Maze-Runner

Part 1: Analysis and Comparison

Q1) 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?

Answer: In order to find a large enough map size, we generate a random map of dimensions 500, 1000 and 2000, and test it against p values for 0.1 to 0.3 (because majority maps are unsolvable for p > 0.3). For each pair of dimension x p value, we calculate the maximum number of nodes explored and maximum amount of time taken to solve the maze.

Depth First Search

P value	Dimension	Time Taken (ms)	Nodes Explored
0.1	1000	31.0	2771
	3000	1727	3555529
	5000	5483	1121729
0.2	1000	427	89335
	3000	3539	712790
	5000	8869	1786445
0.3	1000	3305	688913
	3000	30675	6193048
	5000	87184	17198825

Breadth First Search

P value	Dimension	Time Taken (ms)	Nodes Explored
0.1	500	693	225224
	1000	2887	900177
	2000	12065	3599977
0.2	500	600	199649
	1000	2553	798470
	2000	10642	3193282
0.3	500	497	172515
	1000	2122	687250
	2000	8969	2754841

Euclidean A*

P value	Dimension	Time Taken (ms)	Nodes Explored
0.1	500	1094	223657
	1000	4863	895408
	2000	21092	358092
0.2	500	958	193185
	1000	4202	773659
	2000	17691	3100944
0.3	500	830	171777
	1000	3562	687868
	2000	14854	2751627

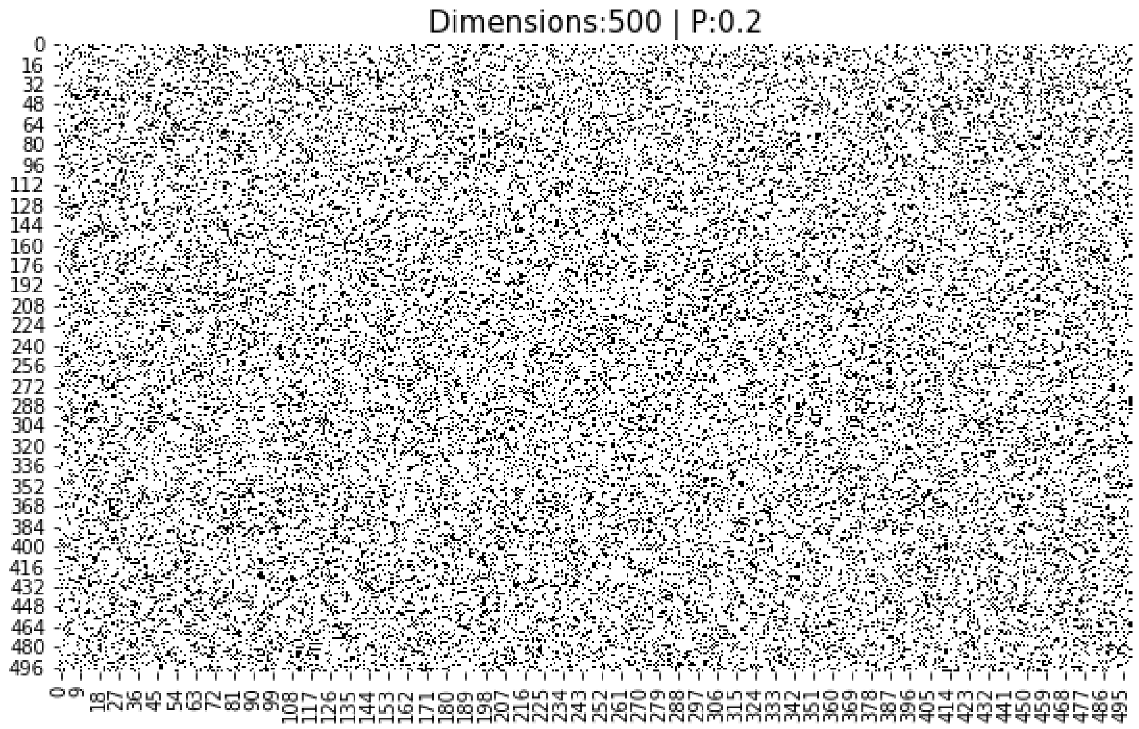
Manhattan A*

P value	Dimension	Time Taken (ms)	Nodes Explored
0.1	500	1088	199511
	1000	4740	780094
	2000	21545	3158889
0.2	500	693	126314
	1000	3166	513909
	2000	16002	1916218
0.3	500	887	171660
	1000	3795	688997
	2000	16171	2752230

As evident by the data, almost all the mazes of dimension 500 explore nodes in the order of 10^6, and can be solved in well under 2 seconds. Thus, we will use 500 as a dim, so that we can generate reproducible results, in a reasonable amount of time.

Q2) 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?

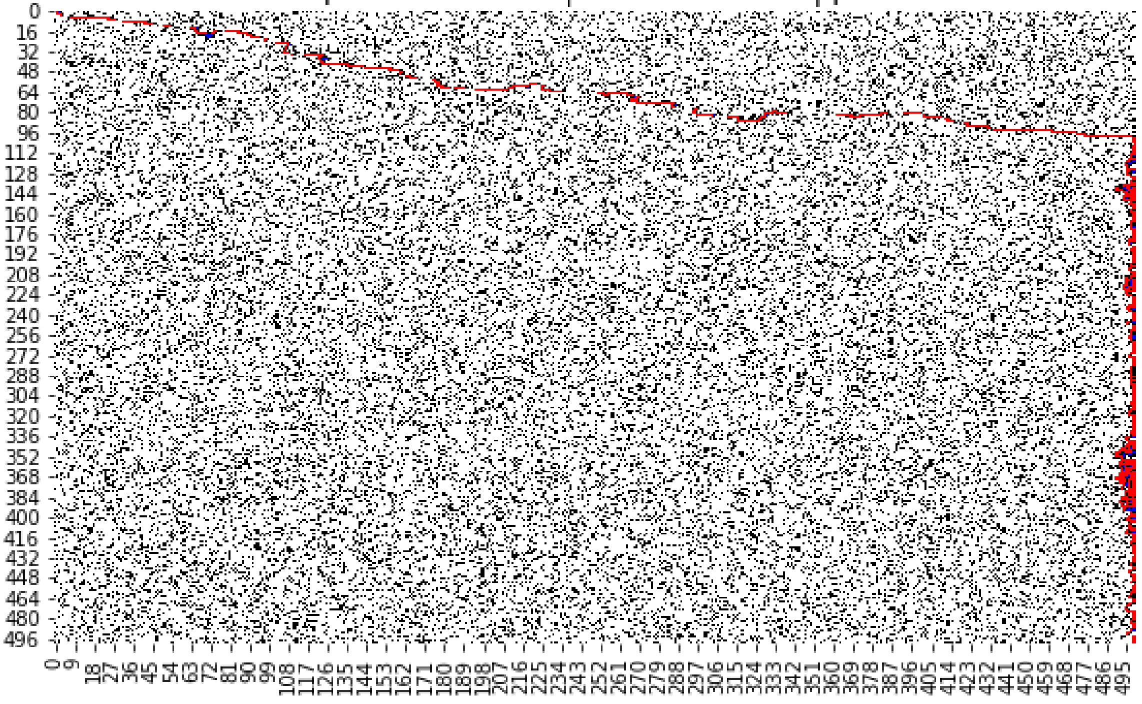
Answer:



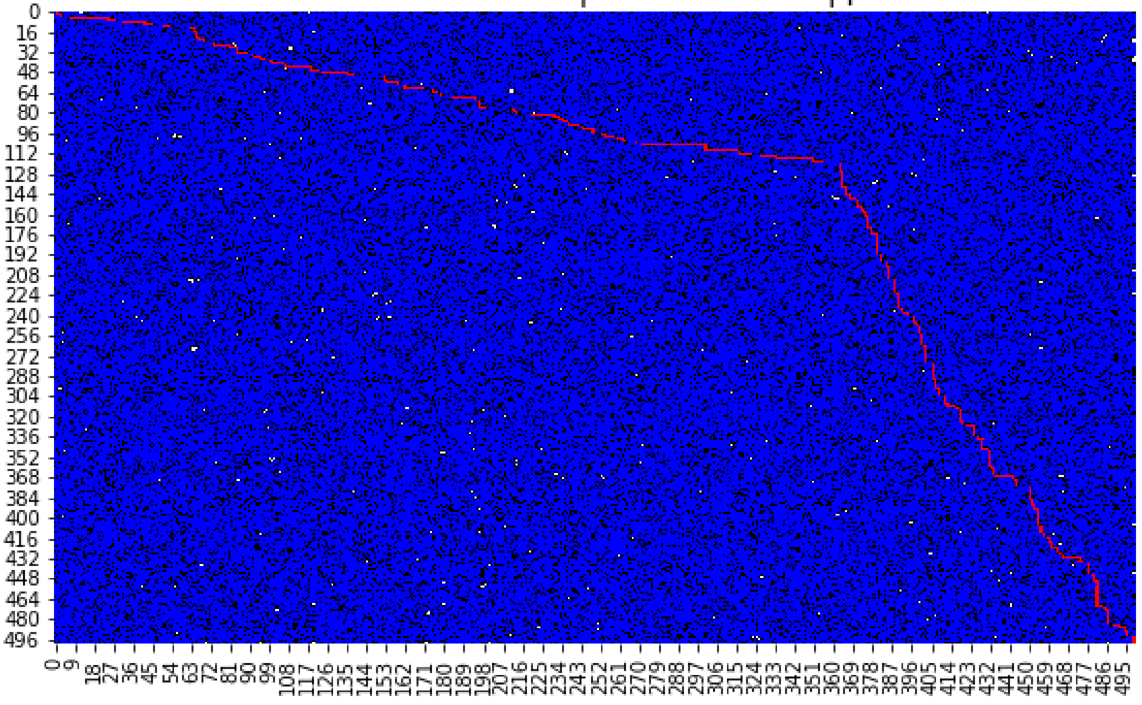
We generate a random maze with 500 dimensions and p = 0.2, then we solve the maze using each of the four algorithms.

- White indicates free node
- Black indicates blocked node
- Red indicates the solution
- Blue indicates the nodes explored

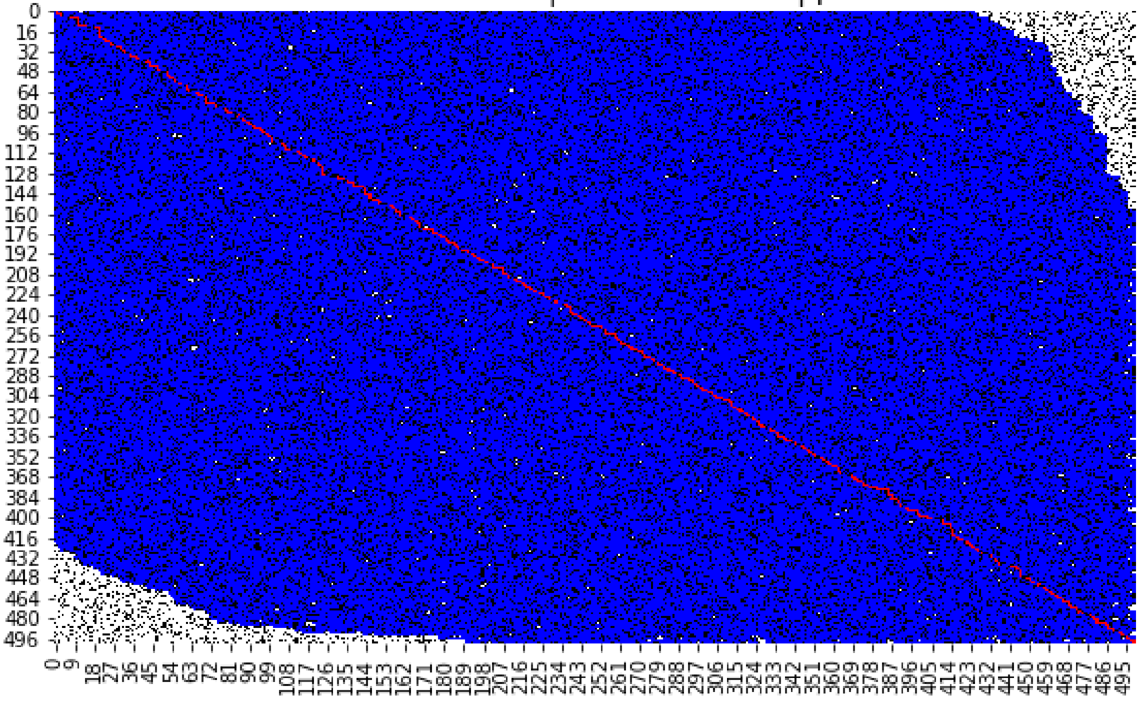
Depth First Search | Dimension:500 | p:0.2



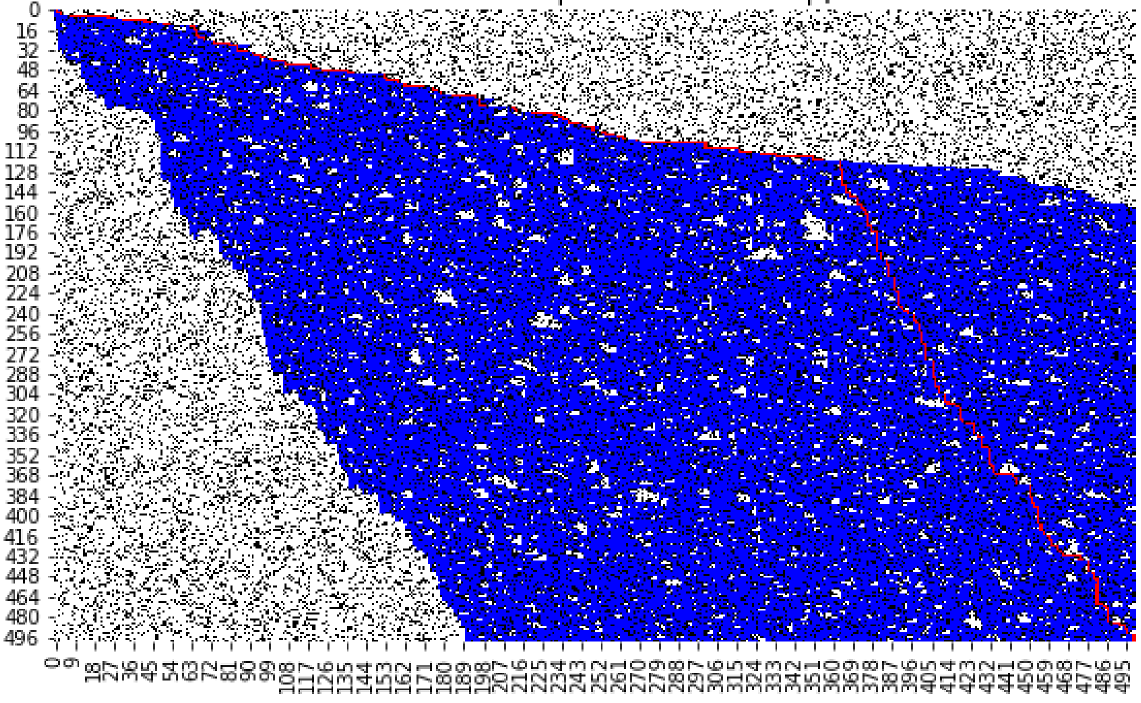
Breadth First Search | Dimension:500 | p:0.4



A-star euclidean | Dimension:500 | p:0.2

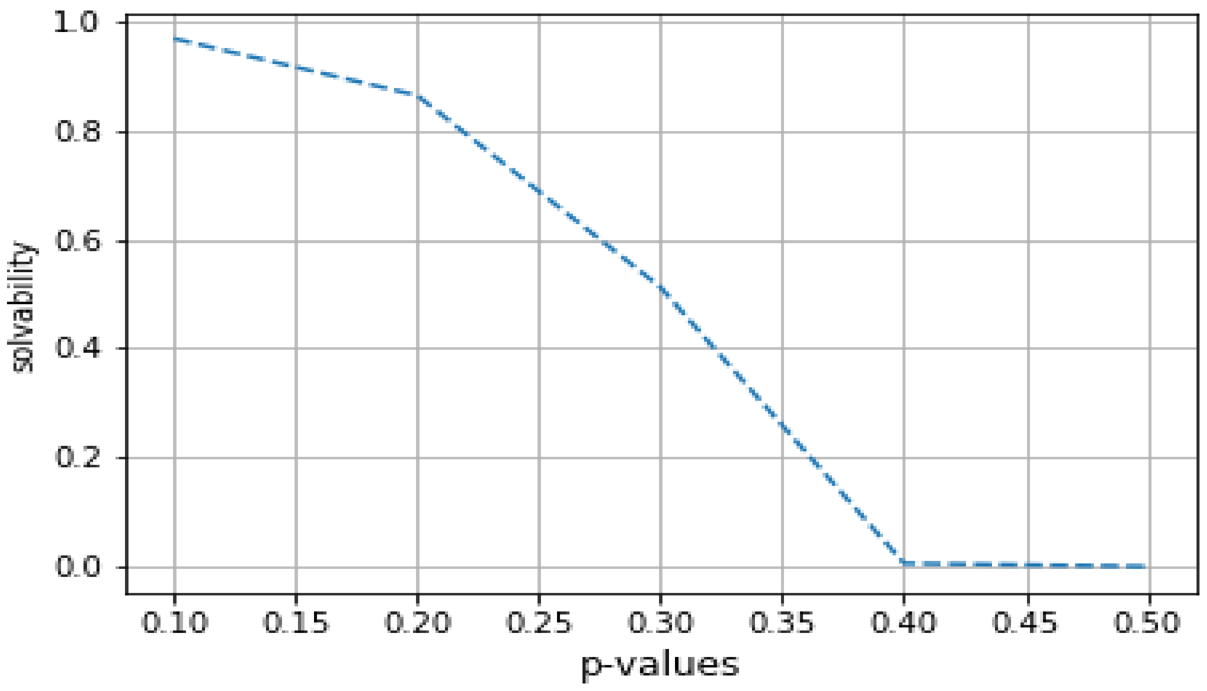


A-star Manhattan | Dimension:500 | p:0.2



<u>Q3)</u> 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 p_0 where for $p < p_0$, most mazes are solvable, but $p > p_0$, most mazes are not solvable.

<u>Answer:</u> We conducted 500 trials with randomly generated maze of 500 dimension and tested it against p values ranging for 0.1 to 0.5



As seen by the graph, for p values greater than 0.35, most mazes are unsolvable, and for p values greater than 0.5, the mazes are unsolvable. As a result, $p_0 = 0.35$.

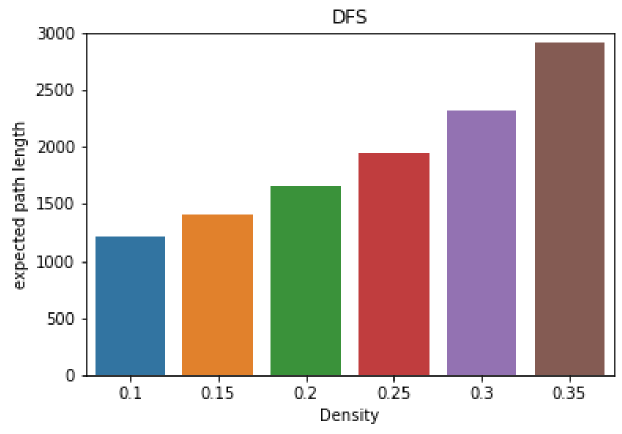
To test for solvability, we use Depth First Search (DFS), because its complete, i.e. if a path exists, it will find a path in the shortest amount of time. The path returned by the DFS is not optimal, but since we are only checking for solvability, we don't really care how long the path is.

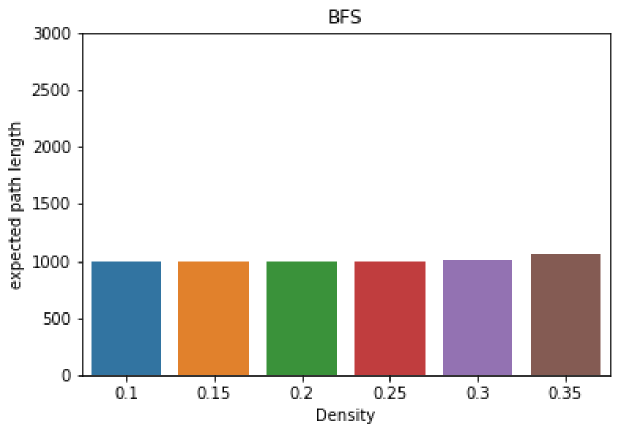
<u>Q4)</u> For p in [0, p_0] as above, estimate the average or expected length of the shortest path from start to goal. You may discard unsolvable maps. Plot density vs expected shortest path length. What algorithm is most useful here?

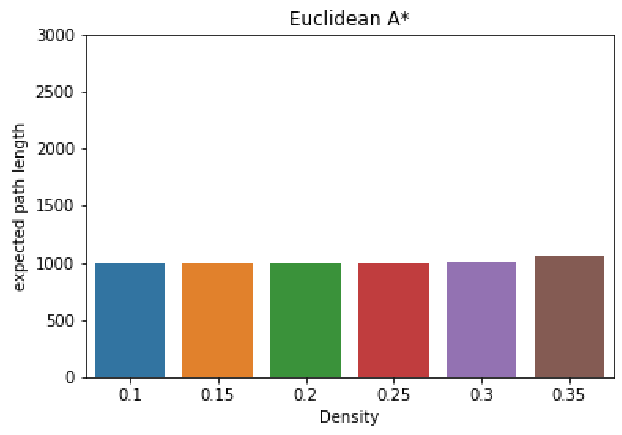
<u>Answer:</u> We ran 10 trials on a map of 500 dimensions for each algorithm and averaged the results.

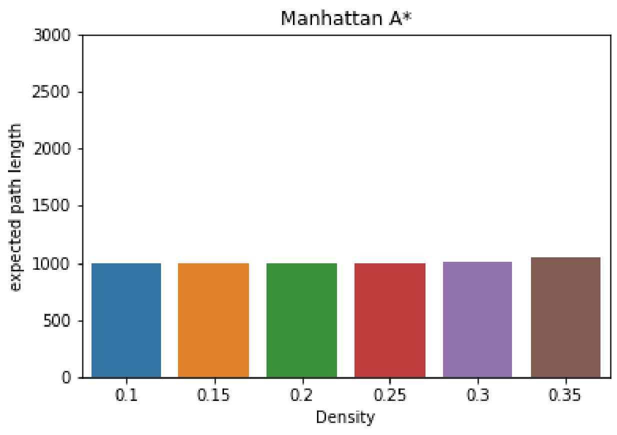
P Value	**DFS**	**BFS**	**A* Euclidean**	**A* Manhattan**
0.1	1218.3	999	999	999
0.15	1411.1	999	999	999
0.2	1650.9	999	999	999
0.25	1947	999.2	1000.1	999
0.3	2323.9	1004.6	1007.6	1001.6
0.35	2913.8	1061	1058.2	1052

The expected path length for BFS and A* Algorithms, is quite similar. BFS gives the optimal path, and so does A* algorithm, as long as it uses a heuristic that is admissible, i.e. it always underestimates the path to the goal node. However, they both explores more number of nodes, and thus take more time for execution. DFS on the other hand, explores fewer number of nodes, but doesn't give the optimal path length.





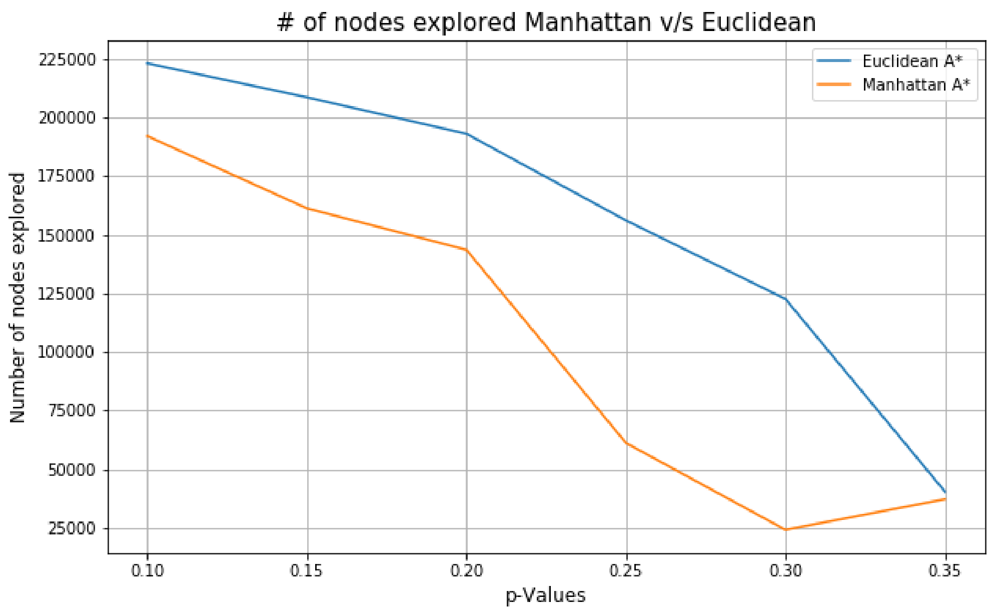




<u>Q5)</u> Is one heuristic uniformly better than the other for running A∗ ? How can they be compared? Plot the relevant data and justify your conclusions?

<u>Answer:</u> In order to compare Euclidean A* to Manhattan A*, we will compare the number of nodes explored in their pursuit of the path. Number of dimensions is 500

	0.1	**0.15**	**0.2**	**0.25**	**0.3**	**0.35**
Euclidean	223144.4	208640	193112.2	156119.8	122578.7	40353.1
Manhattan	192106.4	161282.4	143586.8	61117.6	24050.9	37131.9

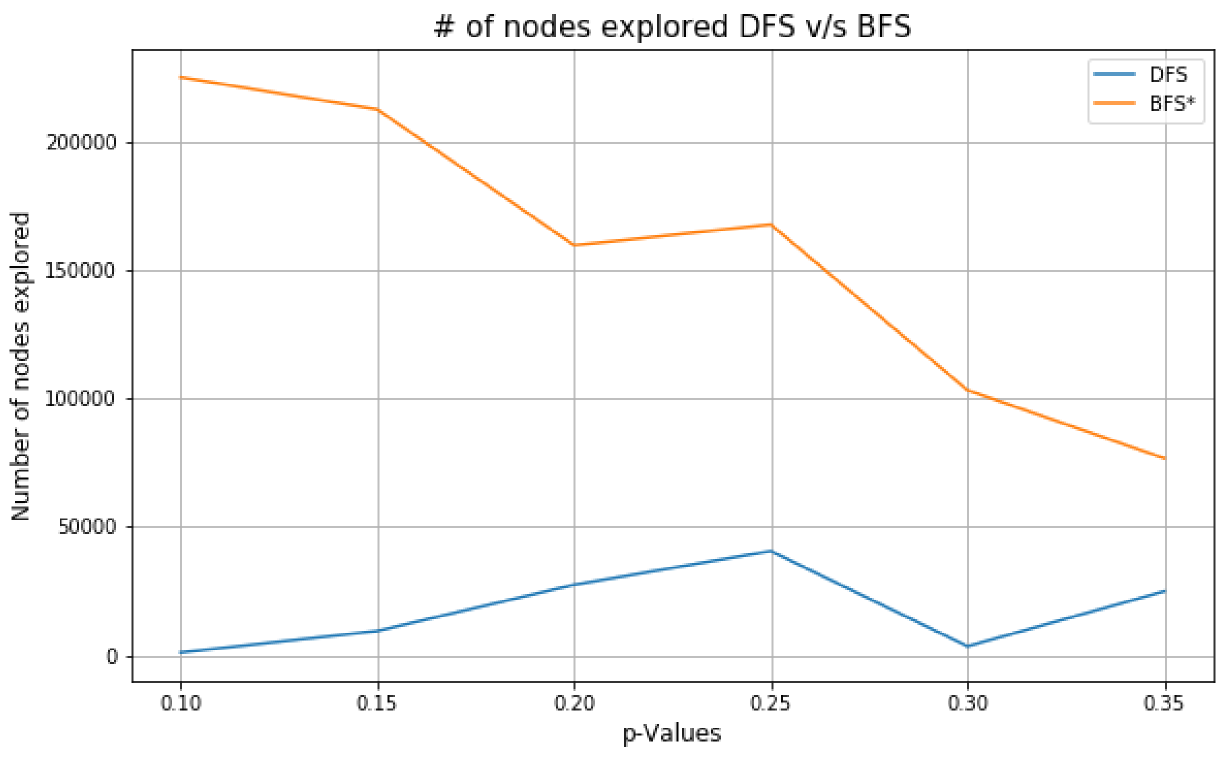


For initial values of p, the Manhattan heuristic explores lesser number of nodes than the Euclidean heuristic. The Manhattan heuristics explores nodes that are closer to the diagonal line whereas the Euclidean heuristic also explores nodes towards the top right and bottom left corners from the diagonals. As a result the Euclidean heuristic explores more nodes. As we move closer to the threshold probability, these algorithms start converging and the number of nodes explore are nearly the same.

Q6) If 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?

Answer: In order to compare DFS to BFS, we will compare the number of nodes explored in their pursuit of the path. Number of dimensions is 500.

	0.1	**0.15**	**0.2**	**0.25**	**0.3**	**0.35**
DFS	1092.5	9470.5	27453.1	40553.0	3524.4	24889.7
BFS	224949.1	212459.3	159569.4	167663.3	103180.9	76678.1



BFS results in the optimal path length, but it explores a lot more number of nodes than DFS, and as a result takes more time to return the path. Many a times , we are only interested in a path and not necessarily the optimal path. In such a scenario, it will be more prudent to use BFS rather than DFS.

- If we know that the solution is far from the root node in the search tree, then it will be better to use depth first search.
- But if we know the solution is not far from the root node, then BFS will be better.
- DFS has a lower memory overhead than BFS

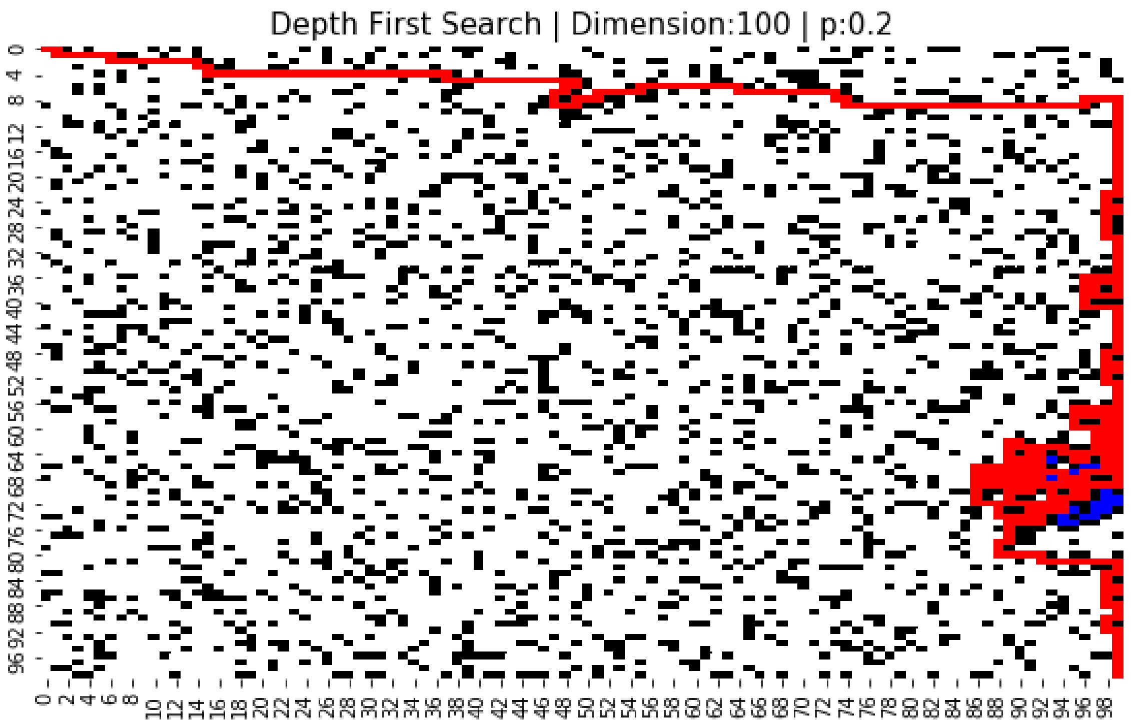
<u>Q7)</u> Do these algorithms behave as they should?

<u>Answer:</u>

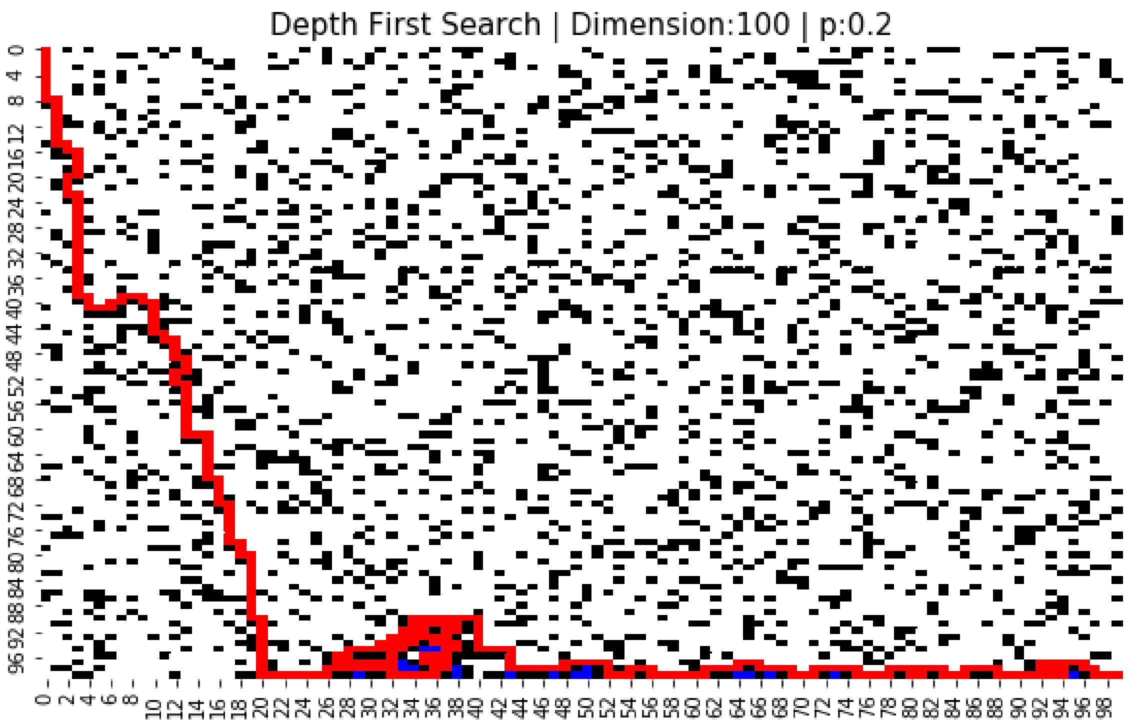
- **DFS:** DFS starts from the root node and explores as deep as possible, before backtracking. It uses a stack to find a path from root node to goal node. DFS is complete i.e. if a maze is solvable the it will return a path. However, this path will not be the most optimal one. While finding the path, it explores a fewer number of nodes , thus has a lower memory requirement.
- **BFS:** This algorithm finds all vertices that are one edge away from starting point, than all vertices that re two edges away and so on. BFS is complete and also results in an optimal path. However, it explores more number of nodes, than DFS, and thus takes longer to find the path.
- **A*:** This starts from the root node, and goes onto the next node, which it currently thinks is the closest to the root node based on some heuristic value. It is complete and optimal, as long as the heuristic s are admissible. An admissible heuristic constantly underestimates the pat to the goal node. It uses apriority queue to decide which node to process next. Based on the heuristic chosen, it explores the lesser number of nodes than BFS but greater than DFS.

<u>Q8)</u> 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.

<u>Answer:</u> Since DFS select a branch, and explores it as deep as possible, it is possible to improve its performance, by choosing the order of neighbouring nodes.



In the above figure, we have processed the nodes in the order Right->Bottom-Top->Left, and the path length that we found was 367.



Now in the same maze, we process the nodes on the order Bottom->Right->Top->Left and we got a path length of 293.

In general, since we are starting from the top left corner, and we want to reach the bottom right corner, we should prioritise the right and bottom nodes, so that we can get a path that is closer to the optimal one. If we prioritise the top or left node, then we will end up with a more convoluted oath and will also explore more nodes in the process.

Part 2 : Generating Hard Mazes

<u>Q1)</u> 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?

<u>Answer</u>: We have designed a genetic algorithm to generate hard mazes. Genetic algorithm has three iterative steps: **Selection, Crossover** and **Mutation**.

1) **Selection**: In this we randomly generate 100 maps of the desired dim and prob. We run our searching algorithm on these mazes and select the top mazes base on the criteria that we want to maximise.

2) **Crossover**: In crossover, we select pairs of the fittest mazes, extracted 50% of the mazes and merged them with each other.

3) **Mutation**: After the mazes have been crossover, we randomly select 25% of free nodes and convert them to blocked nodes.

For our crossover process, we merge the top half of our fittest maze with the bottom half of the least fittest maze, that we selected during our selection process. We continue the process, for our second most fittest and the second least fittest maze. Doing so, ensures uniform variability, such that each of the resulting maze are equally hard to solve.

Q2) 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?

Answer: In our search for the hardest maze we run this genetic algorithm for 100 iterations. Thus we generate the hardest maze, only after the number of iterations exceed 100. One obvious shortcoming of this approach is that there might be a chance to generate t a harder maze if we increase the number of iterations. However one advantage is that limiting the iterations to 100 generates a maze that is reasonably hard, and does not take very long to execute and thus has lower memory requirements. This is especially useful, if you want to generate reproducible and valid results in a short amount of time and understand the genetic algorithm form an academic pint of view

Q3)Try to find the hardest mazes for the following algorithms using the paired metric: –

a) DFS with Maximal Shortest Path

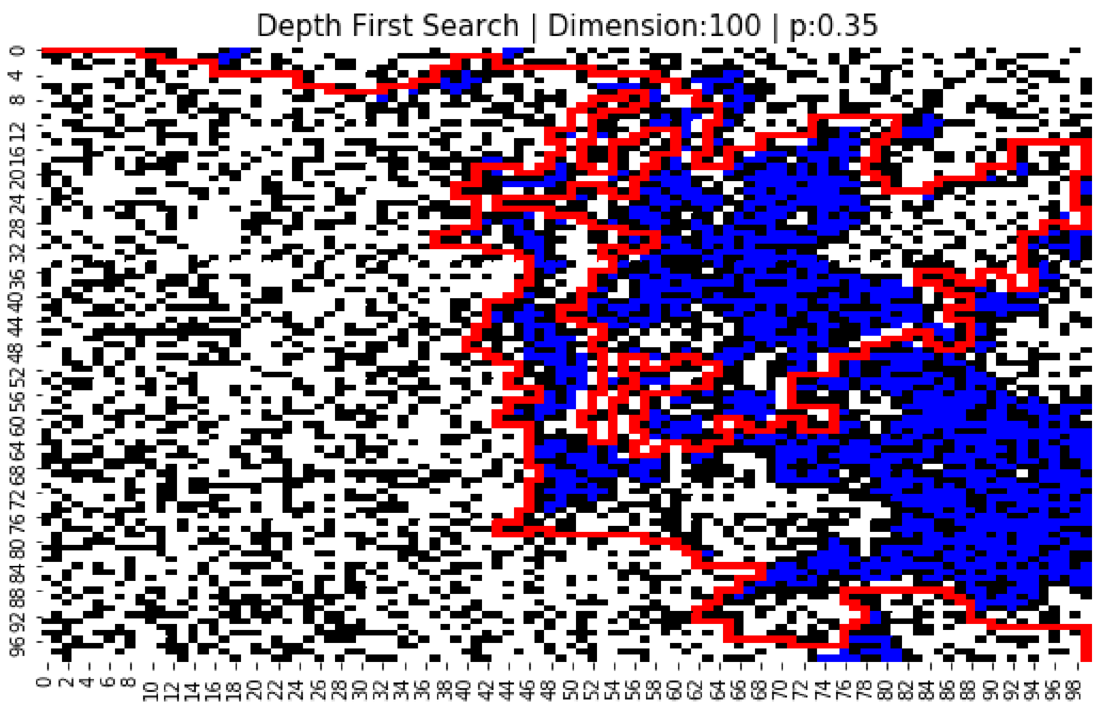
b) DFS with Maximal Fringe Size

c) A∗ -Manhattan with Maximal Nodes Expanded

d) A∗ -Manhattan with Maximal Fringe Size

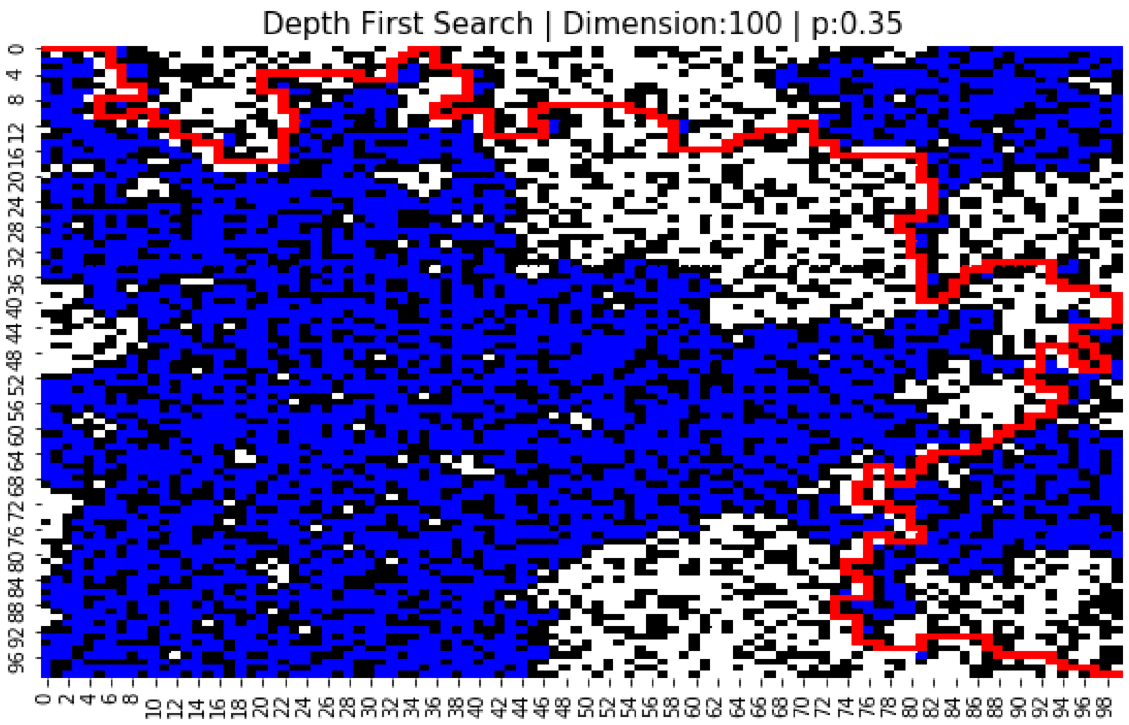
Answer: For each of the above questions, we generated the hardest mazes of dim = 100 and p = 0.35 by applying genetic algorithm for 100 iterations.

a) DFS with Maximal Shortest Path:



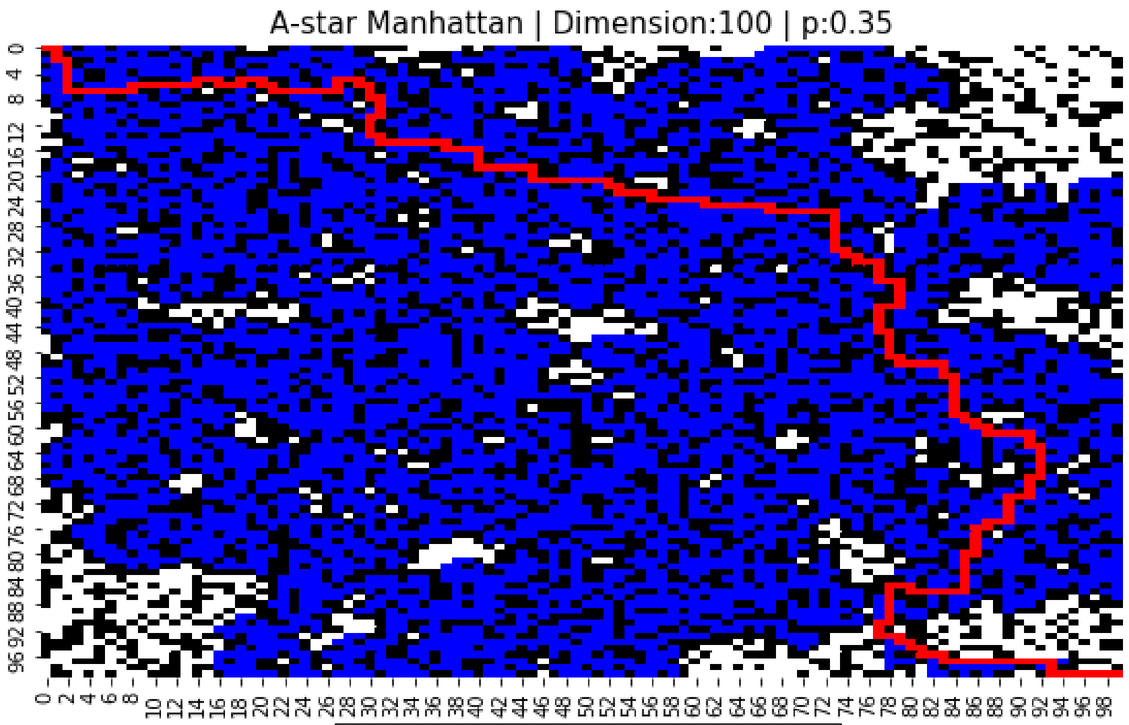
Max Path Length = 769

b) DFS with Maximal Fringe Size:



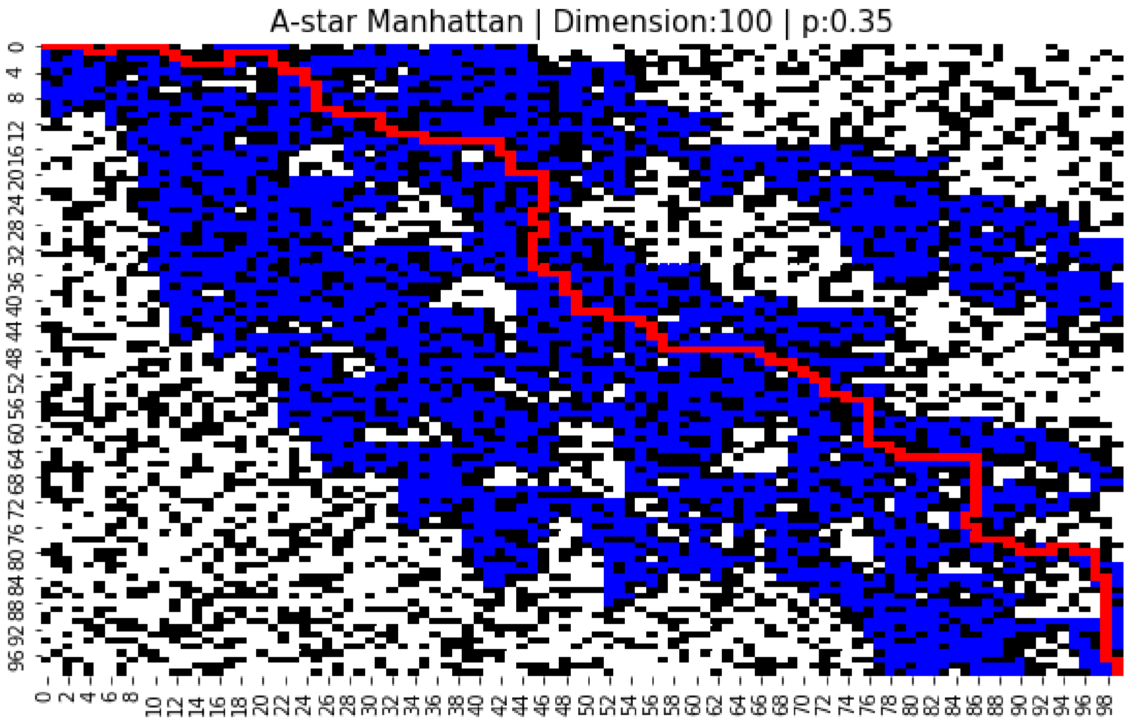
Max Fringe Size = 608

c) A∗ -Manhattan with Maximal Nodes Expanded:



Maximal Nodes Expanded = 5336

d) A∗ -Manhattan with Maximal Fringe Size



Maximal Fringe Size = 324