Assignment 1
Introduction to Artificial Intelligence

BY

Kartik Rattan, 187001624
Mareesh Kumar Issar, 186000297

Date: September 26, 2019

1. Environments and Algorithms

a. Generating a Maze: We generate the maze as follows,

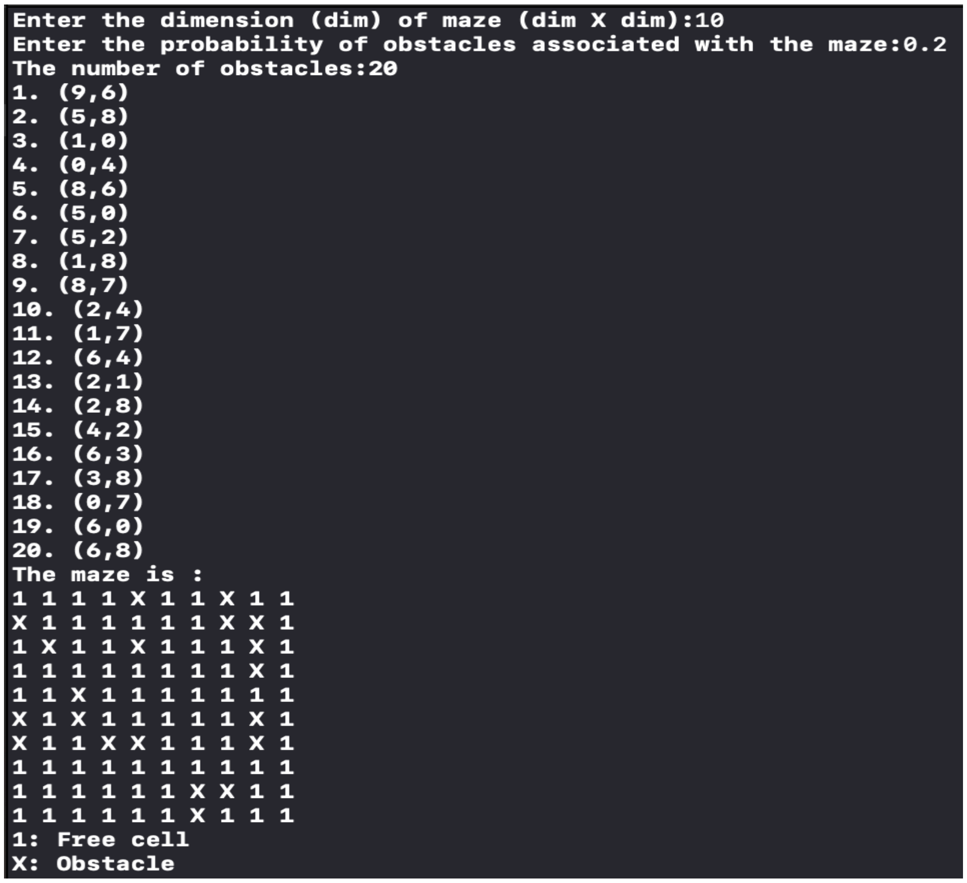
i. Create a maze object with parameters dimension and probability
ii. Generate a 2-D array with dimension X dimension and initialize each entry as 1, depicting there are no obstacles.
iii. For the maze with dimension **d** and probability **p**, generate the number of obstacles **n_o** using the following relation,

n_o = (d X d) X p

iv. Generate random points (y, x) in the 2D array and ensure that none of the points are repeated. Thus, we ensure that each maze with dimension, d and probability, p has exactly n_o obstacles.

For instance, for dimension = 100 & probability = 0.1, I generate 1000 obstacles. Then we randomly generate 1000 points and assign them a value of -1. We ensure that if the number of obstacles are 1000, we generate 1000 different points randomly using rand ().

For ease of understanding and depiction, the following figure shows the maze for dimension = 10 and probability 0.2, which leads to 20 obstacles depicted as X.



```
Enter the dimension (dim) of maze (dim X dim):10
Enter the probability of obstacles associated with the maze:0.2
The number of obstacles:20
1. (9,6)
2. (5,8)
3. (1,0)
4. (0,4)
5. (8,6)
6. (5,0)
7. (5,2)
8. (1,8)
9. (8,7)
10. (2,4)
11. (1,7)
12. (6,4)
13. (2,1)
14. (2,8)
15. (4,2)
16. (6,3)
17. (3,8)
18. (0,7)
19. (6,0)
20. (6,8)
The maze is :
1 1 1 1 X 1 1 X 1 1
X 1 1 1 1 1 1 X X 1
1 X 1 1 X 1 1 1 X 1
1 1 1 1 1 1 1 1 X 1
1 1 X 1 1 1 1 1 1 1
X 1 X 1 1 1 1 1 X 1
X 1 1 X X 1 1 1 X 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 X X 1 1
1 1 1 1 1 1 X 1 1 1
1: Free cell
X: Obstacle
```

Fig1. Maze depiction for dimension = 10 and probability = 0.2

2. Analysis and Comparison

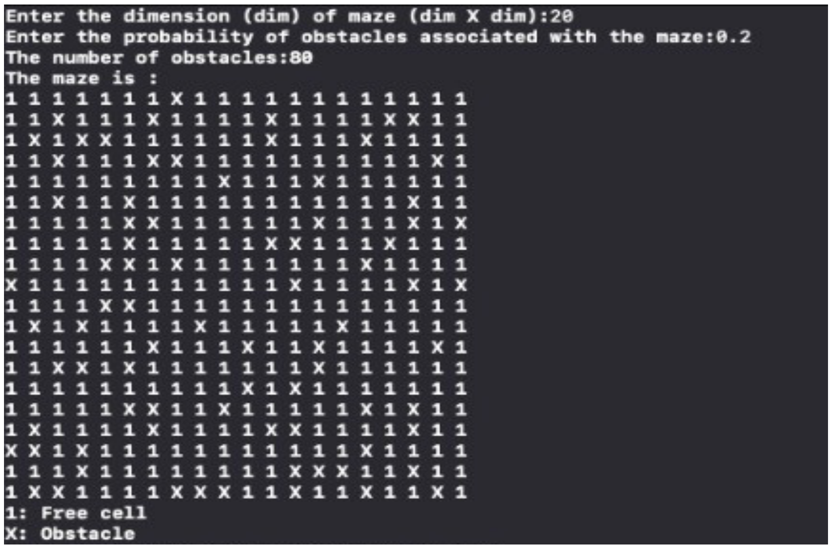
Ques 2.1. Find a map size (dim) that is large enough to produce maps that require some work to solve, but small enough that you can run each algorithm multiple times for a range of possible p values. How did you pick a dim?

Solution. After analyzing the four algorithms, we came to the conclusion that the bottleneck for our system to analyze the four algorithms would be both space (memory) and compute considerations. With respect to both space and compute, the memory space parameter toppled the other. Also, out of the four algorithms, we know that BFS has the maximum memory requirements as it exhausts each level before reaching the goal state. Thus, though in the end it provides us with a minimal solution, the memory expense of it bottlenecks our analysis for the dimension of the maze. Nonetheless, we run the four algorithms for different values of dim and probability, generation the following analysis.

Ques 2.2. 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.

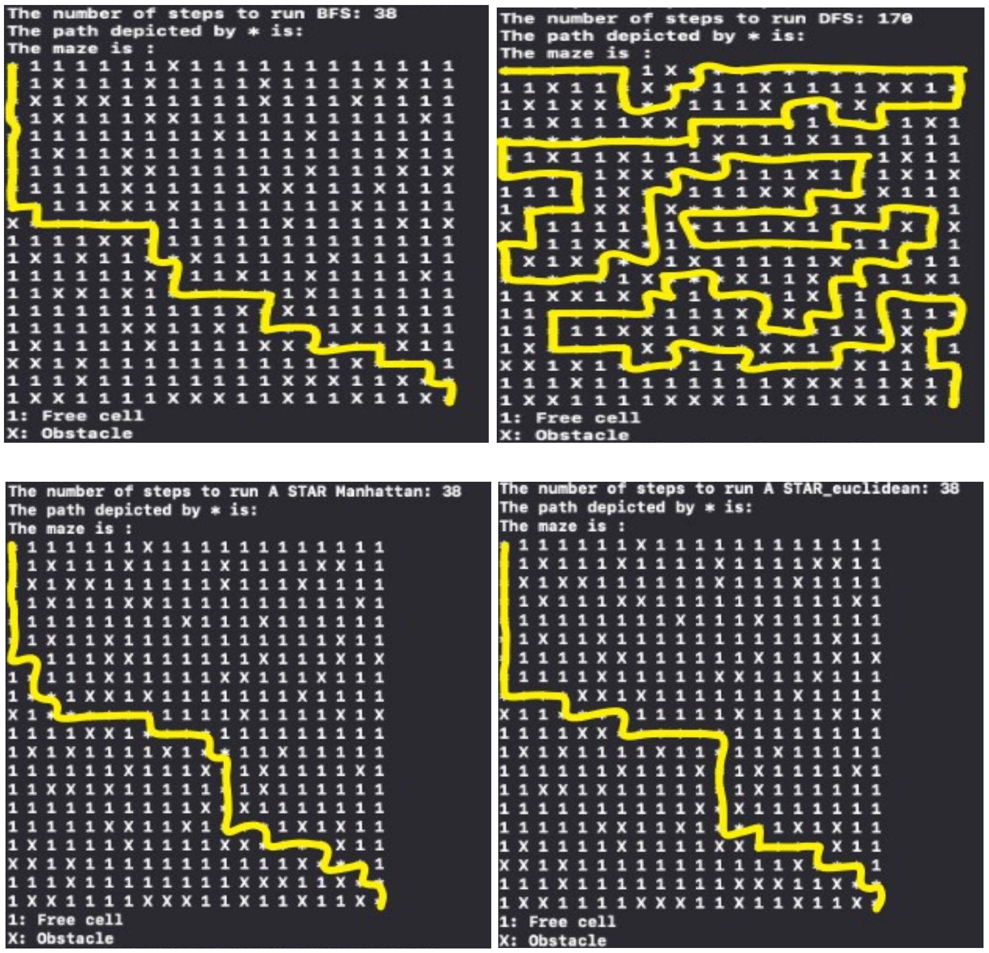
Solution. We ran the four algorithms for p = 0.2 with dimension = 20 to make better visualization. This leads to **20X20 matrix with 80 obstacles**. The following are the results for the analysis:
We represent the original matrix, and then the four algorithms in the following order,

1. BFS
2. DFS
3. A* (Manhattan)
4. A* (Euclidean)



```
Enter the dimension (dim) of maze (dim X dim):20
Enter the probability of obstacles associated with the maze:0.2
The number of obstacles:80
The maze is :
1 1 1 1 1 1 1 X 1 1 1 1 1 1 1 1 1 1 1 1
1 1 X 1 1 1 X 1 1 1 1 X 1 1 1 1 X X 1 1
1 X 1 X X 1 1 1 1 1 1 X 1 1 1 X 1 1 1 1
1 1 X 1 1 1 X X 1 1 1 1 1 1 1 1 1 1 X 1
1 1 1 1 1 1 1 1 1 X 1 1 1 X 1 1 1 1 1 1
1 1 X 1 1 X 1 1 1 1 1 1 1 1 1 1 1 X 1 1
1 1 1 1 1 X X 1 1 1 1 1 1 X 1 1 1 X 1 X
1 1 1 1 1 X 1 1 1 1 1 X X 1 1 1 X 1 1 1
1 1 1 1 X X 1 X 1 1 1 1 1 1 1 X 1 1 1 1
X 1 1 1 1 1 1 1 1 1 1 1 X 1 1 1 1 X 1 X
1 1 1 1 X X 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 X 1 X 1 1 1 1 X 1 1 1 1 1 X 1 1 1 1 1
1 1 1 1 1 1 X 1 1 1 X 1 1 X 1 1 1 1 X 1
1 1 X X 1 X 1 1 1 1 1 1 1 X 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 X 1 X 1 1 1 1 1 1 1
1 1 1 1 1 X X 1 1 X 1 1 1 1 1 X 1 X 1 1
1 X 1 1 1 1 X 1 1 1 1 X X 1 1 1 1 X 1 1
X X 1 X 1 1 1 1 1 1 1 1 1 1 1 X 1 1 1 1
1 1 1 X 1 1 1 1 1 1 1 1 X X X 1 1 X 1 1
1 X X 1 1 1 1 X X X 1 1 X 1 1 X 1 1 X 1
1: Free cell
X: Obstacle
```

Fig. The original matrix



```
The number of steps to run BFS: 38
The path depicted by * is:
The maze is :
* 1 1 1 1 1 1 X 1 1 1 1 1 1 1 1 1 1 1 1
* 1 X 1 1 1 X 1 1 1 1 X 1 1 1 1 X X 1 1
* X 1 X X 1 1 1 1 1 1 X 1 1 1 X 1 1 1 1
* 1 X 1 1 1 X X 1 1 1 1 1 1 1 1 1 1 X 1
* 1 1 1 1 1 1 1 1 X 1 1 1 X 1 1 1 1 1 1
* 1 X 1 1 X 1 1 1 1 1 1 1 1 1 1 1 X 1 1
* 1 1 1 1 X X 1 1 1 1 1 1 X 1 1 1 X 1 X
* 1 1 1 1 X 1 1 1 1 1 X X 1 1 1 X 1 1 1
* * 1 1 X X 1 X 1 1 1 1 1 1 1 X 1 1 1 1
X * * * * * * 1 1 1 1 1 X 1 1 1 1 X 1 X
1 1 1 1 X X * 1 1 1 1 1 1 1 1 1 1 1 1 1
1 X 1 X 1 1 * * X 1 1 1 1 1 X 1 1 1 1 1
1 1 1 1 1 1 X * 1 1 X 1 1 X 1 1 1 1 X 1
1 1 X X 1 X 1 * * * * * * 1 X 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 X * X 1 1 1 1 1 1 1
1 1 1 1 1 X X 1 1 X 1 * * * 1 X 1 X 1 1
1 X 1 1 1 1 X 1 1 1 1 X X * * * * X 1 1
X X 1 X 1 1 1 1 1 1 1 1 1 1 1 X * * * 1
1 1 1 X 1 1 1 1 1 1 1 1 X X X 1 1 X * *
1 X X 1 1 1 1 X X X 1 1 X 1 1 X 1 1 X *
1: Free cell
X: Obstacle
```

```
The number of steps to run DFS: 170
The path depicted by * is:
The maze is :
* * * * * * 1 X * * * * * * * * * * * *
1 1 X 1 1 * X * * 1 1 X 1 1 1 1 X X 1 *
1 X 1 X X * * * 1 1 1 X * * * X * * * *
1 1 X 1 1 1 X X * * * * * 1 * * * 1 X 1
* * * * * * * * * X 1 1 1 X 1 1 1 1 1 1
* 1 X 1 1 X 1 1 1 * * * * * * * 1 X 1 1
* * * * 1 X X * * * 1 1 1 X 1 * 1 X 1 X
1 1 1 * 1 X * * 1 1 1 X X * * * X 1 1 1
1 * * * X X * X * * * * * * 1 X * * * 1
X * 1 1 1 1 * 1 * 1 1 1 X 1 * * * X * X
* * 1 1 X X * 1 * * * * * * * 1 1 * * 1
* X 1 X * * * 1 X 1 1 1 1 1 X * * * 1 1
* * * * * 1 X * * * X 1 1 X * * 1 1 X 1
1 1 X X 1 X * * 1 * * * 1 X * 1 * * * *
1 1 * * * * * 1 1 1 X * X * * 1 * 1 1 *
1 1 * 1 1 X X 1 1 X 1 * * * 1 X * X * *
1 X * * * * X * * * * X X 1 * * * X * 1
X X 1 X 1 * * * 1 1 * * * * * X 1 1 * *
1 1 1 X 1 1 1 1 1 1 1 1 X X X 1 1 X 1 *
1 X X 1 1 1 1 X X X 1 1 X 1 1 X 1 1 X *
1: Free cell
X: Obstacle
```

```
The number of steps to run A STAR Manhattan: 38
The path depicted by * is:
The maze is :
* 1 1 1 1 1 1 X 1 1 1 1 1 1 1 1 1 1 1 1
* 1 X 1 1 1 X 1 1 1 1 X 1 1 1 1 X X 1 1
* X 1 X X 1 1 1 1 1 1 X 1 1 1 X 1 1 1 1
* 1 X 1 1 1 X X 1 1 1 1 1 1 1 1 1 1 X 1
* 1 1 1 1 1 1 1 1 X 1 1 1 X 1 1 1 1 1 1
* 1 X 1 1 X 1 1 1 1 1 1 1 1 1 1 1 X 1 1
* * 1 1 1 X X 1 1 1 1 1 1 X 1 1 1 X 1 X
1 * 1 1 1 X 1 1 1 1 1 X X 1 1 1 X 1 1 1
1 * * 1 X X 1 X 1 1 1 1 1 1 1 X 1 1 1 1
X 1 * * * * * * 1 1 1 1 X 1 1 1 1 X 1 X
1 1 1 1 X X 1 * * * * 1 1 1 1 1 1 1 1 1
1 X 1 X 1 1 1 1 X 1 * * 1 1 X 1 1 1 1 1
1 1 1 1 1 1 X 1 1 1 X * 1 X 1 1 1 1 X 1
1 1 X X 1 X 1 1 1 1 1 * 1 X 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 X * X 1 1 1 1 1 1 1
1 1 1 1 1 X X 1 1 X 1 * * * 1 X 1 X 1 1
1 X 1 1 1 1 X 1 1 1 1 X X * * * * X 1 1
X X 1 X 1 1 1 1 1 1 1 1 1 1 1 X * * * 1
1 1 1 X 1 1 1 1 1 1 1 1 X X X 1 1 X * *
1 X X 1 1 1 1 X X X 1 1 X 1 1 X 1 1 X *
1: Free cell
X: Obstacle
```

```
The number of steps to run A STAR_euclidean: 38
The path depicted by * is:
The maze is :
* 1 1 1 1 1 1 X 1 1 1 1 1 1 1 1 1 1 1 1
* 1 X 1 1 1 X 1 1 1 1 X 1 1 1 1 X X 1 1
* X 1 X X 1 1 1 1 1 1 X 1 1 1 X 1 1 1 1
* 1 X 1 1 1 X X 1 1 1 1 1 1 1 1 1 1 X 1
* 1 1 1 1 1 1 1 1 X 1 1 1 X 1 1 1 1 1 1
* 1 X 1 1 X 1 1 1 1 1 1 1 1 1 1 1 X 1 1
* 1 1 1 1 X X 1 1 1 1 1 1 X 1 1 1 X 1 X
* 1 1 1 1 X 1 1 1 1 1 X X 1 1 1 X 1 1 1
* * * * X X 1 X 1 1 1 1 1 1 1 X 1 1 1 1
X 1 1 * * * * 1 1 1 1 1 X 1 1 1 1 X 1 X
1 1 1 1 X X * * * * * * 1 1 1 1 1 1 1 1
1 X 1 X 1 1 1 1 X 1 1 * 1 1 X 1 1 1 1 1
1 1 1 1 1 1 X 1 1 1 X * 1 X 1 1 1 1 X 1
1 1 X X 1 X 1 1 1 1 1 * 1 X 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 X * X 1 1 1 1 1 1 1
1 1 1 1 1 X X 1 1 X 1 * * * 1 X 1 X 1 1
1 X 1 1 1 1 X 1 1 1 1 X X * * * * X 1 1
X X 1 X 1 1 1 1 1 1 1 1 1 1 1 X * * * 1
1 1 1 X 1 1 1 1 1 1 1 1 X X X 1 1 X * *
1 X X 1 1 1 1 X X X 1 1 X 1 1 X 1 1 X *
1: Free cell
X: Obstacle
```

Fig. from top left (clockwise) BFS, DFS, A* manhattan and A* euclidean for dim = 20 and p = 0.2

Running BFS, and both A* algorithms produce the minimum distance of 38 steps, though as we can see the path taken by all the three algorithms is different. For BFS, it starts from the left side and moves along the boundary looking for next neighbor until it finds an obstacle. It then turns central, avoiding obstacles. Similarly, A* starts to move more centrally early in the maze.

In between A* Euclidean and Manhattan, we observe that Euclidean algorithm is much more central tries to position itself always towards the diagonal from the current step to the goal. A* Manhattan has a similar tendency, but the square term in Euclidean enhances this characteristic. A*using Manhattan distance tends to give us the shortest path by travelling along the edges of the maze whereas using the Euclidean distance as the heuristic the algorithm tends to find the shortest path by moving diagonally from start towards the goal.

DFS on the other hand roams around as much as possible in search of the goal. It has no heuristic guiding it, nor has an aim to look for the shortest path. For DFS, the only aim at the start of the maze is to somehow reach the goal state, and it works exactly like that. Just reach the goal state by hook or crook, irrespective on any effort to optimize its path!

Ques. 2.3. 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.

Solution. We start our analysis to search for the threshold probability, p0 by selecting the dimension= 100, thus a Maze of 100X100. In our opinion, since p0 is a ratio on the dimension, a certain dimension d with probability p0 will represent higher dimensions as well. For instance, dimension = 10 with probability 0.1 means 10X10 maze with 10 obstacles. Similarly, a dimension = 100 with 0.1 has 1000 obstacles. The ratio remains the same. Thus, a p0 for a certain d will represent another d as well.

We run A* Manhattan algorithm 20 times with probability in the range from 0.1 to 0.4 with the steps of 0.05 with an exception of 0.28 since there was a sudden change in the solvability (terminating at 0.4 because it was consistently giving us 0 solvability). We calculate the solvability as:

$$\text{Solvability, } S = n_{true} / n_{total}$$

Where,
n_{true}: The number of times a maze is solvable (returns a path) and
n_{total}: The number of different mazes with same parameters analyzed

The following figure represent our analysis with n_{total} = 20:

Probability	Solvability %	Avg time to run the algorithm once
0.10	98%	4 seconds
0.15	95%	4 seconds
0.20	95%	3.8 seconds
0.25	50%	4 seconds
0.28	30%	4 seconds
0.30	~5%	4 seconds
0.35	~5%	4 seconds
0.40	0%	-

For our analysis, we infer that a probability, p0 of [0.25-0.28] can be considered as a workable range where the mazes are both solvable as well as interesting. Though the solvability is not high (<=50%), it is sharp increase from 5% for p >= 0.3. A probability of 0.4, that is 4000 obstacles for 100X100 repeatedly produce a solvability of 0%.

Ques. 2.4. For p in [0, p0] as above, estimate the average or expected 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?

Solution. For the above analysis, we use breadth first algorithm. Though the memory footprint of BFS is large as compared to other algorithms, it ensures that whatever path it reproduces is the shortest. Though for smaller dimensions and fewer obstacles, both the A* algorithms might give you the same shortest path, but they won't guarantee this. But with BFS and its implementation to study each level before reaching the goal, it maintains that the path returned is the shortest.

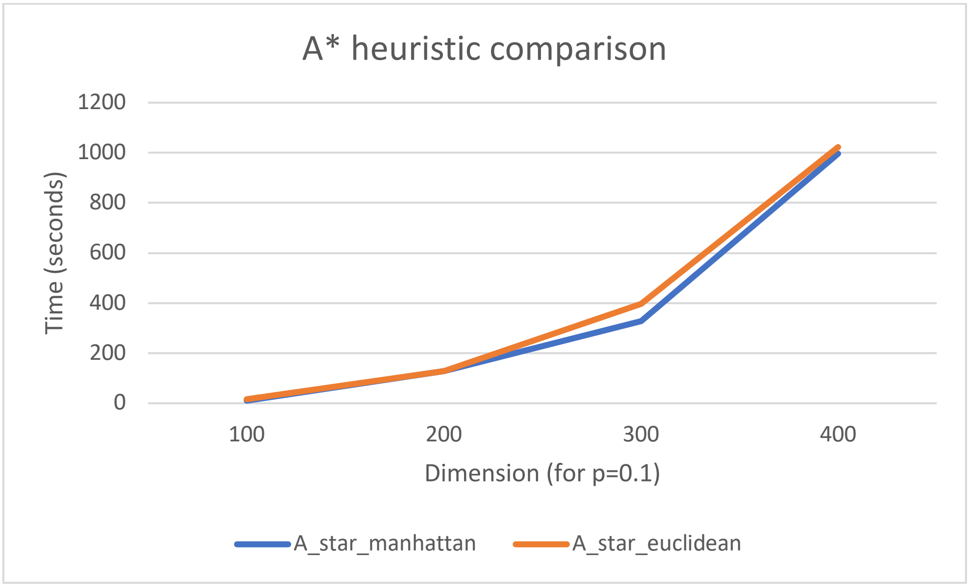
The following tables shows the analysis for BFS with dimension 100X100 and p = [0.1, 0.3]. We run the BFS algorithm 20 times for each probability to find the avg shortest path length.

Dimension	Probability	Obstacles	Shortest Path Length
100 X 100	0.10	1000	198
100 X 100	0.15	1500	198
100 X 100	0.20	2000	198
100 X 100	0.25	2500	200
100 X 100	0.28	2800	200
100 X 100	0.30	3000	202
100 X 100	0.40	4000	0 (no path)

Ques. 2.5. Is one heuristic uniformly better than the other for running A∗? How can they be compared? Plot the relevant data and justify your conclusions.

Solution. A*using Manhattan distance heuristic tends to give us the shortest path by travelling along the edges of the maze whereas using the Euclidean distance as the heuristic the algorithm tends to find the shortest path by moving diagonally from start towards the goal.

For higher dimensional maze , the A* using Manhattan heuristic should run faster than the A* Euclidean because the as the maze gets larger there is more chances of obstacles randomly placed in the middle rather than at the edge thus Manhattan heuristic should provide us a path faster than the Euclidean heuristic.



Ques. 2.6. Do these algorithms behave as they should?

Solution. Yes, these algorithms behave as they should.

BFS: Uses queue as the data structure for storing the fringe. It processes the nodes hop by hop i.e. it processes all the one hop neighbors then the two hop and so on. It has exponential space complexity.
DFS: Uses stack as a data structure for storing the fringe. It processes the graph branch wise i.e. going as deep as possible in one branch then repeating it for the other branches. It has a linear space complexity.
A* with Manhattan heuristic: uses priority queue as the data structure for storing the fringe. It tends to find the shortest path by moving along the sides of the maze as much as possible.
A* with Euclidean heuristic: also uses priority queue as the data structure for storing the fringe. It tends to find the shortest path by moving through the center of the maze(diagonally) as much as possible.
BD-BFS: Uses queue as the underlying data structure to represent the fringe. It processes the nodes by using BFS from the start as well as from the end, each (BFS) processing their all one hop neighbors (in one step of the iteration), then all two hop neighbors and so on. BD-BFS produces the same results as BFS, and uses less memory per fringe w.r.t BFS.

Ques 2.7. 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.

	4	
3	**Current state**	2
	1	

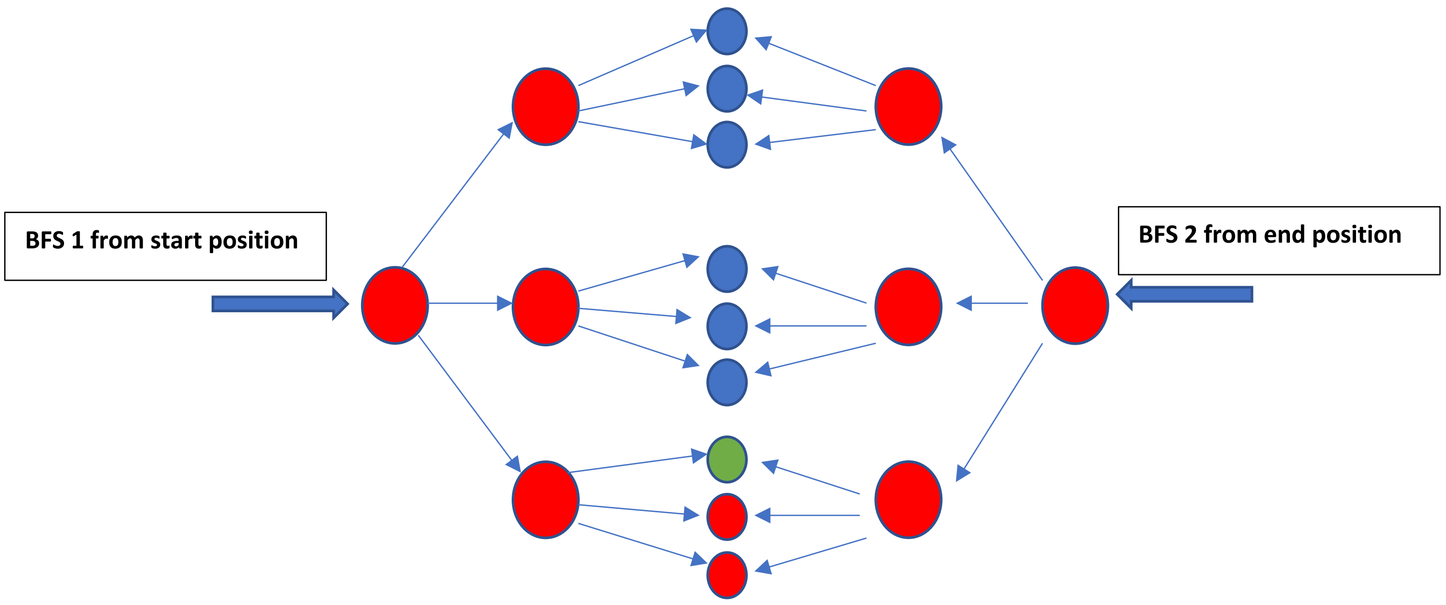
Solution. The above figure considers the start state to be at the top left and the goal state to be at the bottom right.
Consider the case when we are at the current state and we have to decide what neighbors to load on the stack (fringe). We should prefer putting state 1 at the top of the fringe because it brings us closer to our goal. Then next in preference (if state 1 is not free) should be state 2 because it brings us closer to the goal in x direction. If state 2 is not free, then the next state in preference should be state 3 as it is on the same level as the current state. Lastly, if no option is available then we should put state 4 at the top of the stack.

Thus, in summary the neighbor's worth looking (that should be put at the top of the stack) could be those which are closer to the goal from the start state in terms of either their x-coordinate or y-coordinate.

Ques. 2.8. On the same map, are there ever nodes that BD-BFS expands that A∗ doesn't? Why or why not? Give an example and justify.

Solution. Yes, in our opinion, the number of nodes expanded by BD-BFS is higher than A* algorithm. Let's begin our discussion with analyzing BFS first. It begins its search at the start point till the goal point, expanding each and every node on a given level iteratively until it reaches the goal state. Thus, if the goal state is at level 15 from the start state, BFS will expand all the nodes till level 14th in search for the goal state. This does not happen with A* algorithm. It expands only those nodes that pushes it towards the goal state. It's like a sign board on a highway, the heuristics guide A* towards the goal state and instructs it to avoid looking at nodes that takes it far away from the goal state. Now the heuristics might be bad and not give you a correct shortest path, but it will always ensure that you do not explore the nodes that are not worth looking (considering descent heuristics).

Now, with BD-BFS, though it is better than BFS in terms of memory allocation for an individual fringe (there are two fringe but in asymptotic growth rate, we ignore constants). But it still explores all the nodes from both start and goal level until they reach a common level. We depict the relation in the following manner:



Unlike BFS, BD-BFS will explore nodes until you reach a common node (green) from both the sides. It will avoid exploring the blue nodes unlike BFS. Thus, its faster than BFS. But A* will search only those algorithms that brings it closer to the goal. Hence, it will explore less nodes than BD-BFS.

Bonus Question: How does the threshold probability p0 depend on dim? Be as precise as you can.
Solution. p0 is defined as the threshold after which most mazes are not solvable. It is given by the ratio of the number of obstacles/total number of cells. Using p0 we calculate the number of obstacles as dim*dim*p0. Since p0 is a ratio we except it to remain the same irrespective of the value of dim but it may vary slightly with dim due to rounding error from the equation generating the obstacles (as we take the floor of the number of obstacles).

3. Generating Hard Mazes

Ques. 3.1. What local search algorithm did you pick, and why? How are you representing the maze/environment to be able to utilize this search algorithm? What design choices did you have to make to make to apply this search algorithm to this problem?
Solution. We have implemented and analyzed the following local search algorithms:

1. Hill Climbing Algorithm
2. Simulated Annealing Algorithm
3. Genetic Algorithm

The maze environment is converted into an 2D vector matrix (dim*dim) filled with 0's (blocked cell) and 1's (open cell). The 2D vector matrix is part of a maze class which has the maze, its associated functions as well as variables for storing the values associated to the metrics (length of shortest path, total number of nodes expanded and maximum fringe size). We also have a point class for storing the co-ordinate points of the maze. The number of blocked cells is given by (dim*dim*p). While generating the maze we have taken care that the start (0,0) and goal (dim-1, dim-1) states are never blocked also, we ensure that once a cell is blocked, it cannot be blocked again.

Hill Climbing Algorithm

The input to this algorithm is a solvable maze with dimension (dim) and probability of obstacle (p) selected by user.

From this initial maze we construct the neighbor mazes by adding additional one and two obstacles. For adding one obstacle there are (dim * dim – number of obstacles – 2{start and goal states}) locations and for adding two obstacles there are half as many as those available for adding one obstacle.

After generation of each neighbor maze we check whether the neighbor maze is solvable or not. If it is solvable, we calculate the length of the shortest path for that maze and using this value as a priority we store the maze in a priority queue.

Once all the neighbor mazes are generated and stored in the priority queue then we select the best neighbor (top of the priority queue) to be the initial state if its priority value is greater than our current initial state. This process is repeated until a termination condition is encountered.

Simulated Annealing Algorithm

The input to this algorithm is a solvable maze with dimension (dim) and probability of obstacle (p) selected by user.

From this initial maze we construct the neighbor mazes by adding as well as removing additional one and two obstacles. For adding one obstacle there are (dim * dim – number of obstacles – 2{start and goal states}) locations and for adding two obstacles there are half as many as those available for adding one obstacle.

After generation of each neighbor maze we check whether the neighbor maze is solvable or not. If it is solvable, we calculate the length of the shortest path for that maze and using this value as a priority we store the maze in a priority queue.

Once all the neighbor mazes are generated and stored in the priority queue then we select the best neighbor (top of the priority queue) to be the initial state if its priority value is greater than our current initial state. Otherwise we compare the output of our acceptance probability function with a random number generated between 0 to 1 and accept a neighbor with lower probability than the initial state (to become the initial state for the next iteration).

This process is repeated until a termination condition is encountered.

Genetic Algorithm

The input to this algorithm is a solvable maze with dimension (dim) and probability of obstacle (p) selected by user.

From this initial maze we construct the (N=10) children by adding additional one obstacle to the initial state at a random open cell.

After generation of each child maze we check whether the child maze is solvable or not. If it is solvable we calculate the length of the shortest path for that maze and using this value as a priority we store the maze in a priority queue of children maze as well as a priority queue of children and parents maze.

Then we create the next generation (two children) by taking two child matrices at a time from the children maze priority queue. We take 60% of the first matrix ((0.6*dim) columns) and add it to 40% of the second matrix and thus get two children. After this we introduce mutation into the new children matrices by adding an extra obstacle in a random open cell. After generation of the mutated children maze, we check whether they are solvable or not. If they are solvable, we add then to the children and parent maze. Then, we select the best children from the 2N children and parent maze and set them as the children maze for the next iteration. Genetic algorithm has the best chance for finding the hardest maze amongst the algorithms that we have implemented.

Ques. 3.2. Unlike the problem of solving the maze, for which the `goal' is well-defined, it is difficult to know if you have constructed the `hardest' maze. What kind of termination conditions can you apply here to generate hard if not the hardest maze? What kind of shortcomings or advantages do you anticipate from your approach?

Solution. Termination conditions for Hill Climbing algorithm:

1. When the maximum length of the shortest path has repeated itself for 20 times successively.

Termination conditions for Simulated Annealing algorithm:

1. When the maximum length of the shortest path has repeated itself for 20 times successively.
2. After certain predefined number of iterations which represents after certain time.

Termination conditions for Genetic algorithm:

1. When the maximum length of the shortest path has repeated itself for 20 times successively.
2. After certain predefined number of iterations which is analogous to number of generations.
3. No solvable children are generated.

One of our termination conditions is being able to control the number of iterations, this helps us in saving time and computing resources while running our algorithm for a large maze.

Shortcomings that we anticipate from my approach is that for example in hill climbing we might wrongly consider plateau (repeated value for a long time) as a local maximum. After a certain number of iterations, we started seeing the same value being repeated for hill climbing as we were only making neighbors by adding obstacles. Whereas this was not the case in simulated annealing where we added (+1 and +2) and also removed (-1 and -2) obstacles which resulted in larger variations of the output results.

Another major shortcoming of our approach is that we may not be able to find the hardest maze because combining parents in genetic algorithm does not guarantee the children to be harder. Due to this the output from different generations does not change much. Also, there is no significant variations in the results from the children as there are lesser number of obstacles present initially in their parents as we increase only one obstacle at the time of mutation.

Hill climbing and simulated annealing algorithm take a longer time to run for larger mazes as compared to genetic because they generate all the possible neighbors with increasing number of obstacles (+1 and +2).

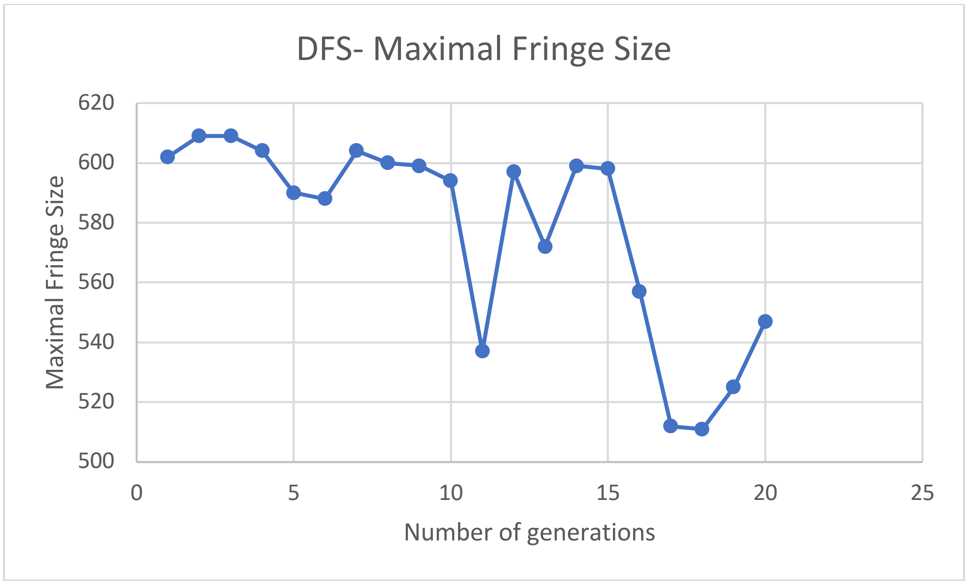
Ques. 3.3. Try to find the hardest mazes for the following algorithms using the paired metric:
DFS with Maximal Fringe Size

A* Manhattan with Maximal Nodes Expanded
DFS with maximal fringe size

Solution. Maximal fringe size computed at return from the function (DFS) call.
Taking dim=50 and p=0.2 with number of generations=20 and Number of children(N)= 10

We get the following results:

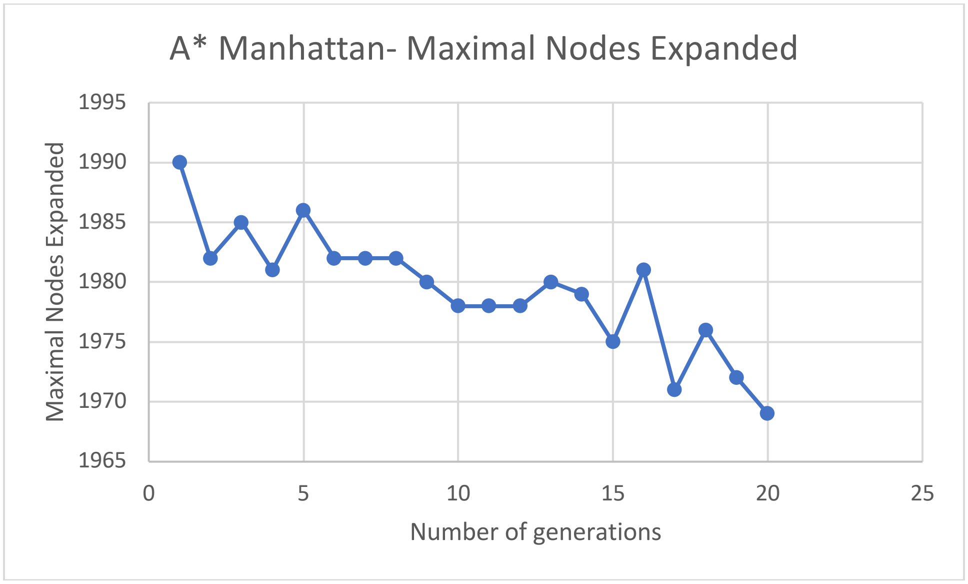
Generation	Maximal Fringe Size
1	602
2	609
3	609
4	604
5	590
6	588
7	604
8	600
9	599
10	594
11	537
12	597
13	572
14	599
15	598
16	557
17	512
18	511
19	525
20	547



The maximum value amongst the results is: 609
A* Manhattan with maximal nodes expanded
Maximal nodes expanded computed at return from the function(A_star) call.
Taking dim=50 and p=0.2 with number of generations=20 and Number of children(N)= 10

We get the following results:

Generation	Maximal Fringe Size
1	1990
2	1982
3	1985
4	1981
5	1986
6	1982
7	1982
8	1982
9	1980
10	1978
11	1978
12	1978
13	1980
14	1979
15	1975
16	1981
17	1971
18	1976
19	1972
20	1969



The maximum value amongst the results is: 1990

Ques. 3.4. Do your results agree with your intuition?

Solution. As the maze gets harder the solution path will tend to become wigglier (passing through the maze). For DFS with Maximal Fringe Size the fringe size at the end of the DFS operation should decrease as with harder maze there will be more obstacles which leads to more points being popped from the fringe. For A*Manhattan with Maximal Nodes Expanded at the end of A* operation should increase because A*Manhattan travels along the edge of the matrix and as the maze becomes harder and harder, the algorithm has to expand more as it won't be able to find a path along the side of the mazes.

4. What if the Maze Were on Fire?

a. **Generating a fire Maze: We generate the maze as follows,**

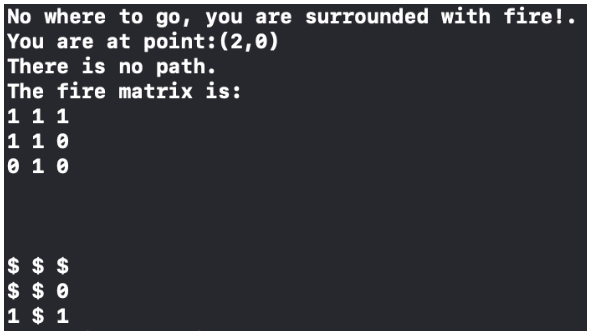
i. Create a maze object with parameters dimension and probability
ii. For the maze, set the point [0, dim-1] as a beginning fire point.
iii. If a free cell has no burning neighbors, it will still be free in the next time step.
iv. If a cell is on fire, it will still be on fire in the next time step.
v. If a free cell has k burning neighbors, it will be on fire in the next time step with probability, p

$$p = 1 - (1 - q)^k$$

Now for the fire to spread, the value of q should be greater than 0.5. This is because the first point on fire is [0, dim-1]. Each of its neighbors have exactly 1 node on fire to begin with. Thus, for all the neighbors to the [0, dim-1], we have a probability of $p = 1 - (1-q)^1$. Thus, for values of $q < 0.5$, $p < 0.5$. Thus, to spread the fire, the probability of it getting on fire should be greater than 0.5 and it would be possible only if $q > 0.5$.

Also, to get a valid maze (in which the fire is not surrounded with obstacles and thus not spread), we iterate through different mazes until we get a maze which has a path from [0, dim-1] to [dim-1, dim-1] as well as from start to goal. This ensures that person can move as well as the fire can spread.

The agent will face the following situations:
1. There are obstacles that makes it unable to reach the goal irrespective of the fire.
2. At a particular instance, the agent has no neighboring cells that are free. He is either surrounded with fire and/or obstacles.



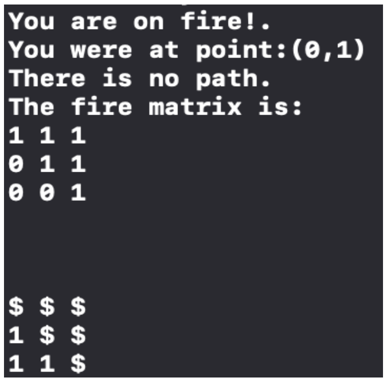
```
No where to go, you are surrounded with fire!.
You are at point:(2,0)
There is no path.
The fire matrix is:
1 1 1
1 1 0
0 1 0



$ $ $
$ $ 0
1 $ 1
```

$ - represents fire
0 – Obstacles

1 – Free space
3. Since we update the maze after the person moves one step, his current position can catch fire after we update the maze for fire. Thus, his current cell will catch fire and the agent will die.

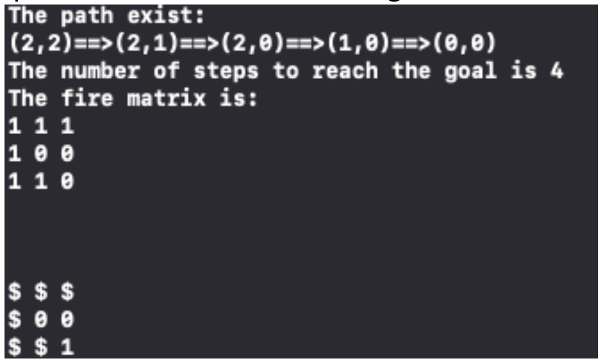


```
You are on fire!.
You were at point:(0,1)
There is no path.
The fire matrix is:
1 1 1
0 1 1
0 0 1



$ $ $
1 $ $
1 1 $
```

4. You reach the goal state by dodging the fire. It is possible that you reach the goal state and the path you followed caught fire, but you passed it and are now in the goal state.



```
The path exist:
(2,2)==>(2,1)==>(2,0)==>(1,0)==>(0,0)
The number of steps to reach the goal is 4
The fire matrix is:
1 1 1
1 0 0
1 1 0



$ $ $
$ 0 0
$ $ 1
```

Ques. 4.1. How can you solve the problem (to move from upper left to lower right, as before) in this situation? As a baseline, consider the following strategy: simply find the shortest path in the initial maze, and run it; if you die you die. At density p0 as in Section 2, generating mazes with paths from upper left to lower right and upper right to lower left, simulate this strategy and plot, as a function of q, the average success rate of this strategy.

Solution. In the first part, according to the baseline solution, we run the algorithm (A* Manhattan) for the shortest path. The path-planning to reach the goal is unintelligible and unassisted with respect to fire/fire spreading. That is, the agent travels along the shortest path with respect to the obstacles and completely disregard the fire. Thus, in this case it does not adapt itself to the spreading of the fire at each step. It is a stubborn approach and is focused on reaching the goal as early as possible and does not care if it gets itself on fire or moves to a node that catches fire.

We run the algorithm for 100 X 100 with probability p0 = 0.25. The value of q is (>0.5) and we iterate the value till q =1. As we have mentioned above, the fire will not spread if the value of q is < 0.5.

We run the algorithm (A* Manhattan) 20 times for each value and present the following analysis:

Probability, q	**Solvability**
<0.5	100%
>=0.5	20%

We use A* Manhattan for the shortest path, and make sure that we move across the left edge to reach the goal state (check the left edge neighbors first).

For q<0.5, the probability for fire to spread will always be zero because we spread fire only if the prob, p_fire = 1-(1-q)^K > 0.5. But since q < 0.5, p_fire will never be greater than 0.5 (as we begin with just one point right top most corner on fire, thus K =1), and hence fire never spreads. Thus, solvability for q < 0.5 is always 100%.

Now for q>=0.5, if the path remains the same throughout (shortest path) with lower dimensions (for instance 100 X 100), the fire will catch up with the agent almost all the times. We can observe that mathematically. The person moves only one step at a time. But the fire can spread from [0,4] for each point. And since q>0.5 and for k=1, p_fire >= 0.5, the fire will spread rapidly. And for subsequent values of k = 2,3, & 4, the respective values are 0.75, 0.875, & 0.9375. Thus, in all the three cases the fire will spread rapidly, unless there is an obstruction and that does not allow the fire to spread. This is the reason that we saw 20% solvability for dimension 100 X 100, instead of 0%.

We observed, as we increase the dimension (1000 X 1000), the fire spreads as above but reaches the agent on the left side after considerable time. Thus, the agent reaches the goal (30-40% solvability, low value because we are also using p0, the hardest maze) because it moves along left and the bottom edge paths (if not obstructed) to reach the goal. The fire never catches up with it (since obstructions also does not allow it to spread rapidly) and thus we see high solvability than before.

Ques. 4.2. 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.

Solution. Taking hint from above, we see that we have to expand our conception of the search space. For this, we use one of the A* algorithms but use a different strategy.

We use the time space as our strategy. The agent moves one step ahead in time, and so does the fire. We look at t+1 time state iteratively and look for a better path. We are taking snapshots of current position of both the agent and the fire, and then deciding a path from the start to the goal.

The agent begins at the starting point and the fire begins on the right topmost point. Now, the agent moves one step which is followed by the one iteration of spreading of fire. Now, the agent looks for points for its neighbors and checks whether the points are on fire or not, in addition to looking for obstructions. Thus, this is an intelligent path planning unlike the above baseline case. The agent does not move on just one path. The path keeps on changing. The agent analyzes all the neighbors at a given point before movement.

This approach is also made assisted as we look for neighbors of neighbors for fire. If the neighbor point is surrounded by fire, we increase the heuristic (this puts them high on the min_priority_queue) for the point so that the path senses danger and does not travel towards fire. Also, the points with fire given a very high heuristic. Hence, the current state never reaches a point with fire or surrounded with fire. But the fire can reach a current state.

We still see low solvability for dimension 100X100 and p0 = 0.25. Again, for q < 0.5, the fire never spreads, and the solvability is 100%. But we see an increase for q > 0.5 in this case as follows:

Probability, q	**Solvability**
<0.5	100%
0.55	40%
0.6	30%
0.7	30%
>0.7	0%

Again, though the fire spreads rapidly here as above, the agent can make some intelligent decisions to not move towards the fire. Thus, the increase in solvability. The solvability is again not as high as possible because fire for q > 0.5 spreads rapidly and catches up with the agent nonetheless for 100X100. For 1000X1000, we see some improved results. But the increase is only for 45% for q = 0.55. Also, most of the positive cases are because the maze is the hardest and obstructions do not allow the fire to spread. Thus, it reaches the goal.