CS520 MAZE RUNNER: ASSIGNMENT 1

- Amit Chawla(ac1910)
- Karthik Govindappa(kg667)
- Ritesh Sawant(rs1782)

Q1)

Design Pattern:

We have used Object-Oriented Design Pattern for our program. There are 3 major classes that encompass the majority of the program

- **Location**: The location class represents a location on the maze and contains helper functions to generate neighbors of the current location, check for validity of the moves, compare two locations for equality etc.
- **Maze:** The Maze class contains the code for generation and manipulation of the maze.

  Maze Generation - For maze generation we have used a binomial sampling with n=1 and probability = Probability of Occupancy and drew a sample of dim X dim to generate the maze.
- **SearchPath:** Contains all the SearchAlgorithms as instance methods. Takes the generated maze object as a parameter and runs the desired search algorithm on it.
- **Result:** All the search Algorithm methods return an object of Result class which contains the path calculated and other metrics like the expanded nodes, max fringe length etc to help analyze these algorithms.

We have parallized most of our simulations and took advantage of the fact, that all the runs are independent.

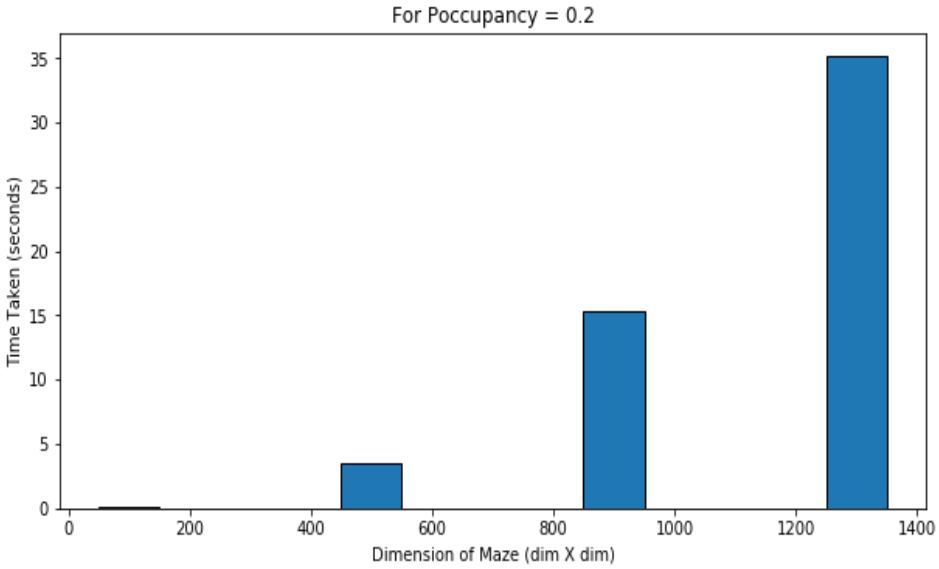
Link to Google Colab Notebook:
https://colab.research.google.com/drive/1zsV-1wlXBZEswE4bDEphBTbdxdgoJ2QU

- Here you can find a running instance of the code for all the problems presented, also we have attached a local copy of the Jupyter notebook with our assignment for reference.

Q2) a)

We ran BFS(will expand every node at each level until it reaches the destination and will take maximum time) for **[**dim_range = range(100,2100,400) and Probability = 0.2 (lesser occupancy probability results in higher number of paths and gives an upper bound on time) **]** on Google colab platform and noted the following execution times.

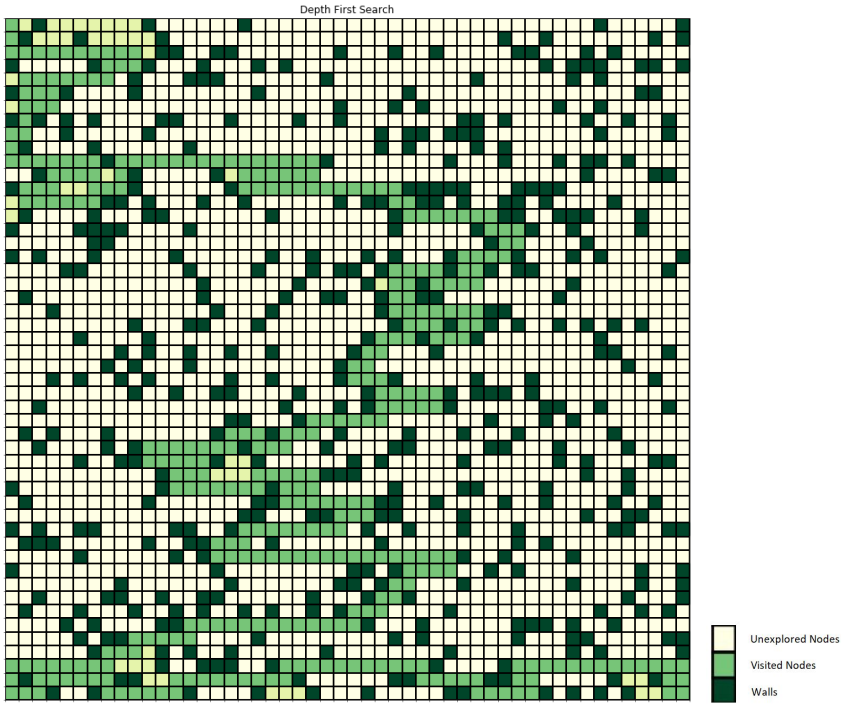
- 100 X 100 - 78.2 ms
- 500 X 500 - 3.47s
- 900 X 900 - 15.4s
- 1300 X 1300 - 35.2s
- 1700 X 1700 - 323us (because no path was found)

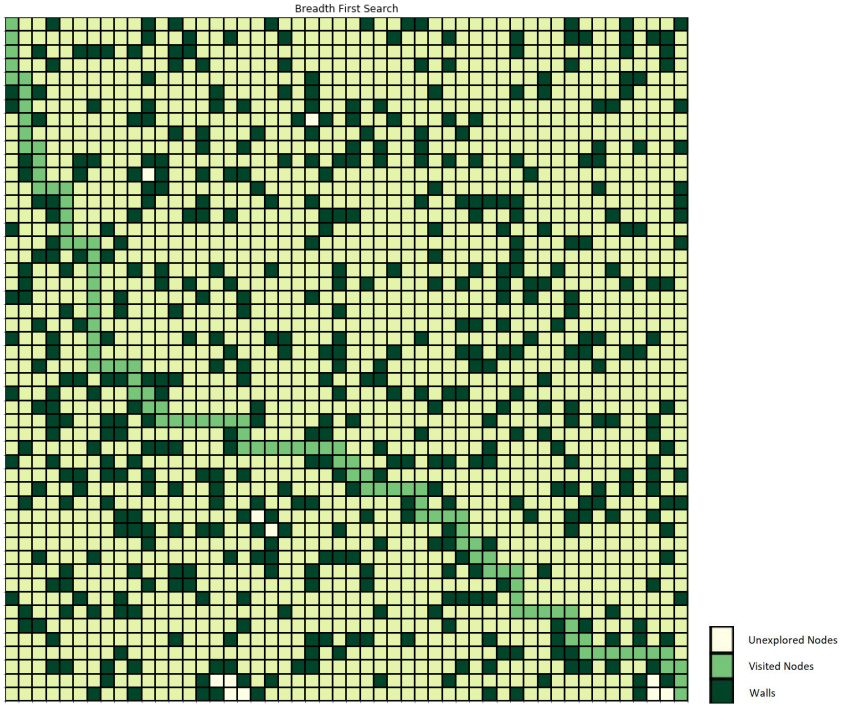


So, we decided to go with a 500 X 500 maze which can be solved in roughly 3.5 seconds for a 0.2 probability.
As the probability will increase, the algorithm will converge faster making it relatively faster to run larger number of simulations.

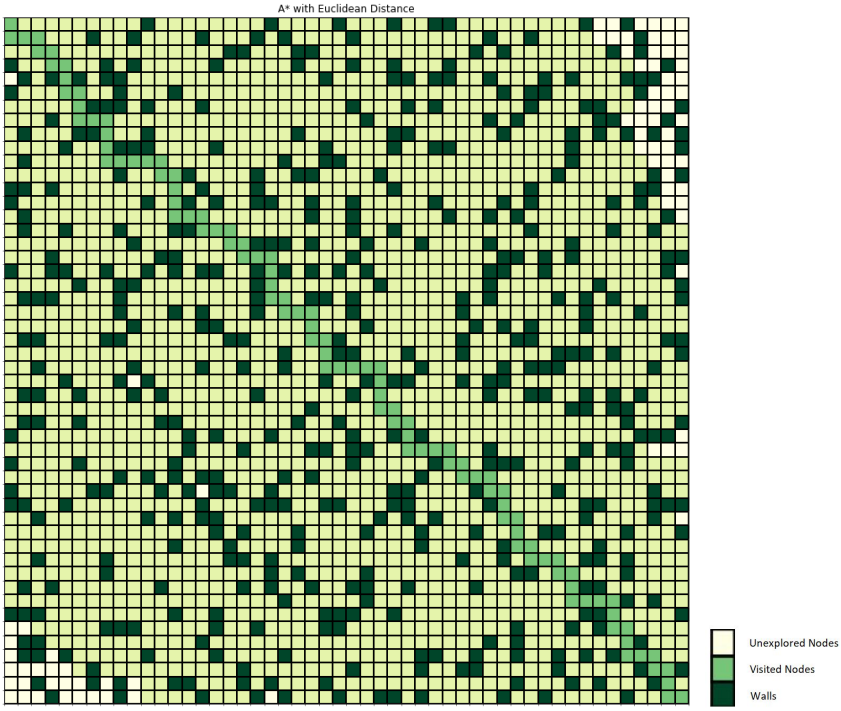
Q2) b)

The below images show the path taken by each algorithm for the same maze of size 50. The order of moves were Up, Down, Right, Left

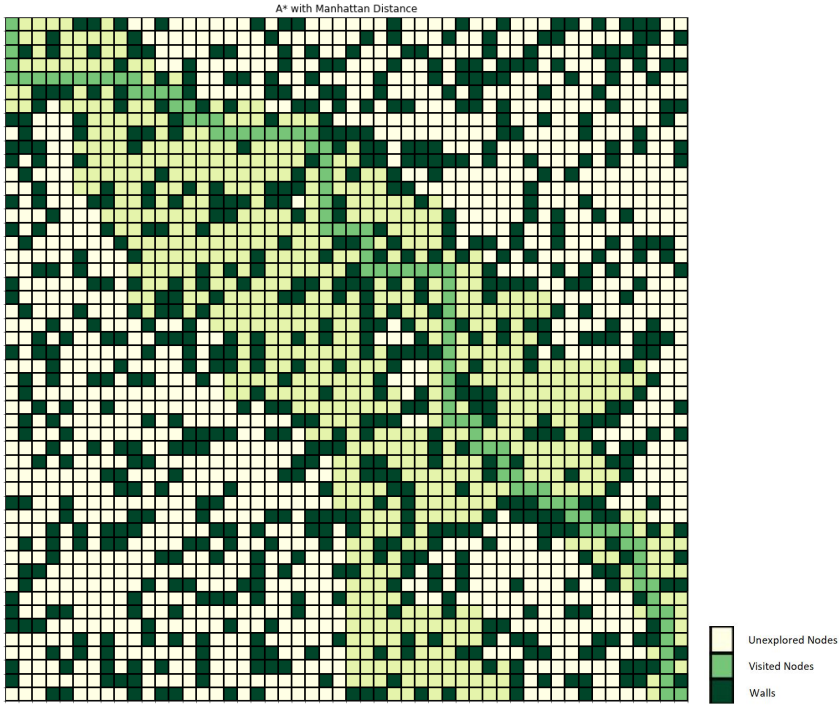


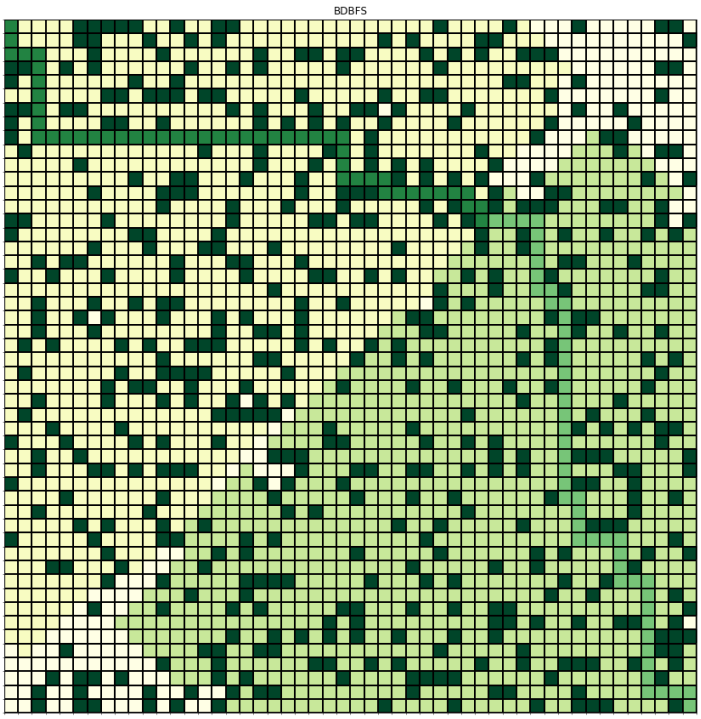


A* with Euclidean Distance



A* with Manhattan Distance



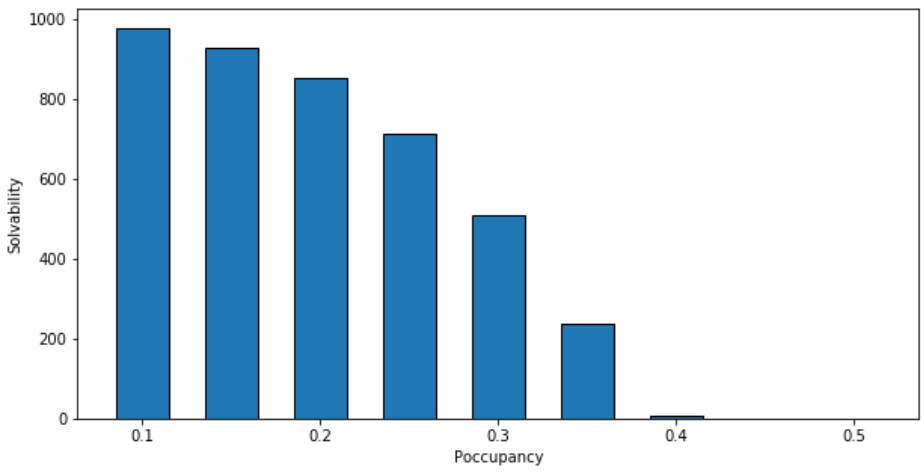


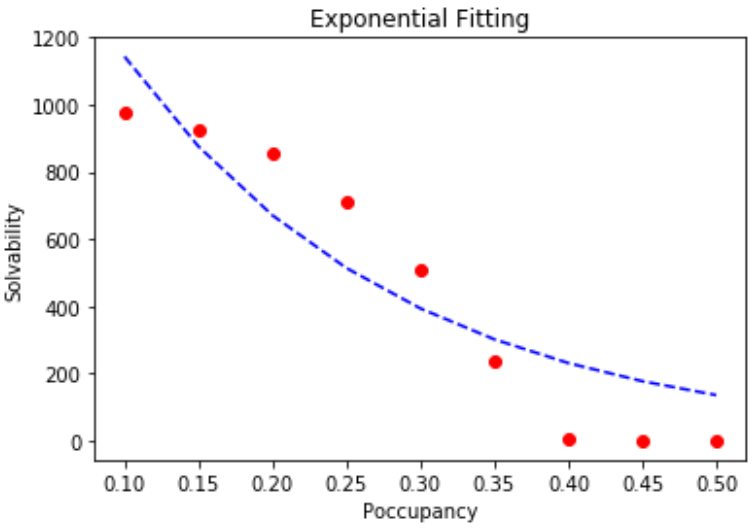
Q2) c)

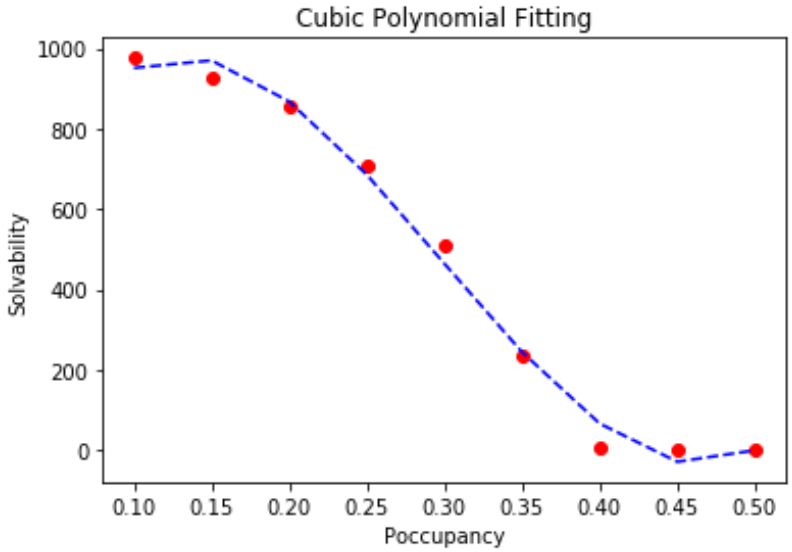
Maze Solvability decreases as the probability of occupancy ($Poccuapancy$) goes up. We ran **1000 simulations** for each for each $Poccuapancy$ values in the range of [0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5] to calculate the Po value.

We used DFS to check for solvability for the following reasons:

- DFS runs in the shortest amount of time.
- DFS has the lowest memory footprint (compared to all 5 algorithms) as it expands the minimum number of nodes. (Although memory was not a constraint, but in order to faster run the simulations we parallelized the runs, hence a lower memory footprint was preferred
- We can clearly see from the Bar plot that Solvability goes to 0 for $Poccuapancy$ > 0.4







- In order to calculate the Po value,
  we've used curve fitting to get the relationship between **solvability** vs ***Poccuapancy***. We tried fitting **polynomial** and **exponential** curves to this data.
- The Polynomial curve better fits the data. A Cubic Polynomial fitting is shown in the second image.

$$\text{Solvability} = c0+c1*p+c2*p2+c3*p3$$

$c0=55680.47189819$, $c1=-50413.77758749$, $c2=10616.46141576$, $c3=330.30322192$

Solvability = 50.9% $Poccuapancy$ = 0.3,

Solvability = 60.2% $Poccuapancy$ = 0.28,

It is clear from this graph that solvability falls rapidly, for $Poccuapancy$ > 0.3 and more than 50% of mazes are solvable for $Poccuapancy$ <= 0.3. Hence,

***Po* = 0.3**

Q2) d)

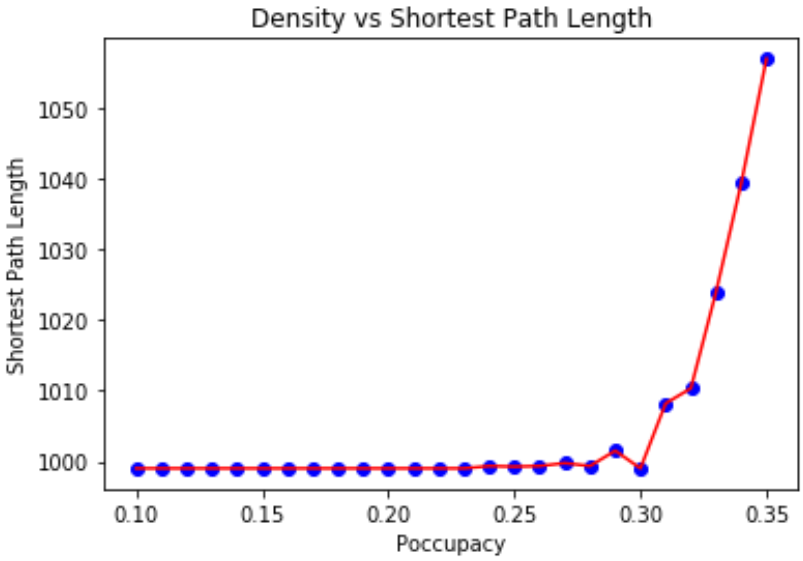
Dim = 500 X 500

Simulation count = 1000

Poccuapancy = [0.1, 0.35]

Algorithm = A* with Manhattan Distance as heuristic

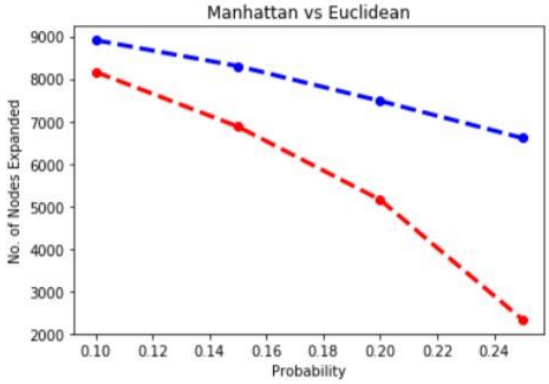
- We used A*, because A* with an optimistic heuristic will always return an optimal path.
- A* is more memory efficient as it expands lesser nodes than BFS and also runs pretty quickly.
- Manhattan Distance was used because Manhattan is an admissible heuristic and needs relatively less computation as compared to Euclid.

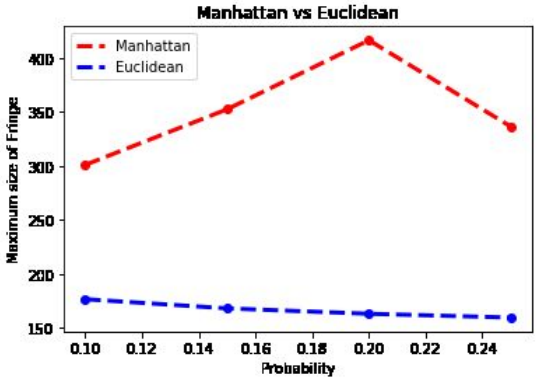


Q2) e)

As we can see from the graphs below that the fringe length of A* is greater than Manhattan and one of the main reasons why this happens is 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, Manhattan can give us better results than Euclidean as it explores the least number of nodes as shown in the graph below.





Q2) f)

Observations :

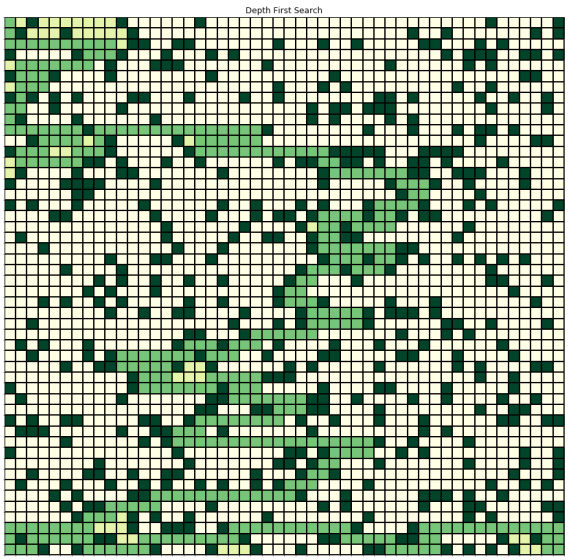
1. The length of the path taken by DFS is longest compared to the others

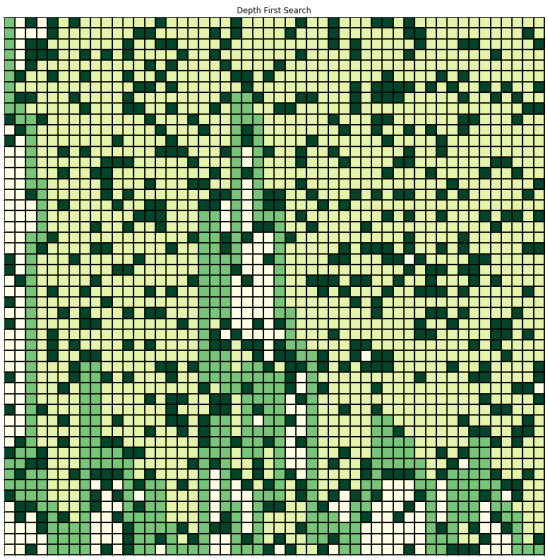
2. BFS explores all the nodes in the graph, as it traverses in a level by level manner.
3. The path taken by A* with euclidean distance tries to be as close to the diagonal from start to goal irrespective of the order of moves.
4. A* with Manhattan expands far lesser nodes as compared to Euclidean.

Based on these observations, and visualizations shown in part b) the algorithms do behave as they should.

Q2) g)

The answer is Yes, We can improve the performance of the algorithm by changing the order of moves we take, or the nodes we load in our fringe. We experimented with a different fashion of moves and found out that a particular combination reduced the number of nodes expanded and also the maximum fringe length by more than half, whereas other combinations doubled the number of nodes explored and fringe length of the algorithm.





The figure on the left has the order of Up, Down, Right, Left whereas the second Figure has the order of moves as Left, Right, Up, Down, which clearly shows the benefit of exploring the nodes in a proper fashion.

Hence we can conclude that looking at the neighbors which are below and to the right of the runner are worth looking at before exploring any other nodes.

Q2) h)

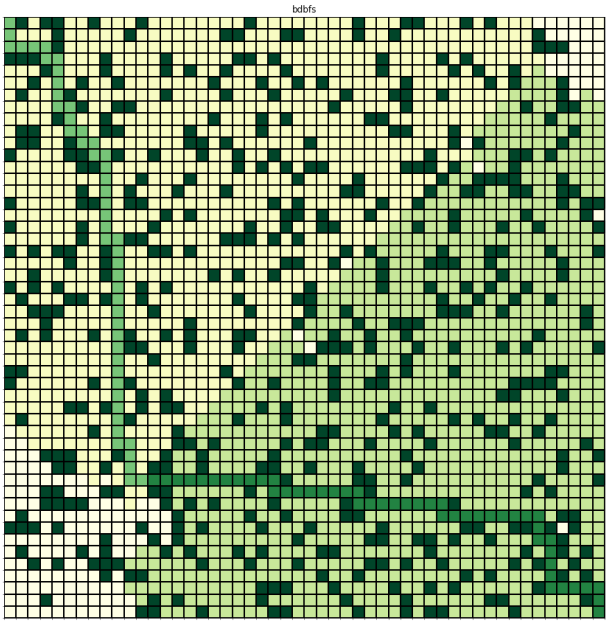
BFS, and by extension BDBFS in general, expands more nodes in order to cover all the levels possible in a tree and it will search nodes closest to the origin first whereas A* will always expand fewer nodes with an admissible heuristic and will never exceed the minimum cost to cover a certain path from the start to a node in the tree. H will always underestimate the remaining cost to the goal.

$$H(n) \leq C^*(n,G)$$

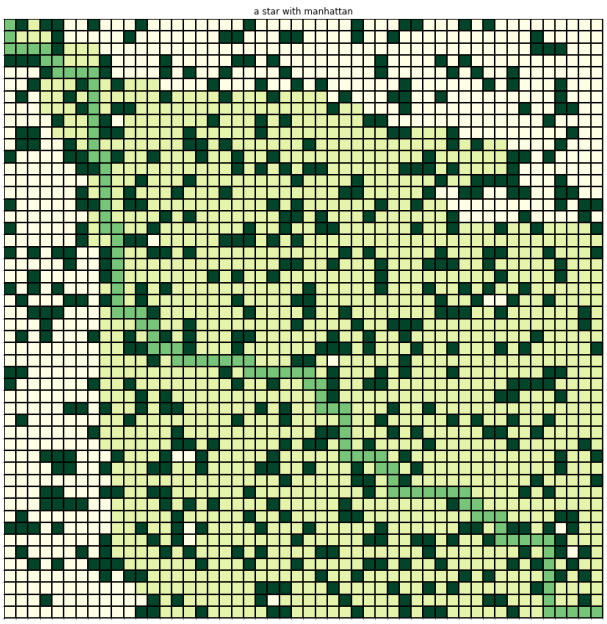
Consider the start node n,

Estimated cost from n to goal < estimated cost from n to goal through n'

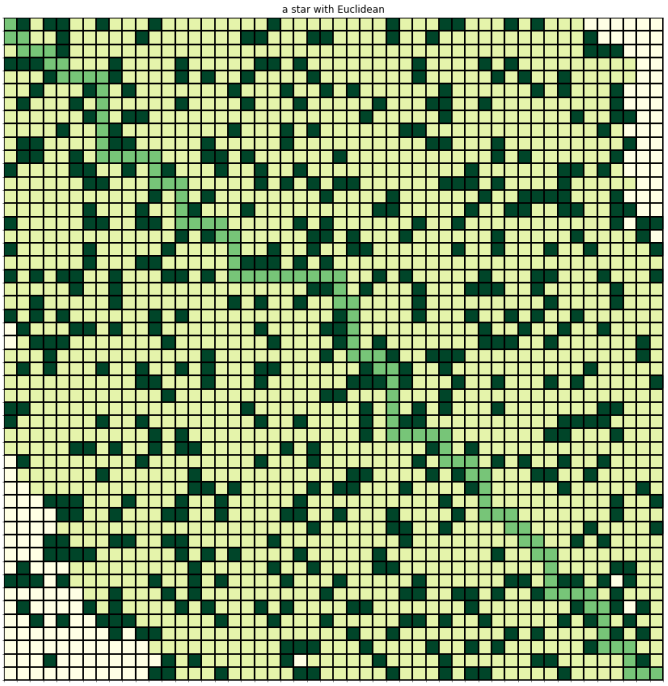
As we can see from the figures below that BDBFS is exploring nodes that are not explored by A* with Manhattan heuristic as well as A* with Euclidean heuristic.Hence we can say that Yes, BDBFS does expand nodes that are not explored by A*.



Bidirectional BFS



A* with Manhattan



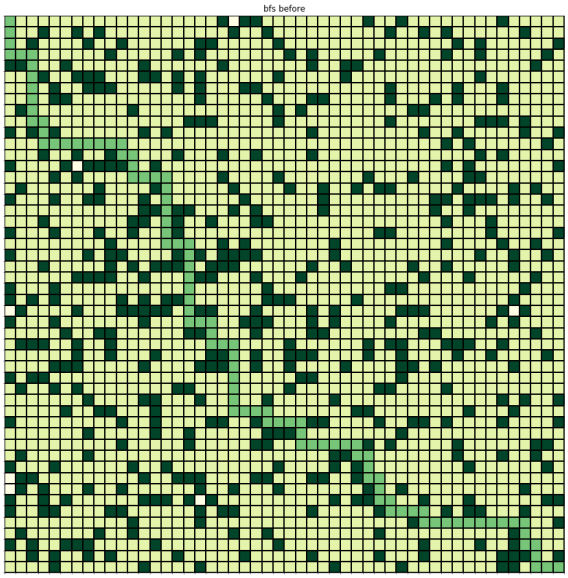
A* with Euclidean Heuristic

.

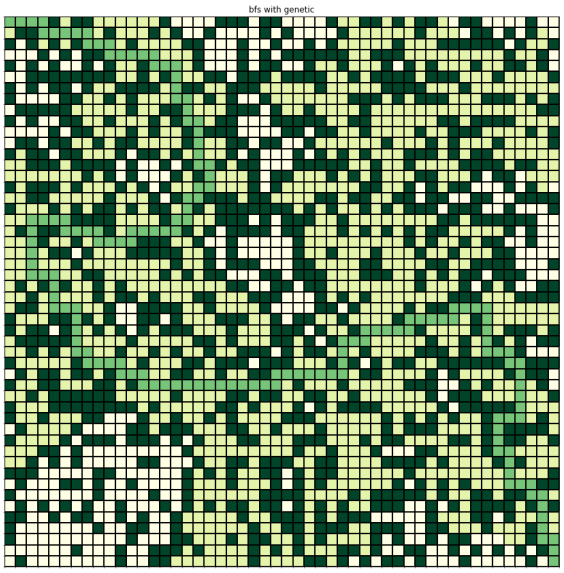
Q3) a)

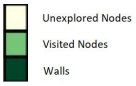
In this part, we chose to go with a Genetic approach as our Local Search Algorithm for finding the hardest mazes using 50 Generations of Populations. We first generated a set of mazes and sorted them by the most Fit Maze by the shortest path from the Start to the Goal node as a condition for being the fittest maze. The parents were generated out of this population by randomly selecting a pair out of the population of the mazes. These Mothers and Fathers were used for generating a set of mutated children by a crossover between the 1st and 2nd halves of the two which were again added to the population of mazes out of which we chose the best ones by trying to find the paths using BFS. For the Mutation, we randomly introduced walls in our maze and increased the difficulty of solving the maze. Eventually, after the crossover and mutation, we get the hardest maze.

BFS before Genetic Algorithm



BFS after Genetic Algorithm





Q3) b)

Termination Conditions:

- Since it is questionable when we have found the hardest maze, the termination conditions we thought of were related to the number of Generations loaded and the percentage of children mutated in our Algorithm. Finding out that the DFS algorithm converges to a local optimum after a few generations even that can be considered as a termination state for the algorithm.
- We are also considering our termination state when at least 10% of the mutated children are not able to find any path to the Goal node.

- Our termination condition has one shortcoming that there is a possibility of the Algorithm getting converged in the first maze itself if it does not have a path.

Advantages:

Mutation of the children in the Genetic algorithm by randomly introducing walls in the maze gives us a better probability of stumbling upon a harder maze which can then be added back to our sample set of maze populations. This quality of Genetic Algorithms cannot be found in Hill climbing algorithms as well as a Beam Search which might get stuck in a Local Optima. One more advantage of genetic algorithms is that it can be parallelized easily onto multiple clusters.

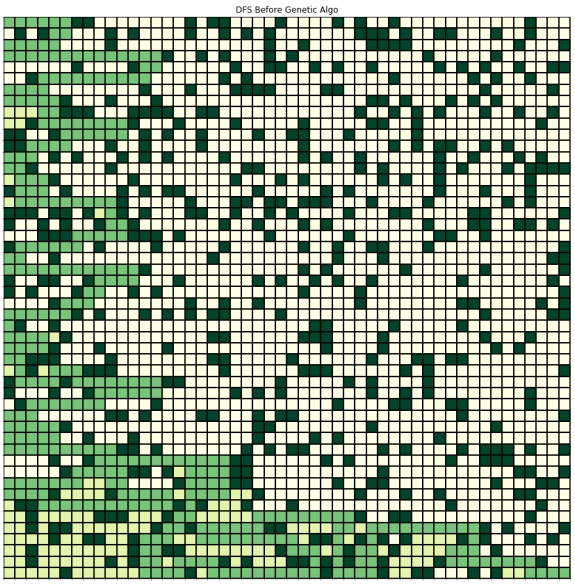
Shortcomings:

One shortcoming of using a Genetic approach is that there might be much harder mazes that we have not considered in our set as our entire universe is based on our first population set generated.

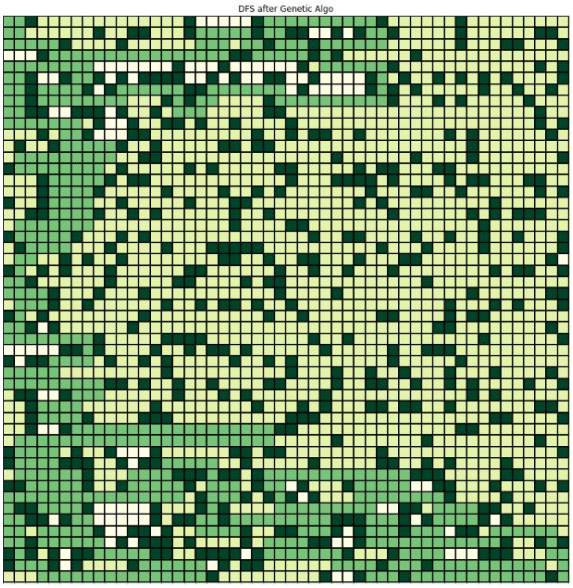
Q3 c)

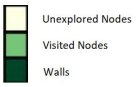
The Hardest maze generated using the paired metrics are as follows:

DFS with Maximal Fringe size before Genetic:

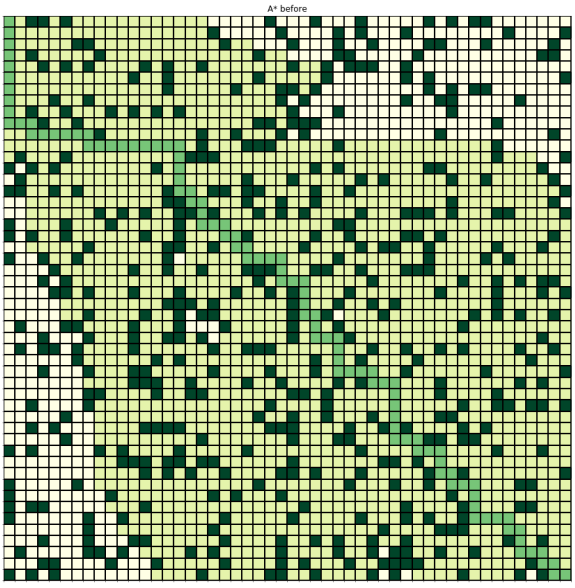


DFS with Maximal Fringe size after Genetic:

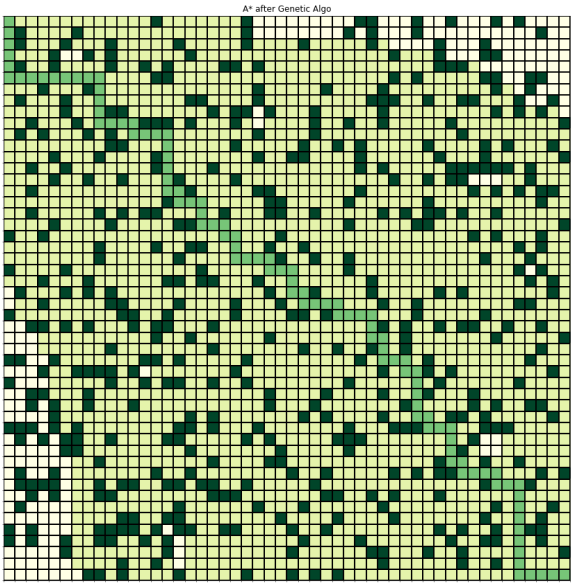


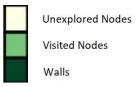


A* Manhattan before Genetic Algo:



A* Manhattan after Genetic Algo:





Q3 d)

- We notice that as the walls increase in the mazes because of the mutations and crossovers from the parents the mazes get harder and harder to solve

- For DFS with maximal fringe size and A* with maximal nodes visited , we expected the number of walls to increase as the generation increase but we did not see that happening and the number of walls remained more or less the same

Q4)

dim = 100 X 100

q_range = [0.05, 1.0], with a step of 0.05

Simulation Count = 500 for all 20 values of q

We have used a smaller dimension for this problem for 2 reasons:

- A lot of the mazes generated were either unsolvable or fire did not have a path to bottom left corner, hence to enable us to run a large number of simulations in reasonable time, we've used a smaller dimension of 100 X 100 to make it possible for us to manually analyze the runs.
- In the following graph, the graph higher up is the baseline, wherein we first calculated the shortest path using A* with Manhattan distance. Then the maze was set on fire as described in the Problem and the runner was made to follow this calculated shortest path and the % mazes where runner was able to reach the destination are plotted.

Dynamic Algorithm: Problem Breakdown

- The runner at each step must choose a path / neighbor and walk on it.
- There will be no fringe, for this algorithm, as we aren't tracking multiple paths. And If the runner moves back on the path, it's not backtracking in time, but a new step that the runner took and the propagated fire remains as it is.
- We are still maintaining a closed set, which will be explained in the algorithm below.

Algorithm

- We chose to perceive this problem as a 0-sum game between our champion, the Runner and a chance player, the Fire.

- From this perspective, we chose to try an **adversarial search** like approach wherein the cells which are on fire or have some probability of being on fire in the next step, give you a negative utility, and this utility is back propagated to the current neighbors of the runner to help it identify the next best path.
- We still want it to reach the destination as soon as possible, so we also go back to our heuristics, which help us do that - namely Manhattan and Euclidean.
- Since, the runner cannot look ahead to all the levels of neighbors, for 2 obvious reasons:
- Firstly, it'll be unfair to our chance player, the Fire.
- Secondly, it would be computationally infeasible to calculate the utility of nodes, all the way to the bottom of the tree, at every step
- We decided to stick with small values of Tree Look Ahead factor, a maximum of 5 levels

Total Utility = Distance_from_Goal(Minimise) + Fire utility(Maximise - stay away from fire)

Runner -> Min Player

Fire Utility: As the runner gets closer to fire, this value increases, and our min player (the runner) wants to keep the total utility at a minimum, so it'll run away from the fire. This fire utility is nothing but the rate of fire spread scaled down by the level at which we found fire and scaled up-to the distance value.

Fire utility = 1 - (1 - q)^k * (distance / level)

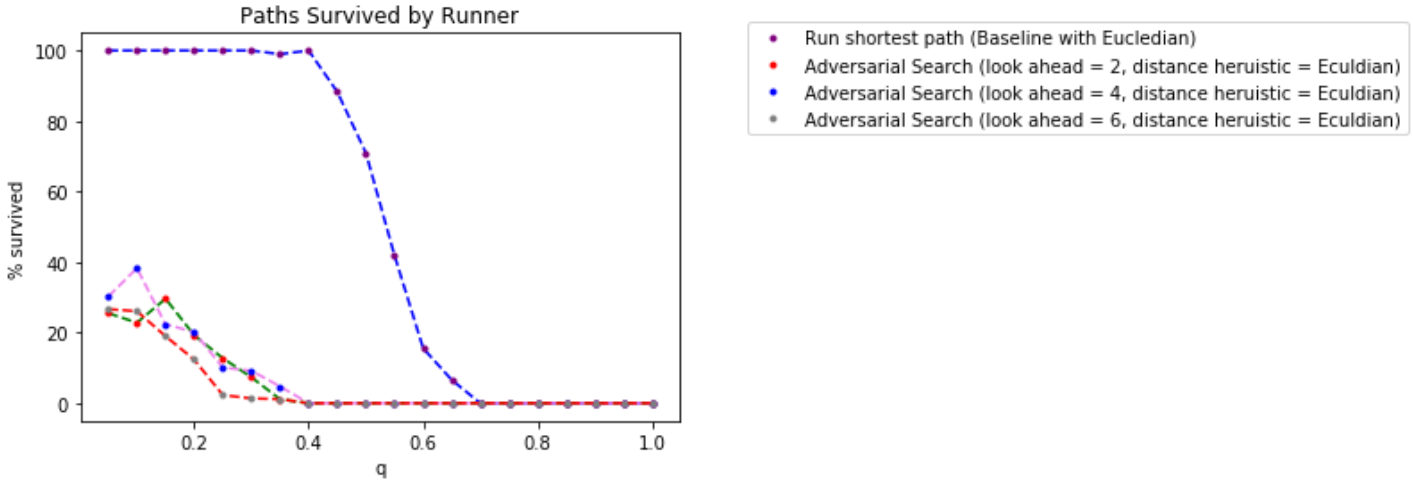
Total Utility = Distance(1 + (1 - (1 - q)^k) / level)

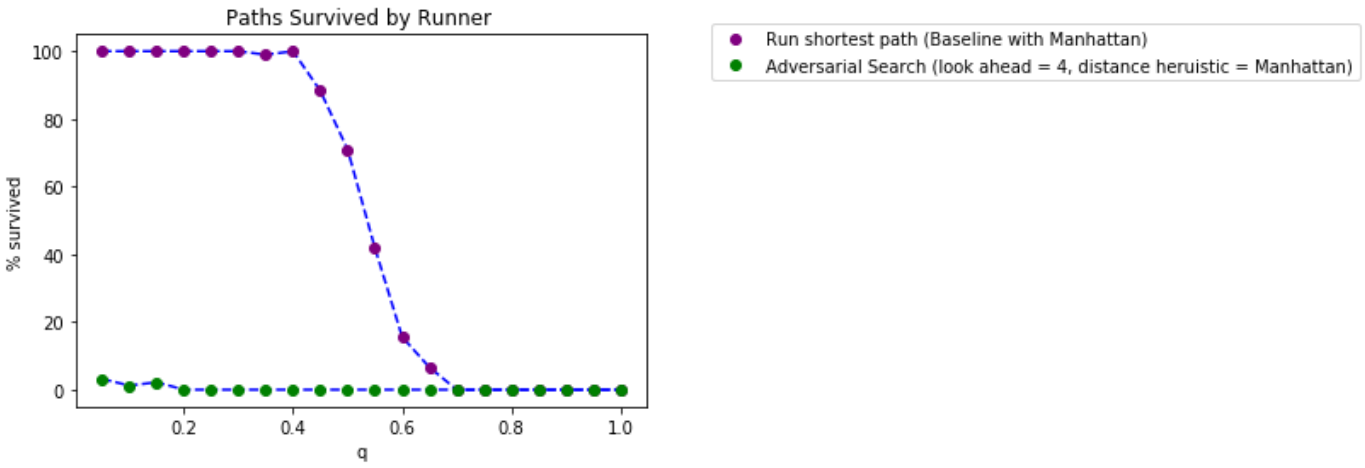
Closed Set:

We observed that in certain cases the, runner was getting trapped in wells surrounded by walls and got stuck there until fire burns. So, we are maintaining a map of visited nodes vs their count. And have put a limitation that the runner cannot visit the same node more than 2 times, to make the runner come out of wells and try other paths.

Distance Heuristic:

At the bottom you can see two plots, one using the Euclidean and other one with Manhattan for the distance heuristic, along with different look ahead levels to calculate the fire utility.





Inference:

- Euclidean tries to stick to the diagonal path (shortest path), and has a better chance of crossing before the fire catches up.
- With Manhattan on the other head, runner gives preference to either moving right or moving down depending on the order of moves and will more often than not end up getting stuck on a well of walls. It takes significant amount of time for the runner to come out of this well. By this time the fire is already spread out and the runner is unable to solve the maze. Hence, Manhattan is not a good distance heuristic for this problem.
- Although it would seem that higher look ahead values would give rise to more solutions, but that does not appear to be the case, as the solvability falls for a look ahead = 6. It appears that, after a point further look ahead is not too helpful.
- For q >= 0.4, the fire spreads too rapidly, and since fire has a much shorter path to the destination node (dim - 1, dim - 1) , the probability that the runner will reach the destination becomes starts to fall rapidly. This trend can be inferred from both the baseline and adversarial searches.