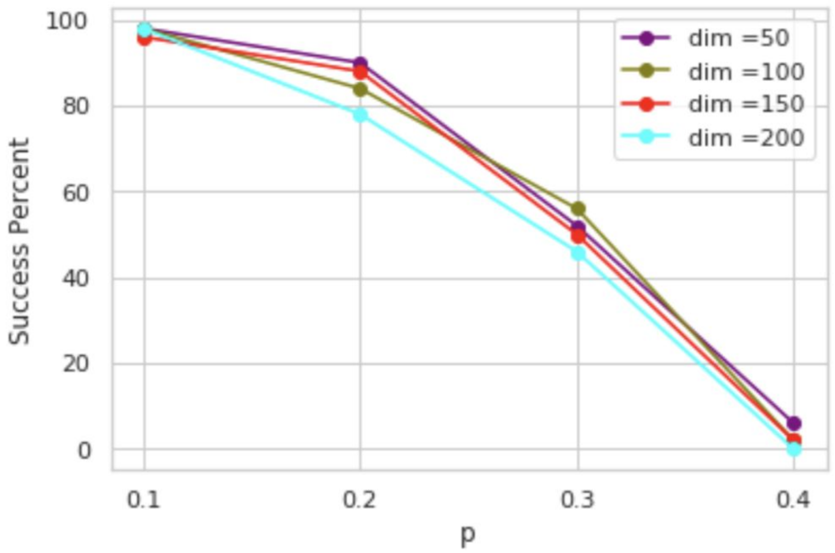


**Fig 15. Success rate for different dimensions v/s probability**

To determine how the threshold probability depends on p, we plotted the maze solvability percent against p, for a range of dim values. From the plot, it is clear that, as the dim increase, the maze solvability at a particular p decreases but the decrease is not significant enough to affect the threshold probability p0. For each dimension, p0 comes out to be 0.3.

## Part 3: Hard Maze

We have used two local search algorithms to generate a harder maze than previous maze.

### 1. Genetic Algorithm:

● Here, the state is the maze of dimension - (dim). A population consists of multiple randomly generated mazes (with probability of cell being blocked as p=0.2) out of which some are solvable and some are not.
● An essential feature of genetic algorithm is that the properties of the state are inherited from parents to children. Here, the dimensions of the maze are preserved during crossover of parents. The child has the same dimensions as that of the parents. For example, if parents are a maze of size [10,10], a valid crossover is taking rows [1-5] from parent-1 and rows [6-10] from parent-2.

**Algorithm:**
○ A population is created by generating 50 mazes with dimension [50x50] and obstacle probability 0.2
○ The fitness of this population is checked for each individual maze. The mazes are solved using A* algorithm with manhattan distance as the heuristic.
○ After exploring different functions based on certain parameters as the cost function (explained below), the algorithm is used to check the fitness of each maze.
○ Based on the fitness, we shortlist the parents for crossover. We have used probability distribution for selection of parents so that parents with lower fitness function are also given some chance to be selected. The probability increases exponentially with fitness of parents.
○ After selection of parents, they are paired up for crossover. Here, parents with higher fitness (out of the selected parents) are given higher probability to mate. Exponential probability distribution is used here as well. We wish to generate children equal to the size of the original population.
○ The children generated after crossover are mutated to generate additional diversity in the new population. In our case, mutations are performed in the form of addition and deletion of obstacles.
○ These mutated children are added to the population and their fitness is calculated in terms of cost to solve the maze.
○ The exponential probability distribution is used yet again, to trim the new population by a factor of half (amounting to the original population size).
○ This new trimmed population becomes the new generation and the same procedure is applied to it to generate multiple generations.
○ An important thing to keep in mind while implementing genetic algorithm is that a population should be diverse. Keeping just the fittest parents is not necessarily the best idea; as less diversity leads to a smaller sample space for the next generation. To broaden the sample space, probability is employed at each step such that there is a slim chance for elimination of fit parents and selection of weak parents.

### Design choices in this algorithm:
○ *Population size :* 50
This population size remains constant throughout each generation.

○ *Fitting function* :
Fitting function determines how hard the maze and is evaluated by the following parameters:
1. path length to reach the goal (calculated the shortest path)

2. number of nodes explored
3. maximum size of fringe.

Different fitting functions that we tried on the basis of the above parameters are:

| Fitting/ Cost Function | Difference percentage |
|---|---|
| Shortest path length | 108.47 |
| Number of nodes explored | 2.8 |
| Max. fringe size | 28.01 |
| Shortest path length + Number of nodes explored + Max. fringe size | 11.06 |
| 1.08* (Shortest path length) + (0.28) Max. fringe size | 72.7 |

**Table 2: Fitting functions for Genetic Algorithm**

The higher the value of the fitting function, the harder the maze is for the search algorithm to solve. The objective is to maximize the value of fitting function as much as possible. Depending on the maximum percentage difference between the cost value of the original maze and the hardest maze generated from multiple iterations, we have chosen the optimal fitting function to be:

*Fitting function = Shortest path length*

○ Selection of parents for breeding :

The probability of selection of a given parent is determined by the probability distribution below:

$$P(X_i) = \frac{1.1^i}{\sum_{k=0}^{n} 1.1^k}$$

This distribution assigns minimal probability of selection to weak parents and minimal probability of elimination to the fitter ones. This small chance of moving away from convergence, increases the diversity of the population and in turn is more likely to provide a better result.
In a similar manner, the pairing of parents for crossover is done using probability. The fitter parents are given more chances to mate as compared to the weaker ones.

○ *Crossover methods* :

1. Single point crossover
2. Two point crossover
3. Uniform crossover
4. Logical operation

- We decided not to use logical (And and Or) operation on individual cells of parents as the operations will either increase the number of obstacles drastically rendering it unsolvable or decrease the number of obstacles at a faster rate. In both the cases, the population generated is not ideal for our problem statement. Thus, we have used other crossover methods.
- In single point and uniform crossover, we have kept a ratio parameter *'alpha'* which generates one of the children with higher proportion of the fitter parent as *'alpha'*.
- In all the crossover methods- single point, two point and uniform, we have merged the parents columnwise.
- We observed that two-point crossover and uniform crossover have comparable results (72% difference percentage between cost value of original and hardest maze). This is better than single point crossover which has 52% difference percentage.
- Thus, we have used ***two-point crossover*** in our algorithm.

○ *Mutation* :

Number of mutations = 0.1* (total number of cells in maze)

For example, if the dimensions of the maze are 10x10,

The number of mutations performed will be: 0.1x100 = 10

Total number of cells = (dimension)^2

Two types of mutations:

a. Addition of obstacle (Number of mutations/2)

b. Deletion of obstacle (Number of mutations/2)

○ *Pruning of population :*

After generation of children the size of the population doubles which demands pruning as otherwise the population size will keep increasing exponentially which will proportionally affect computational requirements. The pruning of population employs probability distribution again. At each step, the population becomes more diverse while still keeping the majority of the fitter individuals.

○ *Convergence:*

Algorithm converges after a fixed number of generations (in our case 100) or value of fitness function becomes 5 times the original fitness.

Advantages: This approach gives us a time efficient solution to generate a hard maze.

Disadvantage: A maze having fitness function greater than 5 times the original fitness function may exist.

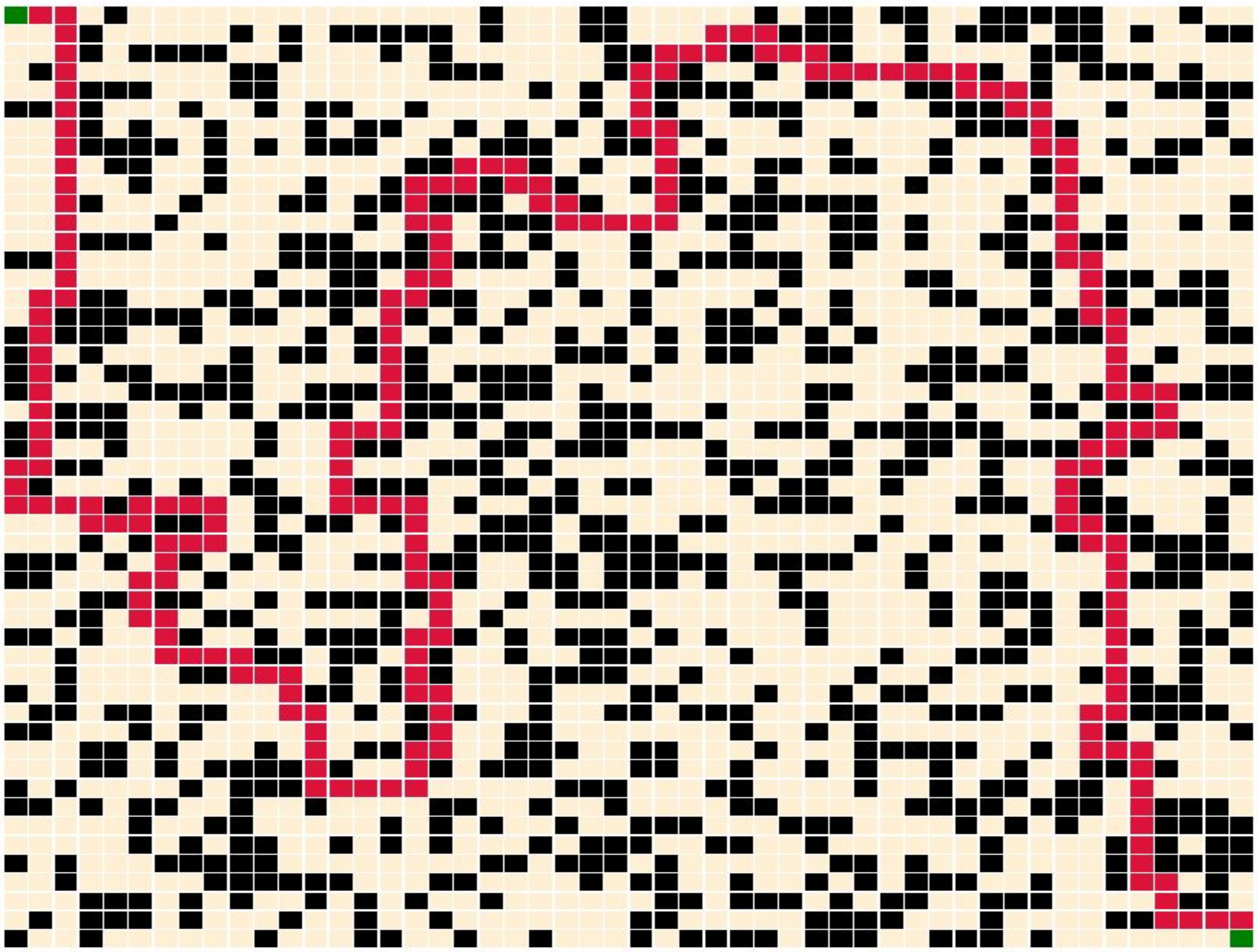


**Fig 15. Hard Maze generated by Genetic Algorithm**

**A*-Manhattan with Maximal Nodes Expanded:**
A* algorithm is guaranteed to find the shortest path. The uninformed searches that give the shortest path generally need to search all possible paths in order to know that the shortest path is indeed the shortest one. The reason this happens is because in the absence of extra information, it would be wrong to assume that one path is shorter than the other without actually exploring both of them. On the other hand, A* algorithm uses heuristics to prune the search space. This statement can mislead you to believe that the count of nodes explored by A* might be a good metric to determine the hardness of a maze. A* algorithm can help you prune the search space, but how much you can prune depends on how accurate the estimates are. Manhattan distance is an estimate based on the relaxation that you can move through obstacles. This will help you prune the nodes with a higher estimate than the smallest estimate to the goal. But, an important point to consider is that a maze becomes harder due to the obstacles and estimates with relaxations based on ignoring the obstacles, may not provide any information on how hard the maze is. The algorithm will explore a certain node irrespective of the actual distance from the goal. Hence, maximizing the nodes explored by A* - Manhattan doesn't seem like a practical approach to obtain a hard maze.

## 2. Hill Climbing:

### Algorithm:

- ○ A maze is generated with a given dimension and probability of obstacles.
- ○ The cost of this maze is calculated in terms of a weighted sum of different parameters. The improvised Depth-First Search algorithm is used to solve the mazes and calculate the costs.
- ○ In hill climbing algorithm, we keep moving towards optima till we find one. This optimum may be on of the local optima. In the case of mazes, we move towards optima by adding obstacles in the path followed in the original maze. To compensate for the addition of obstacles, some chance is given to remove some obstacles as well.
- ○ Local optimum is obtained when the addition of an obstacle makes the maze unsolvable. At this point, we can choose to accept the optimum, or use some technique to search for a better optimum. In this algorithm, the concept of random restart is used to begin the search for a better optimum. To restart the search, we remove an obstacle and start adding obstacles along the path again.
- ○ We repeat this procedure till we meet a set convergence condition. The convergence condition used is either the algorithm runs for a set number of random restarts or the algorithm increases the cost to solve the maze by a given factor (7 in our case).
- ○ The hardest maze is stored and returned as a result of the algorithm. To make more room for a better result, the hill climbing algorithm is run several times on different mazes.
- ○ The hardest of the mazes returned over these several iterations is returned as the final hard maze.

### Parameters in Hill Climbing:

- ● Fitness/Cost function:

Similar to genetic algorithm, fitting function determines how hard the maze and is evaluated by the following parameters:

1. path length to reach the goal (calculated the shortest path)
2. number of nodes explored
3. maximum size of fringe

Depending on the maximum percentage difference between the cost value of the original maze and the hardest maze generated from multiple iterations, we have chosen the optimal fitting function to be:

*Fitting function = Shortest path length + Number of nodes explored*

| **Fitting/ Cost Function** | **Cost value of original maze** | **Cost value of hardest maze** | **Difference** | **Difference percentage** |
|---|---|---|---|---|
| Shortest path length | 41 | 95 | 54 | 13.17% |
| Number of nodes explored | 68 | 178 | 110 | 161.7% |
| Max. fringe size | 52 | 38 | 14 | 36.8% |
| Shortest path length + Number of nodes explored + Max. fringe size | 113 | 188 | 75 | 66.3% |
| **Shortest path length + Number of nodes explored** | 78 | 244 | 166 | 212.8% |
| (1.3*Path length) + (1.6*Number of nodes explored) + (0.36*Max fringe size) | 152 | 446 | 294 | 192.1% |

**Table 3. Fitting function for Hill Climbing Algorithm**

- Number of restarts:

We plotted the cost/fitting value as a function of number of restarts (Fig. ). We can clearly observe that after 100 restarts, the function is quite constant and thus the variations in maze produce a comparable maze. Thus, we have kept the number of restarts as 100 before the algorithm concludes for a particular maze.

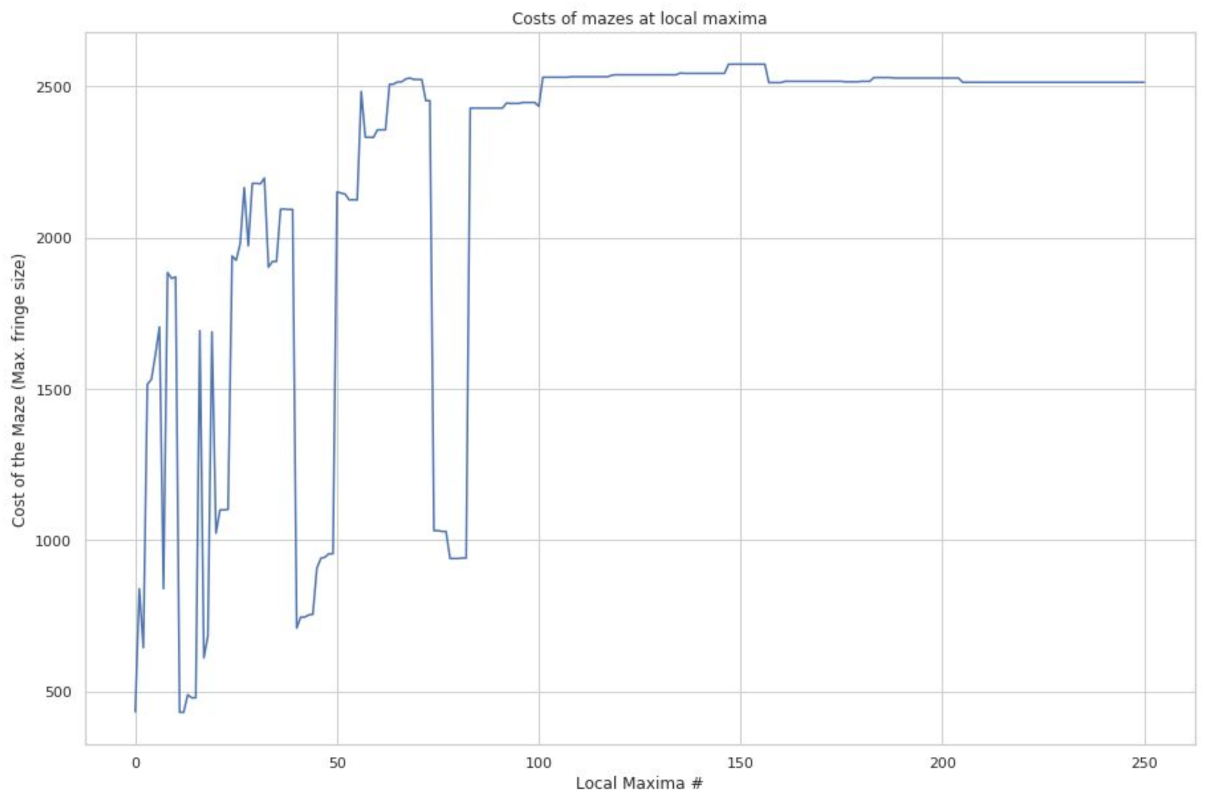
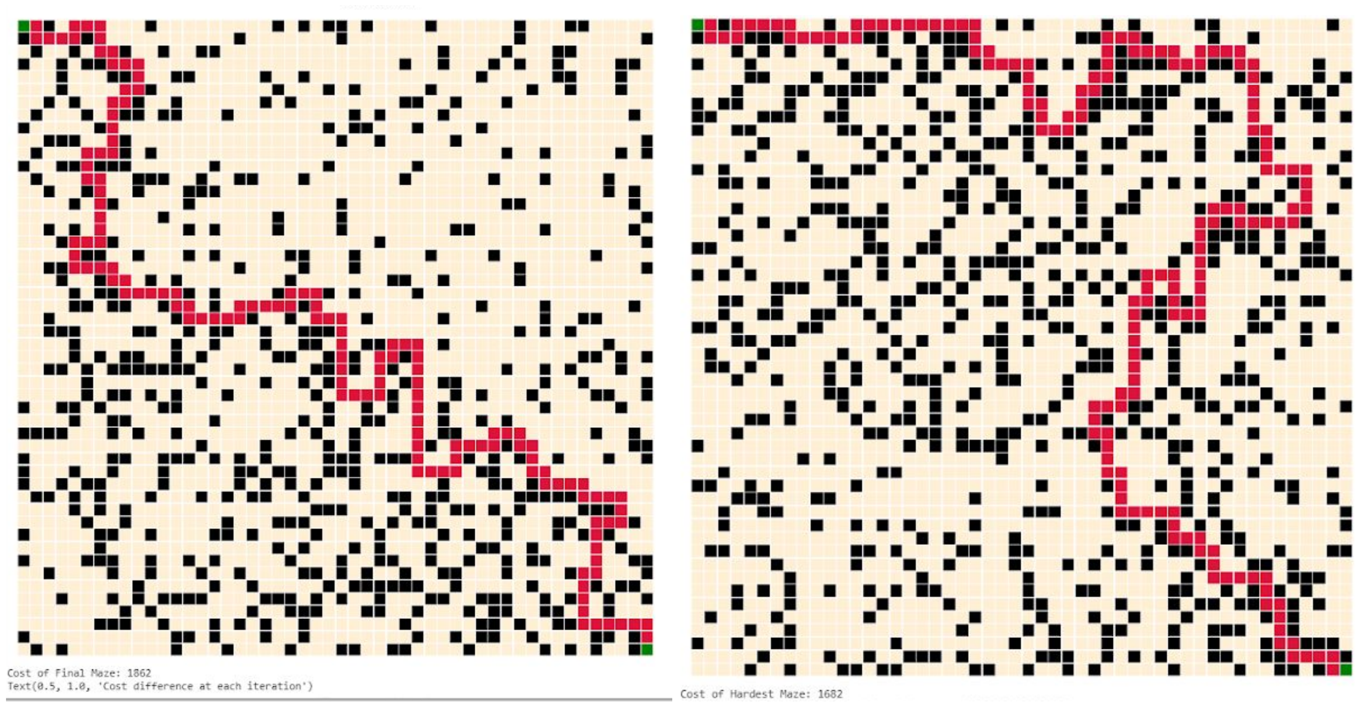


**Fig 16. Cost of the mazes at local maxima**



```
Cost of Final Maze: 1862
Text(0.5, 1.0, 'Cost difference at each iteration')
```

```
Cost of Hardest Maze: 1682
```

**Fig 17. Hard Maze for computer (left) vs Hard Maze to Human Eye (right)**

One important observation during solving this hard maze problem was that visually hard mazes may not necessarily imply computationally hard maze and vice versa. The maze on the left might not look as difficult as it could be to a human eye. In fact the maze on the right might look more difficult than the one on the left. But when we compare the costs, we find out that the cost of the left maze is more than that of the one on the right. This is because the computer doesn't look at the problem in the same way as humans do.

**DFS with Maximal Fringe Size:**

The metric of maximal fringe size paired with the DFS algorithm might seem like a plausible option to consider it as a cost of a maze. However, we should think about the approach used by DFS before jumping to conclusions. Depth first search, as the name suggests, searches the entire depth along a path to the point where further expansion is either not required, or not possible. Thus, in depth first search, at each step, the size of the fringe is only extended by the children of one node, which would lead to a linear increment based on the branching factor. So, even for a harder maze, the size of the fringe wouldn't be large enough for it to be considered a good enough metric for comparison.

## Comparison between hill climbing and genetic algorithm:

- Hill climbing and genetic algorithm are both local search algorithms. In terms of time efficiency, hill climbing is better as it simply mutates the maze and checks for fitting value. In genetic algorithm, additional computation is required in terms of generating children and pruning the population.
- In terms of space complexity also, genetic algorithm is better since it only has two mazes - current maze and hardest maze. Whereas, genetic algorithm requires maintaining a population of N in every generation.
- Genetic algorithm has the advantage that it introduces greater diversity at every generation. This is done in the steps where parents are selectively chosen for breeding, crossover methods, mutations and pruning of population. This diversifies the search space, thus ensuring local convergence.
- Both hill climbing and genetic algorithm will find a local optima and cannot guarantee convergence to global optima.

## Part 4: Maze on Fire

Baseline approach:

Choose the shortest optimal path in the initial maze irrespective of the fire spreading. Check the outcome.

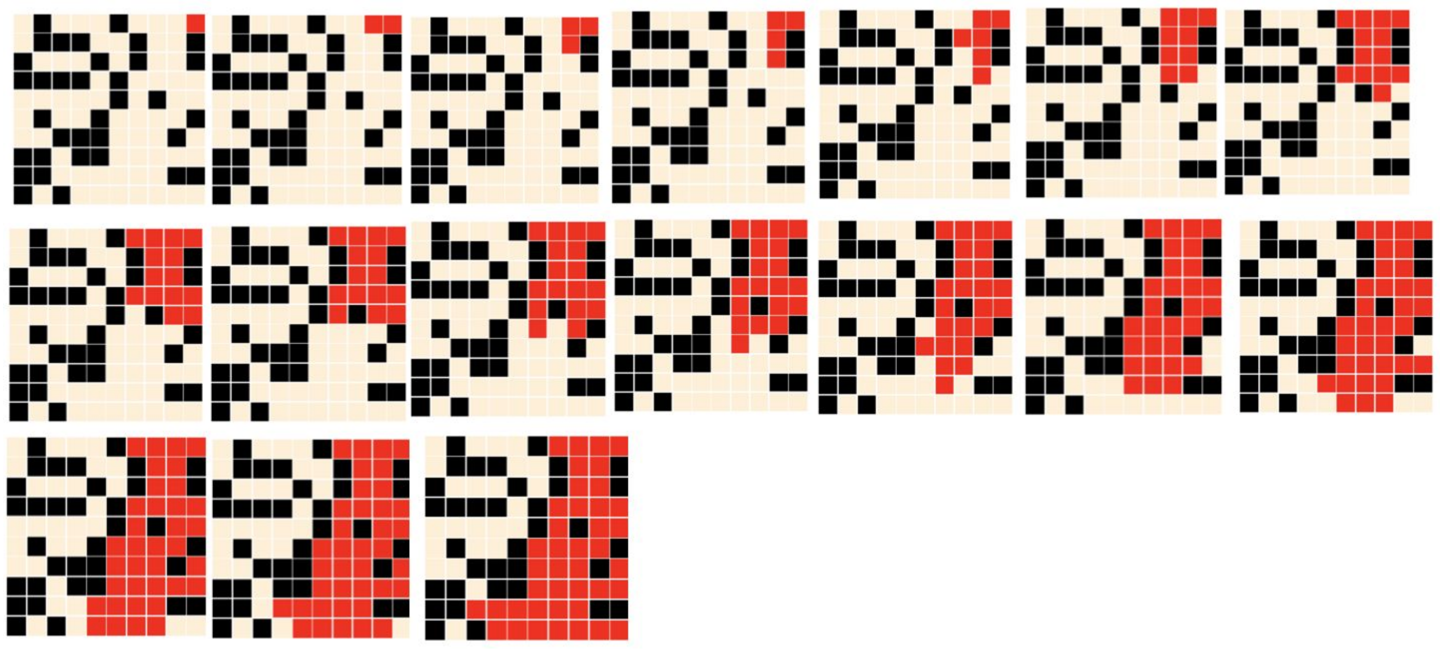


**Fig 18. Spread of fire with q = 0.5, dim = 10**

The plot of probability of success of baseline algorithm as a function of q (the flammability rate ranging from 0-1). The dimension of the mazes is (150x150). The number of mazes generated per q is 50.

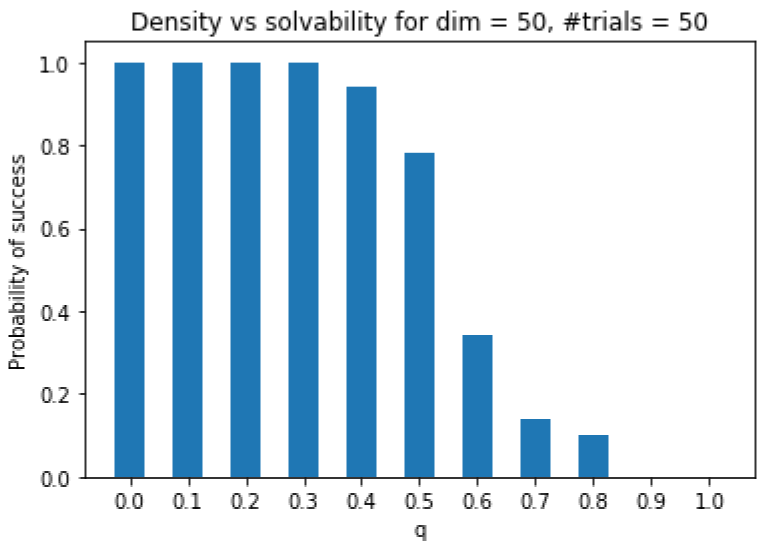


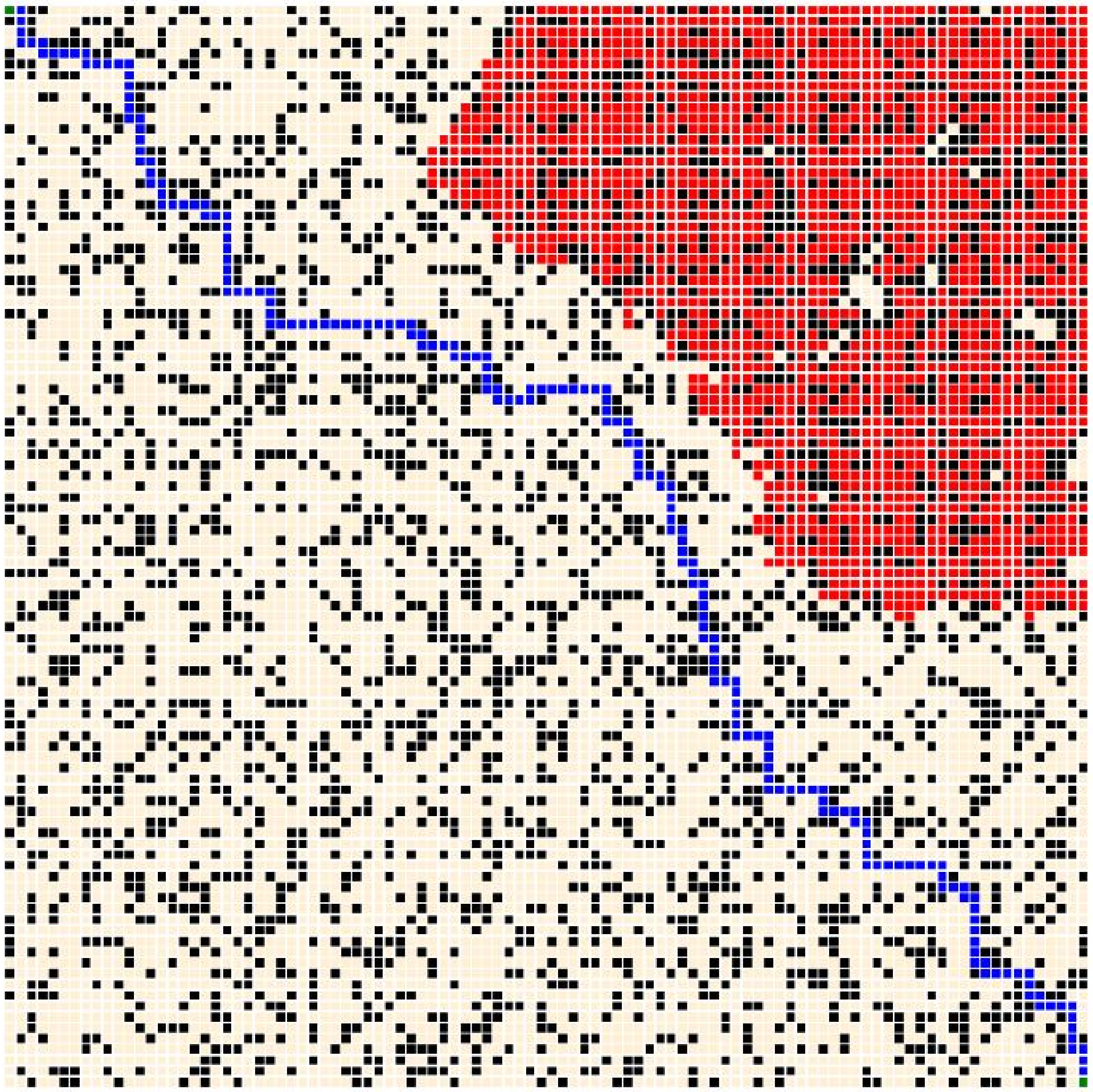
**Fig 19. Success rate of baseline with p = 0.3, dim = 50 v/s q**

*Can you do better? How can you formulate this problem in an approachable way? How can you apply the algorithms discussed? Build a solution, and compare its average success rate to the above baseline strategy as a function of q. Do you think there are better strategies than yours? What would it take to do better? Hint: expand your conception of the search space.*

**Approach:**
Since the fire is spreading based on a probability function, it will be computationally extensive to predict the complete spread of the fire before hand. Hence the best approach for our search algorithm at any step t would be a greedy approach to move to the neighboring room which is currently the most further away from the fire but near to the goal.

**Search Algorithm used - Improved DFS:**
The depth-first search algorithm does not guarantee optimality, but is it something that we always need to consider? Even though there are several algorithms that guarantee optimality, they may not be useful in the scenario in which the goal needs to be found faster. Shorter paths are obviously faster than longer ones, but they may not be a practical choice because of the additional time taken to explore the non-optimal paths and reject them. The improved DFS searches one branch of the search space till it finds the goal or a leaf node (dead end) while prioritizing children based on estimates. As a result of this approach, the improved DFS algorithm expands the least number of nodes and takes the least amount of time to execute.

The improved DFS assigns priority to the children according to a cost estimate based on their distance from the goal. Lesser the distance, the higher the priority of the child. Now, an extra parameter demands adjustments to survive the maze. The first approach includes checking for 'safe children'. The probability of a cell catching on fire at any given time is dependent on the number of neighbors on fire. The children with the fewest cells on fire can be considered safer, and hence can be prioritized on this basis. This extra parameter is added to the improved DFS search.



```
Time taken: 34.060322284698486
```

**Fig 20. Fire runner with minimum neighbors on fire**

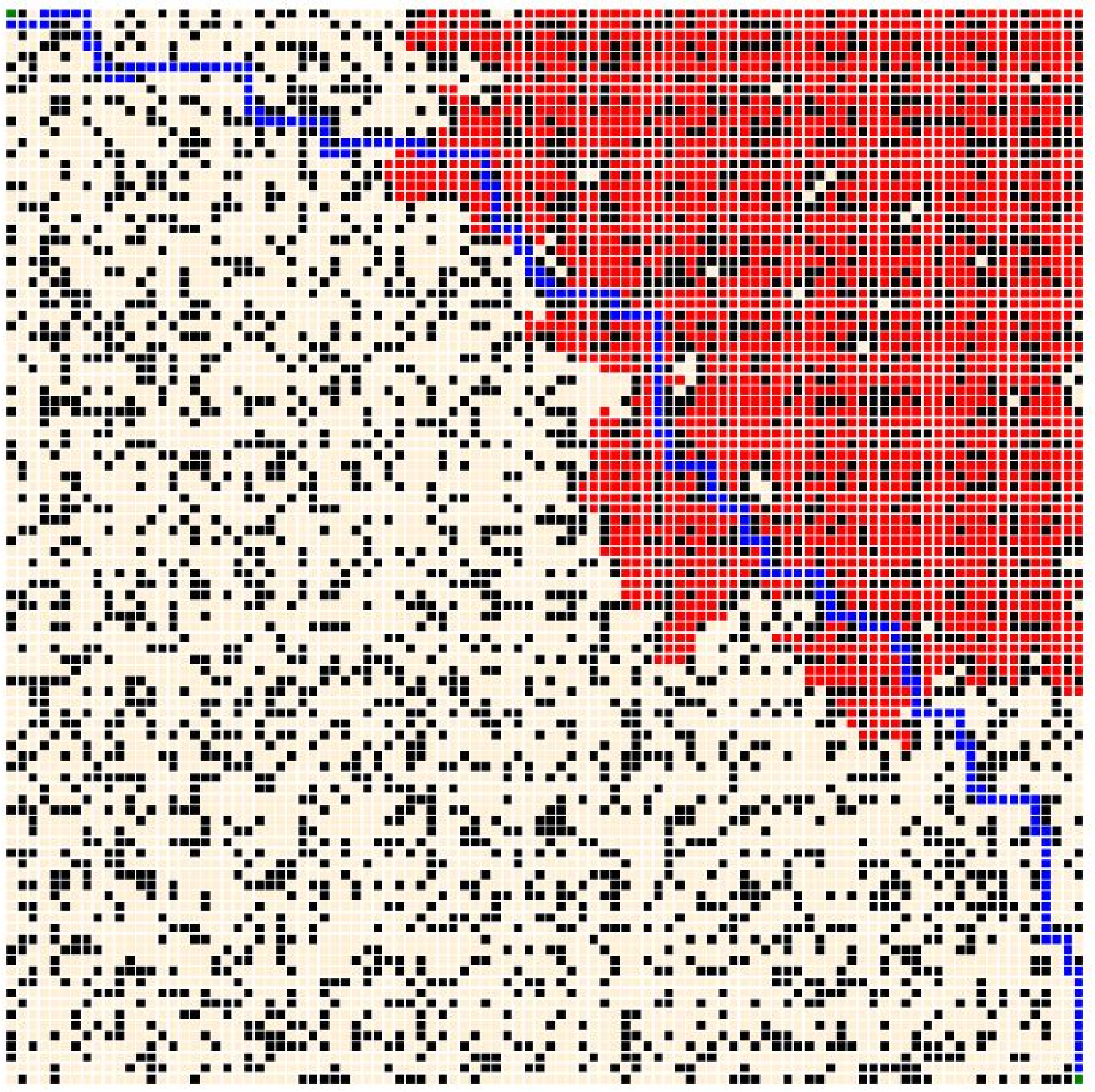
This algorithm provides a viable solution but is not optimal in terms of safety as well as time complexity. It is unsafe as it only checks for fire that is upto 2 cells away from the current state. Also, it takes more time as we check for children of children which are going to be encountered again. On an average, every node is checked at least twice.

A better solution would be to consider a heuristic of 'distance from nearest cell on fire'. The more the distance from the nearest cell on fire, more the priority given to that child. This algorithm works on the principle that we want to 'save children' by moving away from the fire rather than checking for 'safe children'.

Cost Estimate = Manhattan Distance from goal - Distance from nearest fire cell

Maximise
Minimise

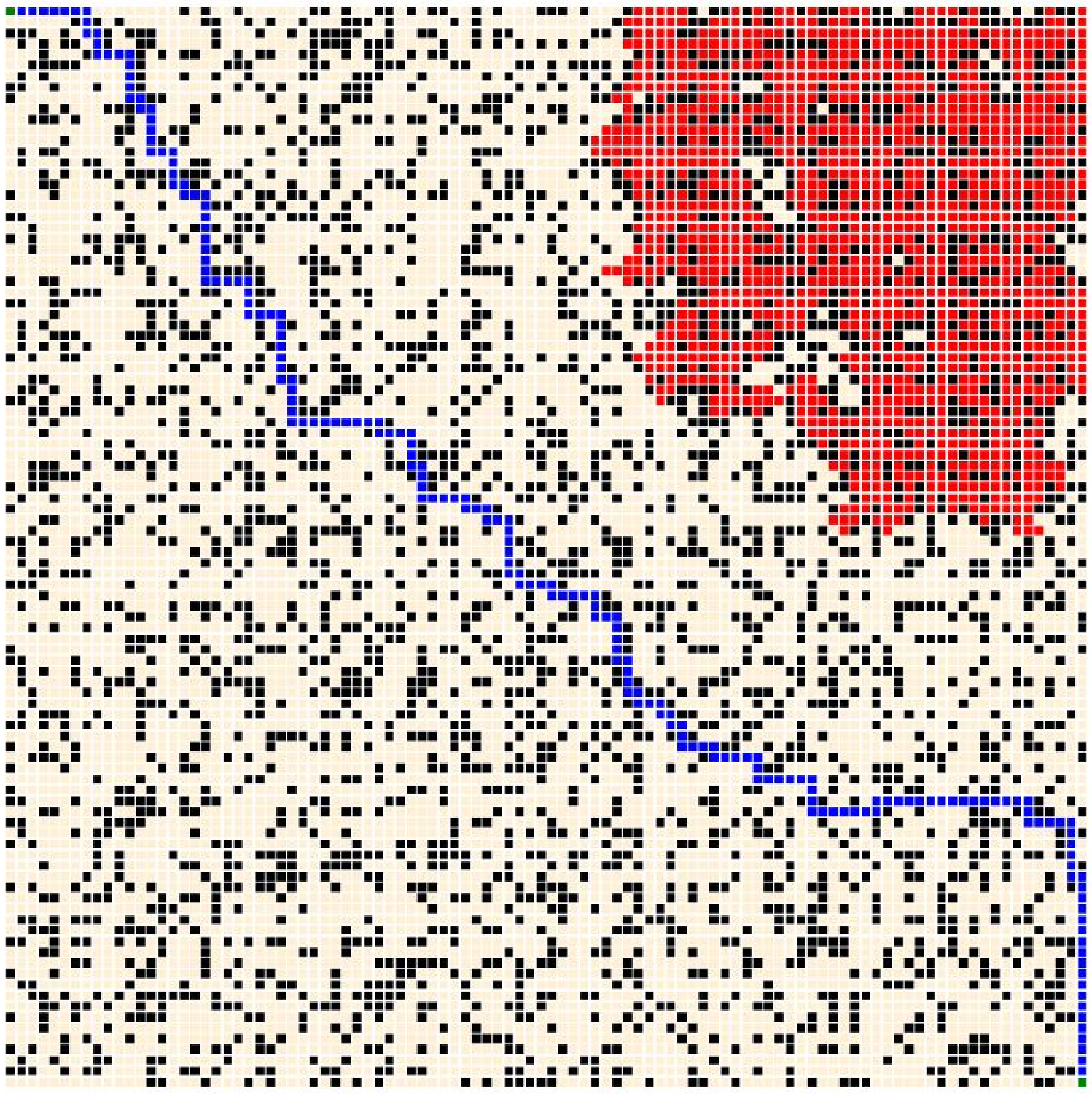
This approach uses a list of all the cells on fire and tries to find the one that is the nearest. The aim is to maximize this minimum distance.



```
Time taken: 9.857760667800903
```

**Fig 21. Fire runner with maximum distance from all cells on fire**

But as the fire spreads, each comparison to maximize the minimum distance from the fire becomes a bigger task and every following iteration takes exponentially more time. Can we think of a way to reduce the computation some more while still being away from the fire? One thing to consider is that cells that have been on fire for a long time may not be critical if we stay away from the cells that have recently caught fire. So, in this third approach we try to maximize the minimum distance from the new cells on fire.



```
Time taken: 2.6146209239959717
```

**Fig 22. Fire runner with maximum distance from new cells on fire**

Using three different approaches we have solved the problem of running through a maze on fire. In every consequent approach we have reduced the running time while still staying away from the fire.

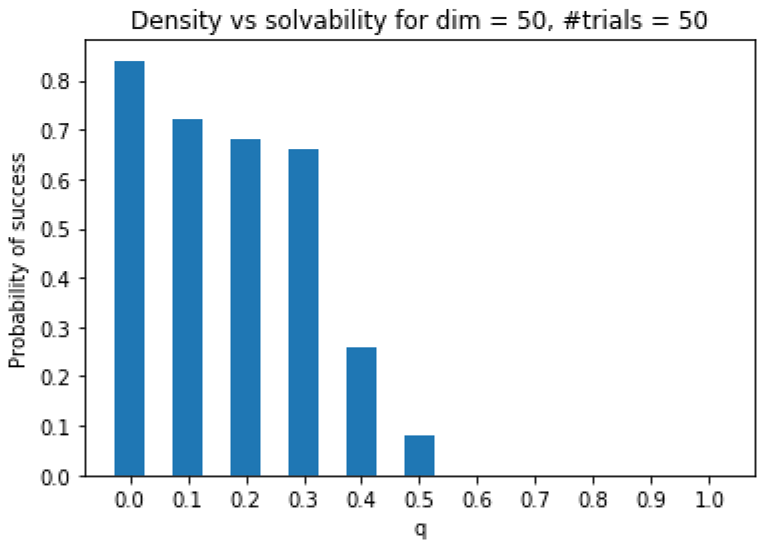


**Fig 19. Success rate of the latest algorithm with p = 0.3, dim = 50 v/s q**

*Do you think there are better strategies than yours? What would it take to do better?*

There are several better strategies than can be employed to get a better solution in this case. For instance, one of such strategies would be using reinforcement learning mechanisms to statistically determine a likely spread of the fire given the probability distribution and the structure of the maze. Markov-chain algorithm is one example of such a mechanism. However for such a large sample space, computing all this information would simply be too extensive and require a very large amount of computing resources. Another strategy that can be used would be an adversarial search assuming the fire to be a chance player. We can assign utility values based on how dangerous a certain spread of fire could be. Despite the spread of fire being an uncertain event, we can use the probability distribution to appropriately estimate the utility and prepare for the worst. But as discussed with reinforcement learning, the large sample space would demand very high computing resources to evaluate all the utility values for all possible paths.

**<u>Contributions by team members:</u>**

The discussion and analysis related to each part of the problem (question 2 analysis, choosing parameters and conditions for hard maze, optimal approach when maze is on fire) were purely group based wherein each of us provided inputs and justifications based on individual understanding.
After the discussion about the implementation strategies, the coding part of the algorithm was divided amongst us as follows:

| | |
|---|---|
| Keya Desai | Maze generation, A* (Euclidean and Manhattan), Maze Visualization, Statistical Analysis and Data Visualization, Fire Maze, Basic Approaches |
| Prakruti Joshi | BFS, Bi-directional BFS, Genetic Algorithm, Hill Climbing Algorithm, Fire Maze, Planning and Integration, Basic Approaches |
| Rushabh Bid | DFS, Improved DFS, Genetic Algorithm, Hill Climbing Algorithm, Fire Maze, Fire Runner, Debugging and Testing, Advanced approaches |

The documentation of the report is also an equally collaborative work based on the discussions and analysis.