CS 520: Assignment 1

Maze Runner

Bharath Reddy

brp97

Bharathreddy1179@gmail.com

Ishaan Singh

is393

ishaansingh23@gmail.com

1. Environments and Algorithms

1.1 Generating Environments

→ We made a RandomMazeGen() function which will take the value of dimensions of maze and return dim x dim array using probability of 0.2, 0.3, 0.4 (which is the probability of a wall being present at any point in the maze). Starting index of the maze is (0,0) with the value 0 and the end index with the value 6. We denote the empty index by '0' and the index which has wall by '1'. In our maze we fill the index with 1 and 0 with random probability. This can be applied to maze of "N" dimensions, as we have initiated a user input for the size of the maze as shown in the below code snippet at line 405.

```
405        mazeDim = int(input("Enter the dimensions of the maze: "))
406
407        #calling the maze call to generate the new maze
408        maze = Maze(mazeDim)
409
410        exitApp = False
```

The dimension is further passed to function Maze, which is essentially a class that has a constructor that generates a matrix of the given dimension. Below are 10 x 10 Maze matrix with different probabilities.

```
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 6]
```

Fig 1 (P = 0.2)

```
[0, 1, 0, 0, 1, 1, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 1, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 1, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 1, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[1, 1, 0, 0, 0, 0, 0, 1, 0, 0]
[1, 1, 0, 0, 1, 1, 1, 0, 0, 1]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 6]
```

Fig 2 (P = 0.3)

```
[0, 0, 0, 1, 0, 0, 1, 0, 1, 0]
[1, 1, 1, 0, 0, 1, 0, 1, 0, 1]
[1, 0, 0, 1, 0, 0, 0, 0, 1, 0]
[0, 1, 1, 1, 0, 0, 1, 1, 0, 1]
[0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 1, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 1, 0, 0]
[1, 0, 0, 0, 0, 0, 1, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 1, 0, 1, 6]
```

Fig 3 (P = 0.4)

1.2 Path Planning

→After generating 10 x 10 Maze with probability = 0.2 we are going to use different search algorithms to get the shortest path from the start to goal node.

```
[0, 0, 0, 0, 0, 1, 0, 1, 0, 0]
[0, 1, 1, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 1, 0, 1]
[0, 1, 0, 1, 0, 0, 0, 1, 0, 0]
[1, 1, 0, 1, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 1, 0, 0]
[1, 0, 1, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 6]
```

Fig 4 (P=0.2)

1.2.1 DFS (Depth First search)

We apply Depth First search on fig 4 and the smallest path from start to end is →

(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (1, 4), (2, 4), (2, 3), (2, 2), (3, 2), (4, 2), (5, 2), (5, 3), (5, 4), (4, 4), (4, 5), (3, 5), (3, 4), (3, 6), (4, 6), (5, 6), (5, 7), (4, 7), (4, 8), (3, 8), (3, 9), (4, 9), (5, 9), (6, 9), (6, 8), (7, 8), (7, 7), (8, 7), (8, 8), (9,8), (9,9)

Fringe Length = 36

1.2.2 BFS (Breath First search)

We apply breath First search on fig 4 and the smallest path from start to end is →

(0, 0), (0, 1), (1, 0), (0, 2), (2, 0), (0, 3), (2, 1), (3, 0), (0, 4), (1, 3), (2, 2), (1, 4), (2, 3), (3, 2), (1, 5), (2, 4), (4, 2), (1, 6), (3, 4), (5, 2), (1, 7), (2, 6), (3, 5), (4, 4), (5, 3), (6, 2), (1, 8), (3, 6), (4, 5), (5, 4), (6, 3), (2, 8), (4, 6), (5, 5), (6, 4), (7, 3), (3, 8), (4, 7), (5, 6), (6, 5), (7, 4), (8, 3), (3, 9), (4, 8), (5, 7), (6, 6), (7, 5), (8, 4), (9, 3), (4, 9), (7, 6), (8, 5), (9, 4), (5, 9), (7, 7), (8, 6), (9, 5), (6, 9), (7, 8), (8, 7), (9, 6), (6, 8), (8, 8), (9, 7), (8, 9), (9, 8),(9, 9)

Fringe Length = 67

1.2.3 A* (Euclidean Distance)

We apply A* using Euclidean Distance on fig 4 and the smallest path from start to end is →

(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (5, 2), (5, 3), (5, 4), (5, 5), (6, 5), (6, 6), (7, 6), (7, 7), (8, 7), (8, 8), (9, 8), (9, 9)

Fringe Length = 19

1.2.4 A* (Manhattan Distance)

We apply A* using Manhattan Distance on fig 4 and the smallest path from start to end is →

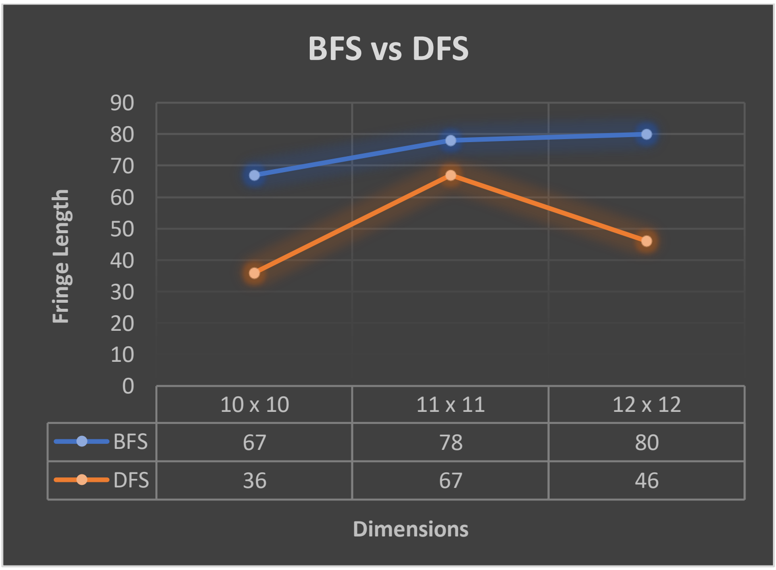
[(0, 0), (1, 0), (2, 0), (3, 0), (2, 1), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (6, 3), (7, 3), (8, 3), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9)]

Fringe Length = 20

1.3 OBSERVATIONS

→1.3.1 DFS vs BFS

- Fringe Length: One of the main reasons why fringe length of BFS is greater than fringe length of DFS is because there is no backtracking in BFS whereas DFS can do backtrack. Another is BFS has to traverse to all the nodes before reaching to the end node.
- Other disadvantage of BFS is that if the dimensions of maze are massive (For example 500 x 500) it can stuck in infinite loop while traversing because it doesn't have backtrack functionality.
- According to our data BFS is not helpful in finding shortest path and it will end up following the longest path
- In respective of memory if the dimensions of the maze is massive than DFS can be choose because we don't have to store the path in the memory.
- We did an experiment to give more explanation on the comparison of DFS and BFS. We constructed 10 x10, 11 x 11, 12 x 12 Maze and note their Fringe Length. The results of the experiment are graphical represented below.
- Through the results of the experiment we can conclude that the fringe length of BFS is greater than the fringe length of DFS. Hence DFS is can give the shortest path with less fringe value.



	10 x 10	11 x 11	12 x 12
BFS	67	78	80
DFS	36	67	46

Fig 5 (BFS vs DFS)

1.3.2 A*(Euclidean) vs A*(Manhattan)

- Fringe Length: One of the main reasons why Fringe length of A*(Manhattan) is greater that A*(Euclidean) because Euclidean distance is essentially displacement between the two points and Manhattan distance is the city block distance between the two points.
- For high dimensions the Manhattan can give us better result that Euclidean.
- Same experiment we conducted between A*(Euclidean) vs A*(Manhattan)
- As we can see in Fig 6, the shortest path to the end point is less for A*(Euclidean) as compared to A*(Manhattan) when the dimensions are increased .

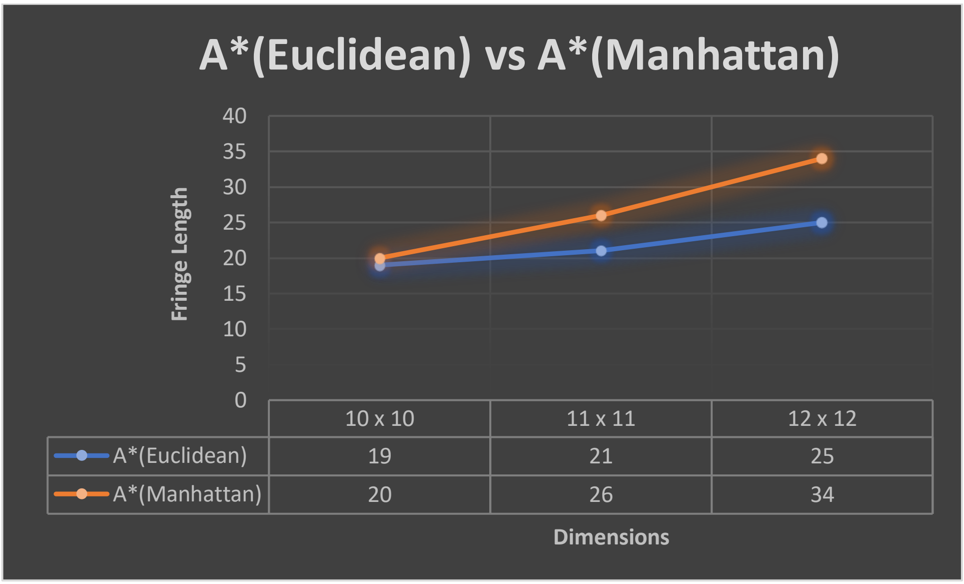
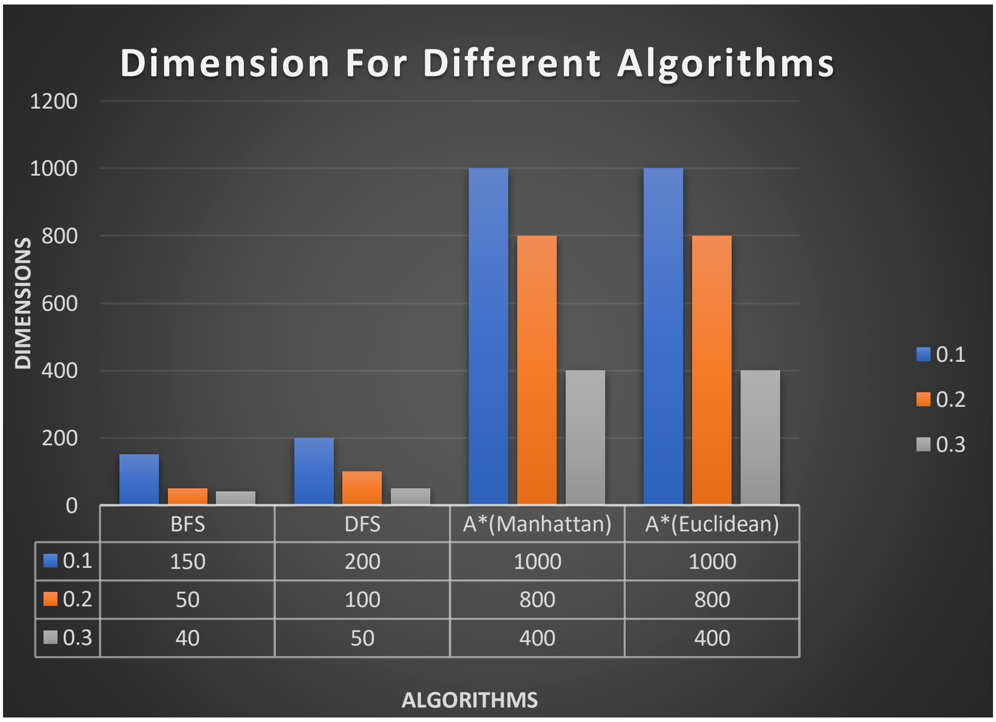


Fig 6 (A*(Euclidean) vs A*(Manhattan))

2. Analysis and Comparison

2(A) 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?

- The primary criteria for picking the dimension of the maze is the processing time required to solve the maze.
- Also, once the algorithms were proved to be working on mazes of dimension 1 to 15 the need to manually verify mazes of higher dimensions was redundant.
- We started with the map size of 5 and gradually incremented the map size and stopped when the time required to solve the map became too much for computer and algorithms to handle.
- Dimensions for each algorithm were such that it is large enough that some work had to be done to solve the maze and small enough that each algorithm gave an output without getting lost in a limbo –
- BFS – 40
- DFS – 50
- A*(Manhattan) – 400
- A*(Euclidean) - 400



	BFS	DFS	A*(Manhattan)	A*(Euclidean)
0.1	150	200	1000	1000
0.2	50	100	800	800
0.3	40	50	400	400

Fig 7. Map Dimensions for different Algorithms

2(b) 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.

```
[0, 0, 1, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[1, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 1, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 1, 1, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 6]
```

Fig 8.10 x 10 Maze(P=0.2)

→ After generating 10 x 10 Maze with probability = 0.2 we are going to use different search algorithms to get the shortest path.

DFS (Depth First search)

We apply Depth First search on fig 4 and the smallest path from start to end is →

[(0, 0), (1, 0), (1, 1), (1, 2), (2, 2), (2, 1), (3, 1), (3, 0), (4, 0), (5, 0), (5, 1), (6, 1), (6, 0), (7, 0), (7, 1), (8, 1), (8, 2), (8, 3), (7, 3), (7, 4), (7, 5), (6, 5), (5, 5), (5, 4), (4, 4), (3, 4), (2, 4), (2, 5), (3, 5), (4, 5), (4, 6), (5, 6), (5, 7), (5, 8), (6, 8), (6, 9), (7, 9), (7, 8), (8, 8), (8, 9), (9, 9)]

Time Taken to Run the Algorithm = 0.0041 Ms

Fringe Length = 41

BFS (Breath First search)

We apply Breath First search on fig 4 and the smallest path from start to end is →

[(0, 0), (0, 1), (1, 0), (1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (0, 3), (3, 2), (0, 4), (3, 3), (4, 2), (3, 4), (4, 3), (5, 2), (3, 5), (4, 4), (6, 2), (3, 6), (4, 5), (5, 4), (6, 3), (3, 7), (4, 6), (5, 5), (7, 3), (3, 8), (5, 6), (6, 5), (7, 4), (8, 3), (3, 9), (4, 8), (5, 7), (7, 5), (8, 4), (9, 3), (4, 9), (5, 8), (9, 4), (6, 8), (6, 9), (7, 8), (7, 9), (8, 8), (8, 9), (9, 8), (9, 9)]

Time Taken to Run the Algorithm = 0.0049 Ms

Fringe Length = 50

A* (Euclidean Distance)

We apply A* using Euclidean Distance on fig 4 and the smallest path from start to end is →

[(0, 0), (1, 0), (1, 1), (2, 1), (2, 2), (3, 2), (3, 3), (4, 3), (4, 4), (5, 4), (5, 5), (6, 5), (7, 5), (7, 4), (8, 4), (9, 4), (9, 3), (8, 3), (7, 3), (6, 3), (6, 2), (5, 2), (4, 2), (5, 6), (5, 7), (5, 8), (6, 8), (7, 8), (8, 8), (9, 8), (9, 9)]

Time Taken to Run the Algorithm = 0.0286 Ms

Fringe Length = 31

A* (Manhattan Distance)

We apply A* using Manhattan Distance on fig 4 and the smallest path from start to end is →

[(0, 0), (1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (4, 2), (5, 2), (6, 2), (6, 3), (7, 3), (8, 3), (9, 3), (9, 4), (8, 4), (7, 4), (7, 5), (6, 5), (5, 5), (5, 6), (5, 7), (5, 8), (6, 8), (7, 8), (8, 8), (9, 8), (9, 9)]

Time Taken to Run the Algorithm = 0.0051Ms

Fringe Length = 27

From the data which we get, it is clear that DFS provide the shorter path than BFS. Among all A*(Manhattan) provides shortest path. The time taken to run algorithm is least for DFS.

2(C) Given dims, 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 determine how does maze solvability depend on p. An experiment is conducted in which 10 x 10 size of maze is made and was solved using different value of p using DFS as it takes least amount of time to solve the maze.

P	Fringe Length	Time
0.1	52	0.051
0.2	61	0.007
0.3	60	0.0351
0.4	0	0
0.5	0	0

Table 1 (Fringe Length and Time for different value of P)

- As the P increases the fringe length increase. The path to reach end node increase.
- When the p = 0.4 and after that p value the fringe length become 0 as there no path found to reach end as the number of road blocks increased.

For estimate the probability that a maze will be solvable by generating multiple mazes and checking them for solvability we chose DFS as it took least time when applied in question 2 part 1 and DFS also requires less memory which makes generating large number of maps easier for the computer.

20 x 20 size maze is made for value of p range from p=0.1 to p=0.8 and for each p vale 10 iterations are made to know the probability of solvable matrices.

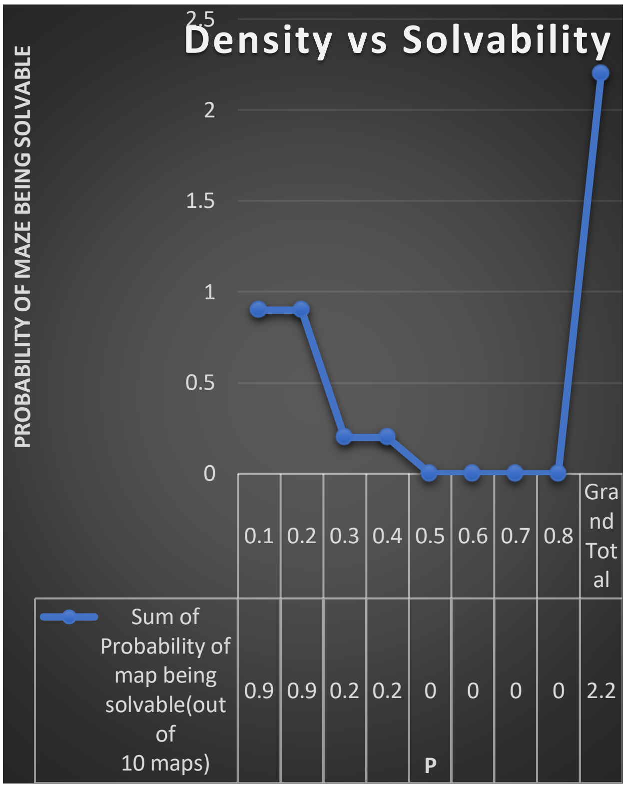


Fig 9

Probability(P)	Probability of map being solvable(out of 10 maps)
0.1	0.9
0.2	0.9
0.3	0.2
0.4	0.2
0.5	0
0.6	0
0.7	0
0.8	0

Table 2

The output from fig 9 gives us clear idea that after the value of p= 0.4 there is no solvable map generated so the value of p_0 = 0.5. Therefore, for p <0.5 there is a solvable map.

2(D) For p in [0, p_0] 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?

→To get the shortest path between starting point and ending point we used A*(Euclidean) algorithm as the results from ques 1 indicated that out of four algorithms A*(Euclidean) gives the shortest path as Euclidean distance is essentially displacement between the two.

For this problem we made ten 20 x 20 for each value of p in [0, p_0] matrix maze and note down the fringe length.

Observations

- From the data plotted it is observed that the average path increases as the value of P increases.
- After p = 4 the average path is 0 as there is no shortest path available and extremely rare to find any path as the number of road blocks increases.

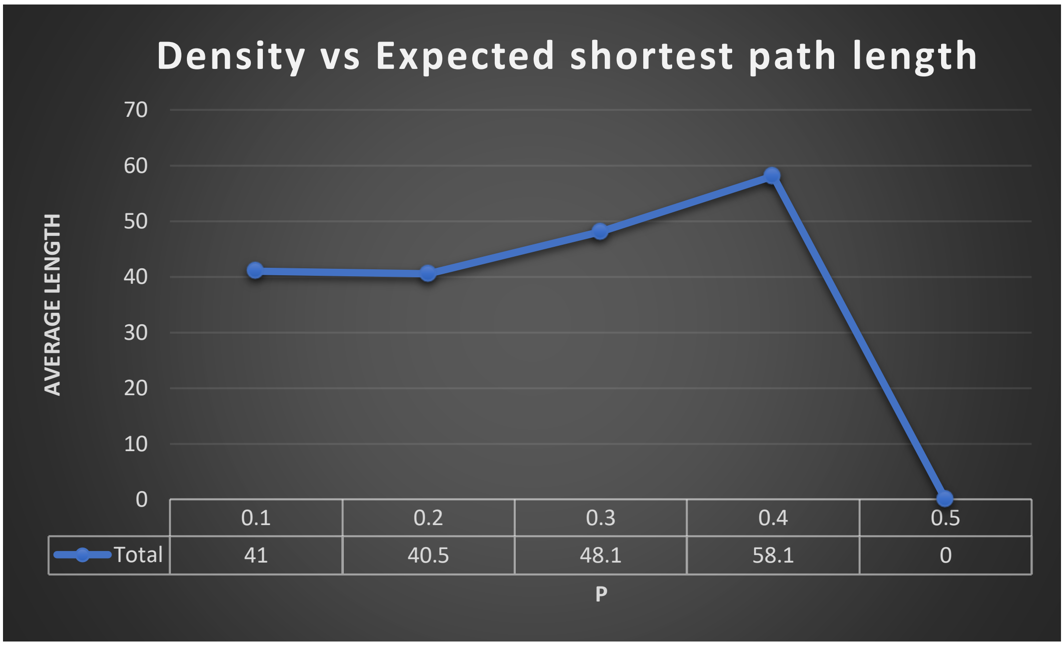


Fig 10

P	1	2	3	4	5	6	7	8	9	10	avg
0.1	42	39	39	39	39	43	45	39	41	44	41
0.2	39	44	39	39	39	39	43	41	39	43	40.5
0.3	52	69	47	49	63	39	39	43	41	39	48.1
0.4	48	46	49	69	56	78	65	54	52	64	58.1
0.5	0	0	0	0	0	0	0	0	0	0	0

Table 3

2(E) Is one heuristic uniformly better than the other for running A∗? How can they be compared? Plot the relevant data and justify your conclusions

→To know Manhattan distance is better or not than Euclidean for running A*, we made ten (20 x 20) size of maze and implement A*(Euclidean) and A*(Manhattan) on each maze with the value of (p = 0.2). Below are the results which we got.

Observations

- As we can see from table the average length of the fringe for A*(Euclidean) is less that A*(Manhattan) and the data from the graph (fringe length for A*(Euclidean) & A*(Manhattan)) also shows that to get the shortest path from start to end A*(Euclidean) performed well as compared to A*(Manhattan)
- From the table we can conclude that when we talk about time to run an algorithm A*(Manhattan) is better than A*(Euclidean). Data which we got from 10 iterations, from that we plotted a graph (Time taken by A*(Euclidean) & A*(Manhattan) which shows that time taken to run A*(Manhattan) is less as compared to A*(Euclidean).

Hence from the data we conclude that to get the shortest path A*(Euclidean) is better but when we want fast algorithm A*(Manhattan is better).

Iteration	A*(euclidean) , Fringe length	A*(euclidean) ,Time	A*(manhattan),Fringe Length	A*(manhattan),Time
1	41	0.019	48	0.017
2	48	0.021	51	0.017
3	39	0.014	39	0.011
4	41	0.015	44	0.019
5	46	0.017	46	0.015
6	40	0.021	47	0.017
7	55	0.021	51	0.018
8	47	0.018	47	0.017
9	45	0.012	41	0.012
10	39	0.015	39	0.012
Average	44.1	0.017	45.3	0.015

Table 4

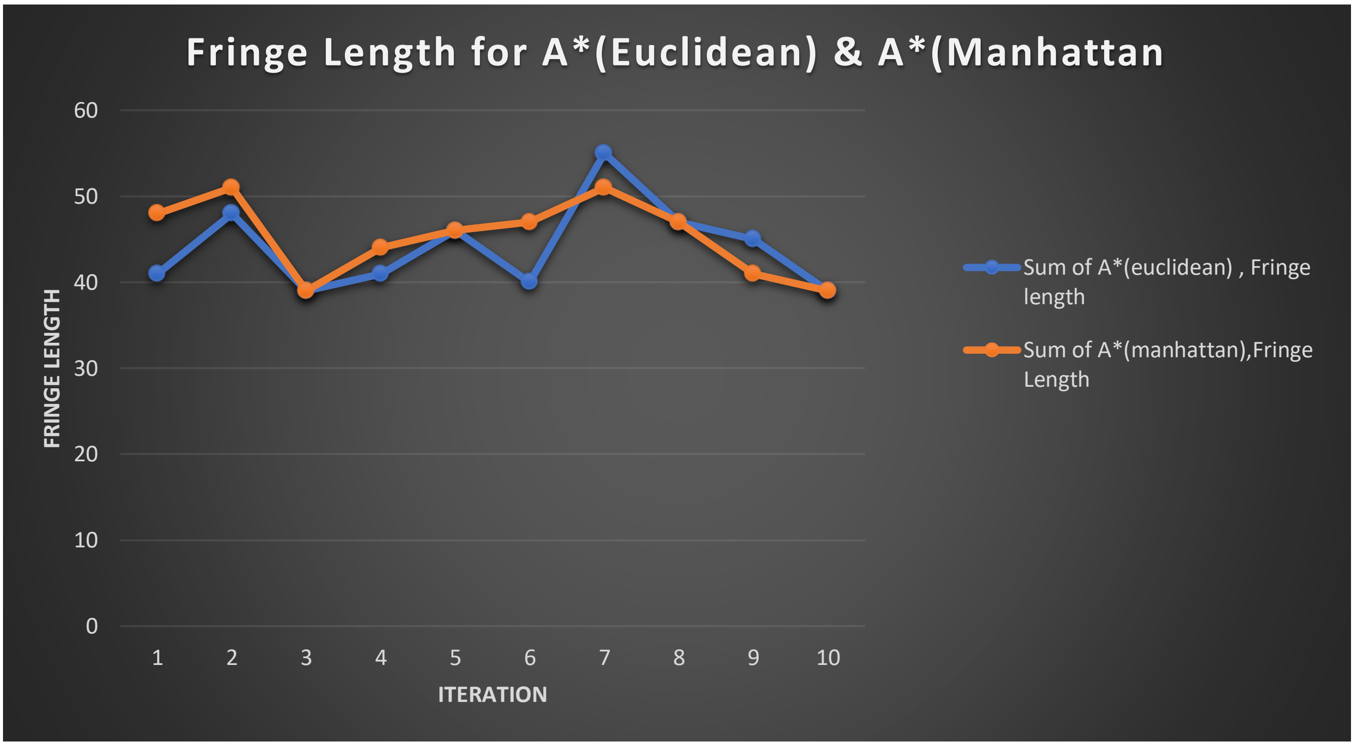


Fig 11

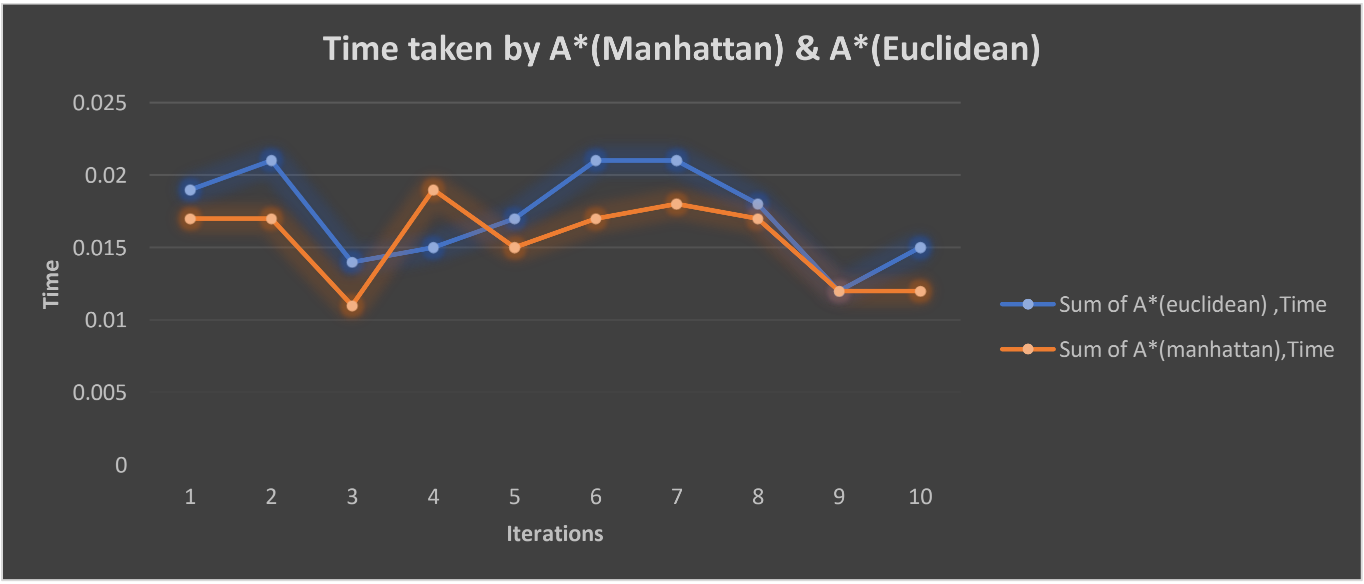


Fig 12

2.(F) Is BFS will generate an optimal shortest path in this case - is it always better than DFS? How can they be compared? Plot the relevant data and justify your conclusions.

→No, BFS won't generate an optimal shortest path for [0, p_0] as for 20 x 20 size of maze as compare to A*(Euclidean) as it will traverse each node before reaching to end point. To Justify that BFS is better than DFS or vice versa we made ten 20 x 20 matrices for p=[0, p_0].

P	1	2	3	4	5	6	7	8	9	10	AVG
0.1(DFS)	132	290	189	367	127	162	341	167	335	125	223.5
0.1(BFS)	359	282	343	335	333	265	317	322	303	300	315.9
0.2(DFS)	125	130	86	303	86	240	259	91	306	83	170.9
0.2(BFS)	225	302	249	277	228	238	260	218	258	254	250.9
0.3(DFS)	95	112	157	176	243	80	250	109	186	184	159.2
0.3(BFS)	147	104	150	201	139	102	128	215	136	219	154.1
0.4(DFS)	0	0	0	0	0	0	0	0	0	0	0
0.4(BFS)	0	0	0	0	0	0	0	0	0	0	0
0.5(DFS)	0	0	0	0	0	0	0	0	0	0	0
0.5(BFS)	0	0	0	0	0	0	0	0	0	0	0

Table 5

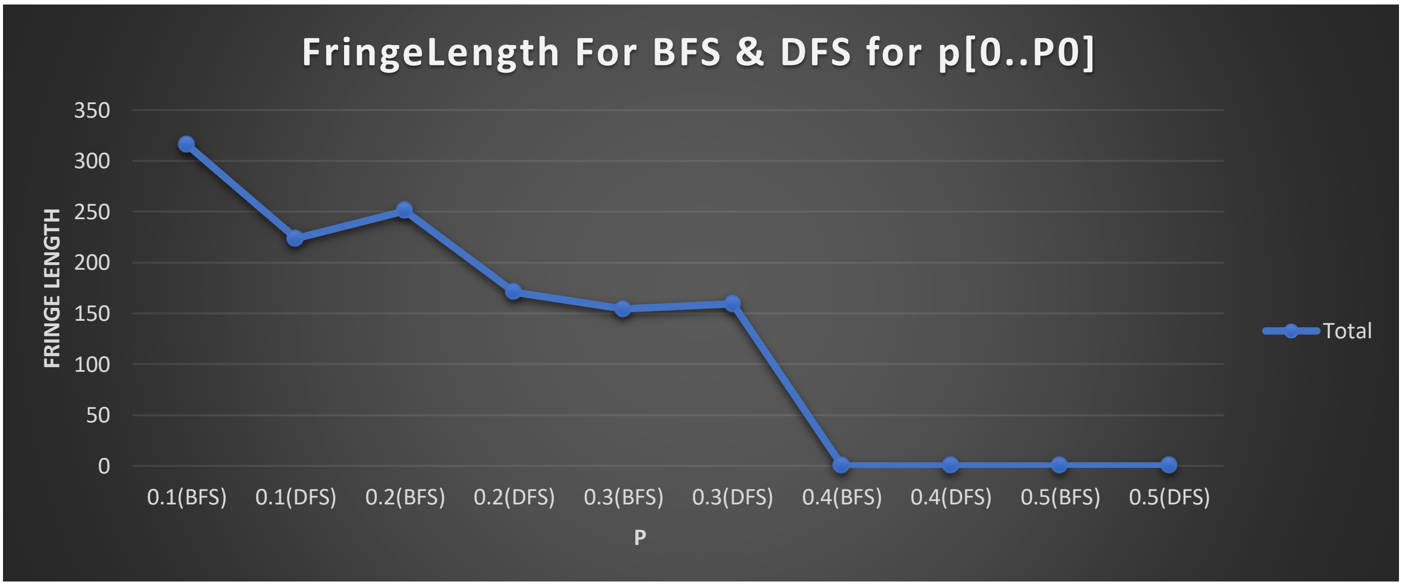


Fig 13

Observations

- From the data shown in the table we can conclude that DFS follow shorter path as compared to BFS for the different values of P.
- It can be possible in some cases BFS can give shorter path as compared to DFS, but in most of the cases DFS gives shorter path.

2(G) Do these algorithms behave as they should?

→From the results which we got from the previous questions we can conclude that

- A*(Euclidean) gives the shortest path between the start and end point.
- In terms of time taken to solve the maze A*(Manhattan) gave the best result.
- BFS follows the longest path as compared to other algorithms
- DFS works better than BFS in term of fringe length.

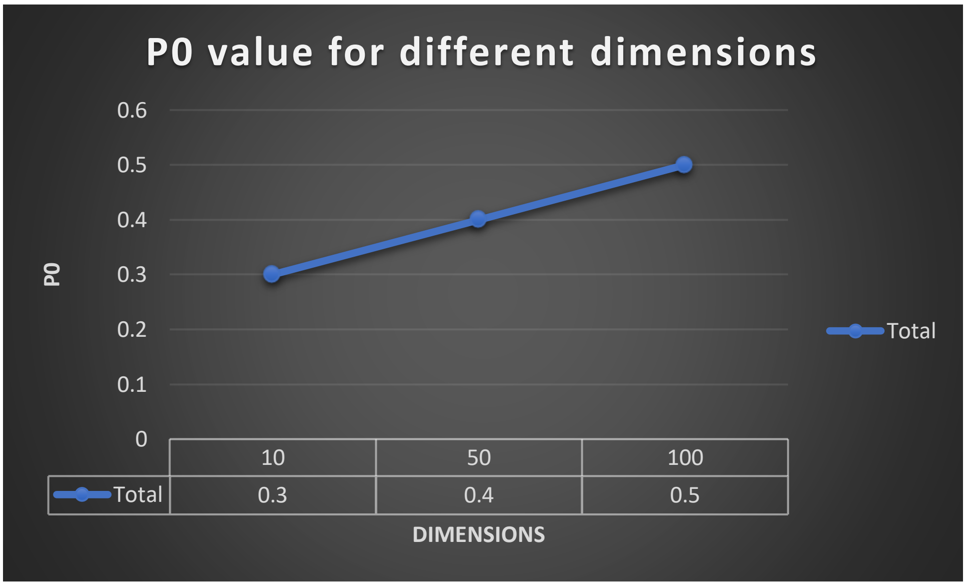
2(H) For DFS, can you improve the performance of the algorithm by choosing what order to load the neighbouring rooms into the fringe? What neighbours are 'worth' looking at before others? Be thorough and justify yourself.

→Yes, it can improve the performance of algorithm if we choose in which direction the algorithm should traverse.

We know that the shortest path to reach end from starting point is to go east first and then south. In our algorithm DFS is choosing random neighbours to traverse so it can traverse to some extra nodes which is not essential to reach the end point and if we code in a manner that it should select the neighbour in the east and then to another nodes and then choose south other than other nodes than it can helps to decrease the fringe length.

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

→To show threshold probability depends on n, We made 10 x 10 dimension of maze and find that p0 value is 0.3(we get a solvable path for p < po) and when we increase the dimension of maze by 100 x 100 the threshold probability increase by 0.5 so from that We can say that threshold probability will increase as the dimension of the maze increase as for 10 x 10 size of maze threshold is p0 =0.3 and for maze 100 x100 the threshold is p0 =0.5.



3 Generating Harder Maze

3(A) 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?

→We implemented hill climbing to generate hardest maze because in random walk what we essentially do is we repeatedly generate a new maze while keeping track of the hardest maze generated so far. This is in a sense un-informed search and hence not very efficient to find hardest maze. But we overcome this by using Hill Climbing where we choose a random point and we check if the wall is present there or not and if the wall is present, we remove the wall and if it is not, we add the wall. During this process we compute the fringe length for the newly obtained maze and If this fringe length is greater than the old maze than we store it as a harder maze. We continue this process till we reach a local maxima. Now in order to come out of the local maxima we perform a random restart by generating a random a new maze of the same dimension and again performed the above-mentioned steps. This will gradually help us to reach global maxima.

```
while flag:

    point = choice(setup_cells(rows, cols))

    if (newMazes[point[0]][point[1]] == 1):
        newMazes[point[0]][point[1]] = 0
    else:
        newMazes[point[0]][point[1]] = 1

    if (fringLength < solveMazeAManH(newMazes, rows, cols)):
        print("Harder Maze--------------------")
        for i in range(len(newMazes)):
            print(newMazes[i])
        print('Fringe Length for Harder maze')
        print(fringLength)
        #print(newMazes)
        fringLength = solveMazeAManH(newMazes, rows, cols)
        count = 0


    else:
        count += 1
        if count >= 10:
            flag = False
print(fringLength)
```

3(B) 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?

→Using the hill climbing method to find a hard maze if not the hardest maze came with its own set of advantages and disadvantages. Firstly, we can never be sure if the maze obtained is the hardest maze of just another harder maze for a given dimension. We might be able to physical see and draw conclusions for a maze of small dimensions but that to will not be possible for maze of higher dimensions. Secondly, there is no way to definitively know if the algorithm is stuck in a local maxima or global maxima. We try to address this problem by performing N random restarts by generating matrices of same dimensions. Thirdly, no one can deny the fact that hill climbing is better that "Random walk", as instead of generating a random maze every single time its builds over the existing maze to result in a harder maze. This is in some sense an informed search.

3(C) Try to find the hardest mazes for the following algorithms using the paired metric.

→We used to find the hardest maze

– A∗-Manhattan with Maximal Nodes Expanded

– A∗-Manhattan with Maximal Fringe Size

```
    else:
        count += 1
        if count >= 10:
            flag = False
print(fringLength)
print("one")
newMazes = creatMaze(rows)
pCout += 1
if pCout >= 100:
    pFlag = False
```

```
[0, 1, 1, 0, 0, 0, 1, 0, 0, 1]
[0, 0, 0, 1, 1, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 1, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 1, 0, 1, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 1, 0, 0, 0, 1, 6]
```

Fringe length = 21

Maze 1

```
Harder Maze-------------------
[0, 0, 0, 0, 0, 1, 1, 1, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 1, 1]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 6]
Fringe Length for Harder maze
24
```

Maze 2

```
Harder Maze-------------------
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 1, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 0, 0, 0, 1, 1, 0, 0]
[1, 0, 1, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 1]
[0, 0, 1, 1, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 1, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 1, 0, 6]
Fringe Length for Harder maze
31
```

Maze 3

```
Harder Maze-------------------
[0, 0, 0, 1, 0, 1, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 1, 0, 0, 0, 0, 1, 0]
[0, 0, 1, 1, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 1, 1, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 1, 0]
[0, 0, 1, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 1, 0, 6]
Fringe Length for Harder maze
36
```

Maze 4

```
Harder Maze-------------------
[0, 0, 1, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
[1, 1, 1, 1, 0, 0, 1, 1, 1, 0]
[1, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 0, 1, 0, 1, 1]
[0, 1, 0, 0, 1, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 1, 6]
Fringe Length for Harder maze
41
```

Maze 5

```
Harder Maze-------------------
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 1, 0, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 1, 0, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 1, 6]
Fringe Length for Harder maze
43
```

Maze 6

From multiple iteration, Maze 6 is the hardest maze with fringe size of 43.

3(D) Do your results agree with your intuition?

→Yes, our result is inline with our intuition because the results obtained clearly shows that the fringe length require to solve the maze gradually increase as the difficulty of the maze is increased. This was for the reason that the number of walls in the maze increased which resulted in greater backtracking and thus exploring multiple nodes.

4. Thinning A*

4.1 Simplifying the maze

Here the problem statement required us to remove a certain fraction of the walls (ones) in the maze and solve the thus obtained simpler maze. We can safely conclude that the fringe length obtained for the simpler maze will always be equal or less than that of the original (harder) maze.

Now the first part of this problem statement, to remove the walls was achieved through the following implementation:

```
31  def CountOne():
32      count = 0
33      for i in range(len(originalMaze)):
34          for j in range(len(originalMaze[0])):
35              if originalMaze[i][j] == 1:
36                  count = count + 1
37
38      p = random.randint(1, 10)
39      print("one")
40      print(p)
41      f = (p / 10) * count
42      print("two")
43      print(math.ceil(f))
44      return math.ceil(f)
```

```
85  def updateCurrMaza():
86
87      fraction = CountOne()
88      print(fraction)
89      k = 0
90      while k < fraction:
91
92          tempPoints = matrixPoints
93          point = random.choice(tempPoints)
94          tempPoints.remove(point)
95          if (originalMaze[point[0]][point[1]] == 1):
96              removeIndex.append(point)
97              originalMaze[point[0]][point[1]] = 0
98              k += 1
```

In the above code snippet the function “countOne()” checks for the number of walls (ones) in the maze. Following this it generates a random number “p” between 1 to 10 which when multiplied with count and divided by 10 as in line 41 gives a random fraction of walls (ones) to be removed from the maze. This number is then passed to the function “updateCurrMaza()”.

In updateCurrMaza() a while loop runs “n” times where n equals the fraction of walls to remove. The new maze is then stored in a temporary 2d matrix “originalMaze” as in line 97.

4.2 Solving the simpler maze

Now we use A* (Manhattan distance) to solve the simplified maze obtained above. We store the fringe length required to solve the simplified maze and use this as a reference for the original (harder) maze.

```
189
190             neighbor = leastPathChildMan(heuristic, current, end)
191             backTrackPriority.append(current)
192             current = neighbor
193             frinLength += 1
194         except:
195             print("No path Found!")
196             return 0
197     return frinLength
198
```

This can be seen in the above code snippet at line 193, were we add 1 to the “frinLength” every time we explore a new node or back track. But if we see that there is no path available to solve the maze we return 0. This “frinLength” will be used a heuristic to solve the harder maze.

4.3 Solving the original(harder) maze:

Now in our attempts to solve the original maze using the fringe length obtained by the simpler maze as heuristic we used the following approach.

```
311         if not currentNeighboursss:
312
313             if not backTrackPriority:
314                 print("No path available!")
315             else:
316                 while not currentNeighboursss:
317                     current = nextPopMan(backTrackPriority, end)
318                     backTrackPriority.remove(current)
319                     neighboursDFSandA(newMaze, current, rows, cols)
320         tempMat = []
321         c=0
322         neighbour = ()
323         for x in currentNeighboursss:
324             tempMat = np.array(temp)
325             tempMat = tempMat[x[0]:rows, x[1]:cols]
326             if c==0 :
327                 tempLen = solveUsingAMansss(tempMat) + len(backTrackPriority)
328                 neighbour = x
329                 c+=1
330             else:
331                 if solveUsingAMansss(tempMat) + len(backTrackPriority) < tempLen:
332                     neighbour = x
333
334         backTrackPriority.append(current)
335         current = neighbour
336         print(current)
337
```

We first take a starting point of (0,0) on the maze matrix. Following this we use the “neighboursDFSandA()” function to find all the neighbors where the A* algorithm can move to. We then iterate through the neighbors to find the neighbor whose fringe length to the goal node plus the node traversed to obtain the total fringe length so far was closest to the fringe length of the simple maze. This implementation can be seen in above code snippet at line 327.

Now we switch the current node with the neighbor closest to the fringe length of the heuristic. Then we repeat this process till we reach the goal node. Now in case we hit a dead end we back track with the help of the “backTrackPriority” list which keeps track of all the previously traversed node. This is seen in line 334.

4.4 Observations:

To address the question: Is there a value of "q" where solving the thinned maze first as a heuristic for solving the original maze actually makes solving the maze easier? We implemented an algorithm to randomly remove a fraction of ones from the maze matrix. The implementation of this was clearly explained in section 4.1.

```
New Maze
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[1, 0, 0, 0, 0]
[0, 0, 1, 0, 1]
[0, 0, 0, 0, 6]
```

Fig 4.1

```
Thinning the matrix:
3
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 6]]
path for the simpler maze!
The path to be take is:
(0, 0)
(1, 0)
(2, 0)
(3, 0)
(4, 0)
(4, 1)
(4, 2)
(4, 3)
(4, 4)
```

Fig 4.2

The above fig 4.1 is a new maze generated with ones at (2,0), (3,2) and (3,4), now it can be seen in figure 4.2 that the "q" or thinning factor here is 100% thus all the walls (ones) were removed in this case and the simplest maze is thus obtained.

Through multiple trial and error cases we found that setting q=1 produced the best results because this resulted in removing all the walls of the maze and finding the path of the simplest maze possible for a given dimension. This essentially gave us the city block path from the start node to the end node.

5. Maze on fire.

5.1 Setting the cells on fire:

All prior solutions discussed so far are in some sense ‘static’. The solver has the map of the maze, spends some computational cycles determining the best path to take, and then that path can be implemented, for instance by a robot actually traveling through the maze. But what if the maze were changing as you traveled through it?

5.1.1 Conditions in place:

- Any cell in the maze is either ‘open’, ‘blocked’, or ‘on fire’. Starting out, the upper right corner of the maze is on fire.
- You cannot move into cells that are on fire, and if your cell catches on fire you die.
- But each time-step, the fire may spread, according to the following rules:
    - If a free cell has no burning neighbours, it will still be free in the next time step.
    - If a free cell has k burning neighbours, it will be on fire in the next time step with probability 1 − (1/2)^k .

5.1.2 Maze Burner Algorithm:

If we view the maze in matrix form the starting point of the maze is (0,0). In our implementation the maze is traversed using the A*(heuristic: Manhattan distance) algorithm. The next step taken by the algorithm is to the “unwalled” Neighbour of (0,0).

Now taking into consideration the pre-requisites of setting the maze on fire, post the first move by the A* algorithm the “maze burner” algorithm sets the top right corner block of the maze on fire, as seen in the below code snippet.

```
151
152     if not cellsOnFire:
153         tempCellsOnFire.append((0,cols-1))
154         return
155
```

The code in line 152 checks for any burning maze blocks. If no block of maze is on fire then it sets the right most block on fire, as seen in line 153. Here (0, cols-1) represent the rows and columns of the maze matrix.

Now let’s take into consideration the scenario after A* has made its next move on the maze matrix. The Maze burner algorithm has to now burn the maze blocks that are adjacent to a burning block with a probability of 1 − (1/2)^k (where k is the number of burning neighbours). This is achieved by the Maze burner in the following manner:

```python
157     for point in copyCellsOnFire:
158         neighboursOfBurning(point, rows, cols)
159         for neighbour in aboutBurningNeighbours:
160             neighboursToBurn(neighbour, rows, cols)
161             k = 0
162             for tempNeighbour in neighboursOfBurningToBurn:
163                 if tempNeighbour in copyCellsOnFire:
164                     k += 1
165
166             power = 0.00
167             power = math.pow((1/2),k)
168             prob = 0.00
169             prob = (1 - power)*10
170
171             rand = random.randint(0,9)
172
173             if rand <= prob:
174                 tempCellsOnFire.append(neighbour)
175
```

In the above code snippet line 157 checks for maze blocks that are already on fire. Line 158 checks all the neighbour of the burning block. Now we iterate each of these neighbours and check the number of maze blocks burning around these neighbours. This is done in line 159 “aboutBurningNeighbours” gives us details of all the burning neighbours” K”. Once we have found the value of K, we us it to compute the value of (1/2) ^k as in line 167. We then subtract this value from 1 to find the probability with which the new maze block can catch fire. Once we have the probability, we generate a random variable between 0-9 and check if its value is less than the computed (probability value) * 10 as it can be seen in lines 166 through 174. This is how we were able to burn a maze block with a probability of 1 – (1/2) ^k.

5.2 Traversing Burning Maze:

The traversal of the maze was done using the A* algorithm (heuristic: Manhattan distance). The starting point of the maze is (0,0). Starting from this point the algorithm investigates the neighbours of (0,0). It then finds the neighbour that is closest to the goal node (rows-1, cols-1) that is not walled or set on fire. This was implements using two functions “leastPathChildMan ()” and “neighboursDFSandA ()”.

```python
def neighboursDFSandA(current, rows, cols):  # finding all the neighbours of a node in DFS

    currentRow = current[0]
    currentCol = current[1]

    currentNeighbours.clear()


    # checking for the north neighbour
    if (currentRow - 1 >= 0 and burningMaze[currentRow - 1][currentCol] != 1):
        if ((currentRow - 1, currentCol) in unvisited and (currentRow - 1, currentCol) not in cellsOnFire):
            currentNeighbours.append((currentRow - 1, currentCol))

    # checking for the south neighbour
    if (currentRow + 1 <= rows - 1 and burningMaze[currentRow + 1][currentCol] != 1):
        if ((currentRow + 1, currentCol) in unvisited and (currentRow + 1, currentCol) not in cellsOnFire):
            currentNeighbours.append((currentRow + 1, currentCol))

    # checking for the east neighbour
    if (currentCol + 1 <= cols - 1 and burningMaze[currentRow][currentCol + 1] != 1):
        if ((currentRow, currentCol + 1) in unvisited and (currentRow, currentCol + 1) not in cellsOnFire):
            currentNeighbours.append((currentRow, currentCol + 1))

    # checking for the west neighbour
    if (currentCol - 1 >= 0 and burningMaze[currentRow][currentCol - 1] != 1):
        if ((currentRow, currentCol - 1) in unvisited and (currentRow, currentCol - 1) not in cellsOnFire):
            currentNeighbours.append((currentRow, currentCol - 1))
```

From the starting point of (0,0) the algorithm now finds the neighbours of this point using the function shown in the above code snippet. The current maze block is passed to this function and it computes the position of the neighbour that are not walled or on fire. The comments above the if conditions describe which neighbours availability is checked. This function updates a list “currentNeighbours” that contains the list of neighbours that are available for traversal from the current node.

The above obtained list is then passed to "leastPathChildMan ()". This computes the Manhattan distance of each node to the final goal Node and returns the neighbor closest to the goal node.

```
129 def leastPathChildMan(heuristic, current, end):#finds the next child which is closest to th
130
131     hOfX = heuristic
132     shortestPoint = ()
133     shortestDist = 0
134
135     for point in currentNeighbours:
136         gOfX = calManhattanDis(current,point) + calManhattanDis(point,end)
137         fOfX = gOfX + hOfX
138         if shortestDist == 0:
139             shortestDist = fOfX
140             shortestPoint = point
141         elif shortestDist > fOfX:
142             shortestDist = fOfX
143             shortestPoint = point
144
145     return shortestPoint
```

The Manhattan distance is computed with the help of the function "calManhattanDis()" which returns the distance between two points. Line 136 computes the total Manhattan distance between the current point to the goal node. It then returns the point with the shortest Manhattan distance to the goal node.

Once we obtain the next point to traverse to we update the current to this point and add the previous current point to the list "backTrackPriority".

```
236                 neighbor = leastPathChildMan(heuristic, current, end)
237                 backTrackPriority.append(current)
238                 current = neighbor
239                 print(current)
240                 setCellOnFire(rows, cols)
241                 for temp in tempCellsOnFire:
242                     if temp not in cellsOnFire:
243                         cellsOnFire.append(temp)
```

"backTrackPriority" helps us keep track of all the already traversed node. This help us in backtracking when we find ourselves surrounded by a wall or a node on fire.

Now that we have traversed to the next node as in line 238. We now call "setCellOnFire()" so as to check for new maze blocks that might have caught fire. We then append the new cells on fire in to the list "cellsOnFire" as in line 243 to help us keep track of all the cells that are currently on fire.

5.2.1 Overcoming a dead end:

Now lets us consider the possibility the node we are on does not have any neighbors. In this case we will need to back track to a prior node that still has other neighbors that can get us to the goal node.

```
        try:
            if not currentNeighbours:

                if not backTrackPriority:
                    print("No path available!")
                    return
                else:
                    while not currentNeighbours:
                        current = nextPopMan(backTrackPriority, end)
                        backTrackPriority.remove(current)
                        neighboursDFSandA(current, rows, cols)
                        setCellOnFire(rows, cols)
                        for temp in tempCellsOnFire:
                            if temp not in cellsOnFire:
                                cellsOnFire.append(temp)

                        print(cellsOnFire)

                        if current in cellsOnFire:
                            print("Sorry, but you were burnt :( ")
                            return
```

The above code snippet is an implementation of this back tracking. In line 215 we check for the neighbors of the current node. If we see that we are at a dead end the algorithm then check if the previous node we arrived from is still available. Else if the previous node has caught fire then we have “No path available!”. If there are still node available in the “backTrackPriority” this tells us that we still have a path available. Now in line 222 we call the “nextPopMan()” function that checks the backTrackPriority for still viable nodes. The check for viability is done by checking the neighbors of every node in the backTrackPriority list. Once we find this node we pass this node as the current node and remove all the other nodes from the list.

Now during the course of this if at any point the current node we are on catches fire then we show that “we were burnt”.

The test for death happens at two stages. One when we are back tracking or if the node we are on spontaneously catches fire.

```
if current in cellsOnFire:
    print("Sorry, but you were burnt :( ")
    return
```

For both these cases the test condition is the same. We check is the current node we are on is in the list “cellsOnFire”. If yes then the condition for our death is true and we print “Sorry, but you were burnt :(” and terminate the program.

--END--