Maze Runner

Aditya Vyas, Vedant Choudhary, Nitin Reddy and Siddharth Sundararajan

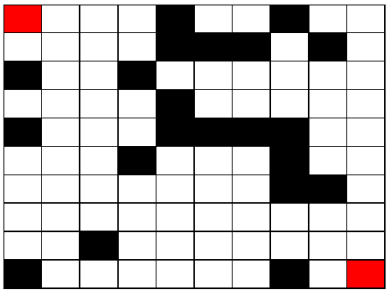
February 24, 2019

Abstract

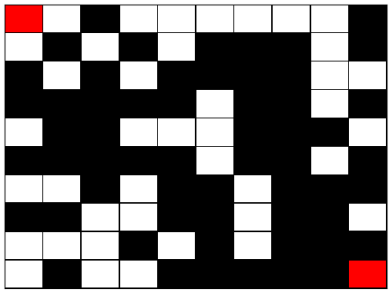
In this report, we explore various search algorithms such as Depth First Search, Breadth First Search etc., in a simple and complex way. This report is organized as follows. In section 1, we present results of paths found by the search algorithms. As part of Section 2, results from analysis and comparison of different algorithms are seen. In Section 3, our views and outcomes of generating hard mazes are observed. This is followed by presenting our approach to the problem of Thinning A^*. We conclude with our strategy to solve a maze on fire.

0.1 Performance of Different Search Algorithms

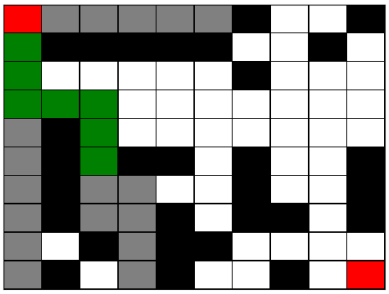
Our mazes are rendered as shown in the following images. The source and goal nodes are marked with red. A black cell represents an obstacle and a white cell represents an open cell, where the algorithm can reach.



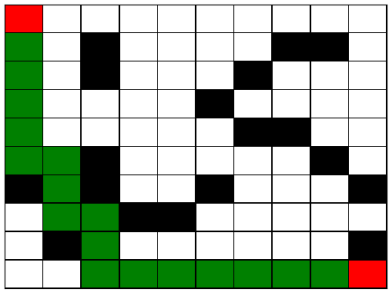
(a) A simple maze



(b) A very difficult maze



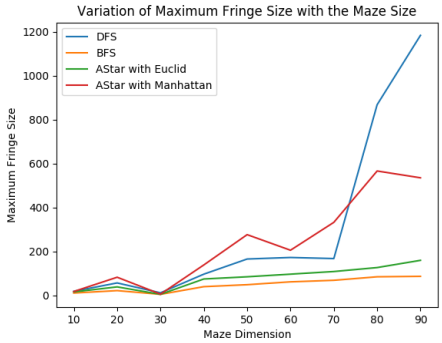
(c) A path finding algorithm searching through the maze



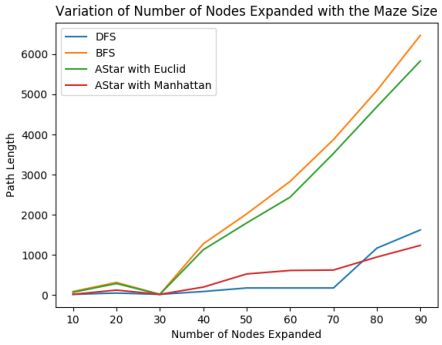
(d) Path found

Figure 1: Different Renderings of our Maze Environment

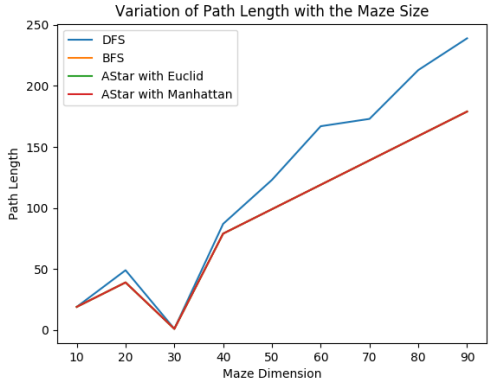
The number of obstacles is controlled by a probability value eg. In Fig-1(b), the probability was very high and so the majority of cells are obstacles. When our algorithm starts traversing the maze, its current path is highlighted in green as shown in Fig-1(c). Also, those cells which were already explored by the algorithm are highlighted in grey. Finally, when the algorithm reaches the goal, the rendering pauses for a few seconds to show the final path. In Fig-1(d), the DFS algorithm has already found the path in one go without backtracking.



(a) Maximum fringe size



(b) Number of nodes expanded



(c) Path length

Figure 2: Performance Metrics of the Algorithms

We now compare the performance of the path-searching algorithms based on 3 different metrics:

1. Path Length - From the above Fig-2(c), we see that the path length increases with the maze dimension for all the algorithms in general. However, DFS always has the longest path length than BFS and A-Star algorithms for any maze dimension. You only see 2 lines because the path lengths for BFS and A-Star algorithms is the same.

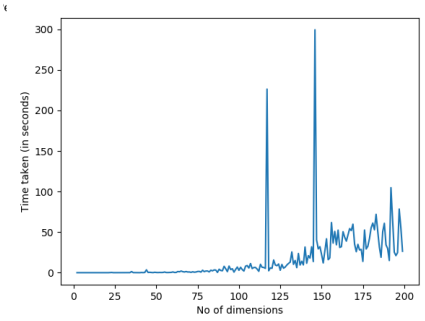
2. Maximum Fringe Size - From Fig-2(a), the maximum fringe size of BFS and A-Star-Euclid always stays lower than DFS and S-Star-Manhattan. The maximum fringe length is the maximum for A-Star-Manhattan upto maze size of 70 after which DFS takes over.

3. Number of Nodes Expanded - The number of nodes expanded is the highest for BFS and A-Star-Euclid because they find the optimal path and

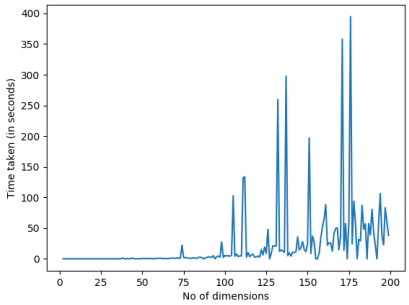
for this they have to visit all nodes. DFS has the least number of nodes expanded because it just goes into depth and finds any path, which is obviously not always the optimal path.

0.2 Analysis and Comparison of Different Search Algorithms

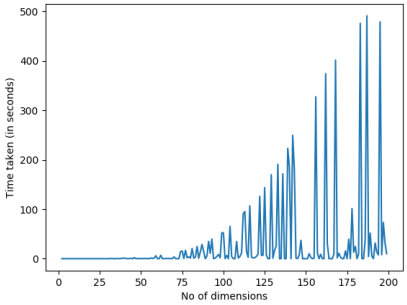
0.2.1 How do you a pick dim?



(a) Maze created with probability of 0.1



(b) Maze created with probability of 0.2



(c) Maze created with probability of 0.3

Figure 3: Performance of A* with Manhattan heuristic vs. Maze Dimension

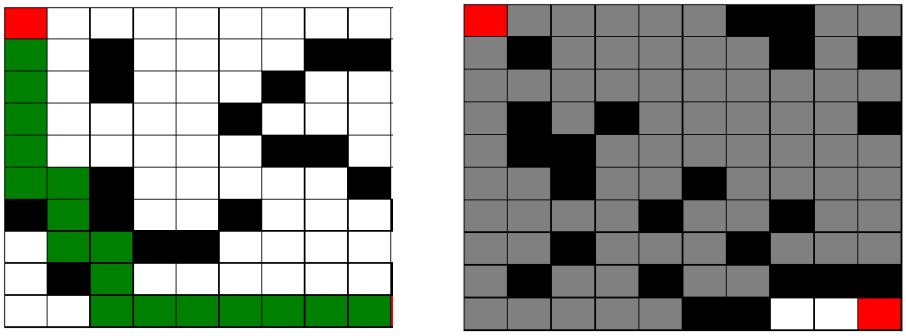
The A^* algorithm with Manhattan heuristic is used to pick the best dimension size. The best dimension size is defined as a size that requires some work to solve, but small enough that it can run each algorithm multiple times for a range of possible p values. Note that the A^* algorithm with Manhattan heuristic is used as it finds the best optimal path and it is relatively fast. The maze is created with three probabilities of 0.1, 0.2 and 0.3. The time taken (in seconds)

to solve the maze against the maze dimension is compared for the three probabilities.

By looking at the plots it is seen that till the maze reaches a dimension of 75 the time taken is reasonable. As the probability increases and the maze dimension increases, the time taken to solve the maze grows from a maze dimension of 75. Hence, this seems to be the optimal maze size to solve our mazes.

0.2.2 Solvable mazes

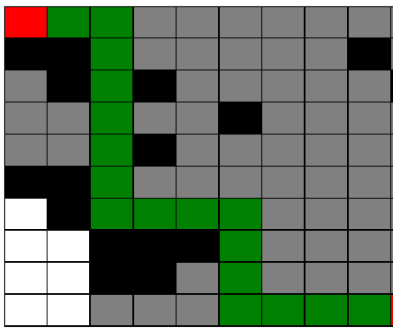
For p $\approx$ 0.2, generate a solvable map, and show the paths returned for each algorithm. Do the results make sense?



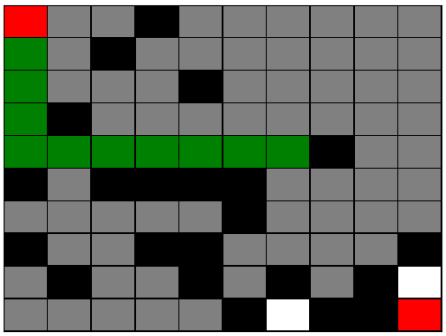
(a) Path found

(b) No path found

Figure 4: Path taken by DFS Algorithm

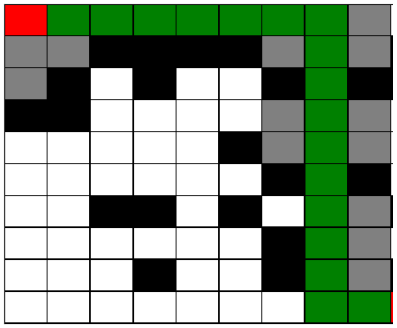


(a) Path found

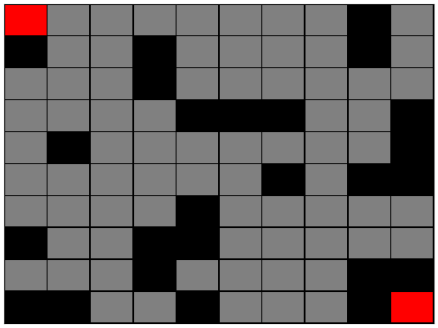


(b) No path found

Figure 5: Path taken by BFS Algorithm

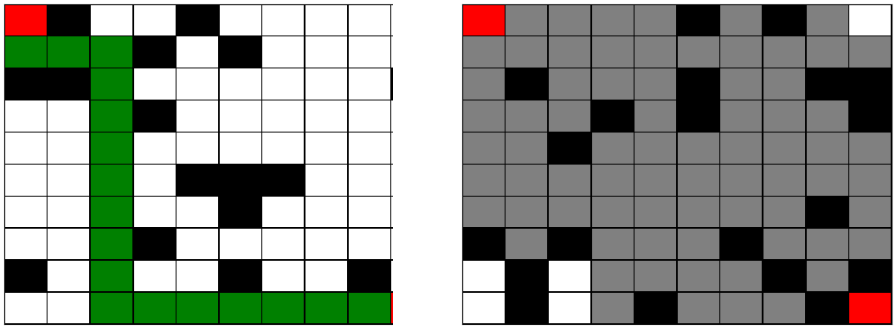


(a) Path found



(b) No path found

Figure 6: Path taken by A^* Algorithm with Euclidean distance as a heuristic



(a) Path found (b) No path found

Figure 7: Path taken by A^* Algorithm with Manhattan distance as a heuristic

Yes, the results seen for each algorithm is exactly how they should behave. Figures 5-8 show the behavior of the corresponding algorithm for two cases - when a path is found and when there is no path. Note that the final path taken by each algorithm is the optimal path it could take, given a path was present.

0.2.3 Does maze-solvability depend on p?

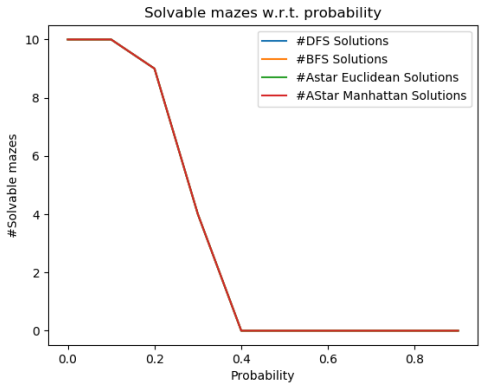


Figure 8: Shortest Path Length vs. Probability

By taking the optimal maze dimension size of 75, we see that the maze-solvability depends on the probability p. Figure 9 captures this dependency. The plot compares the number of mazes that are solvable for each algorithm as the probability increases. Note that the maximum number of solvable mazes is set as 100.

As expected, all the algorithms have the same number of solvable mazes. There is no best algorithm that can be used here for this problem. But, in order to find if a maze is solvable or not faster, we can deploy the A^* algorithm because it tries to find the optimal path. In order to find the threshold p_0, where for $p < p_0$, most mazes are solvable, but $p > p_0$, most mazes are not solvable, we look at the plot made. It is seen that for $p = 0.2$ most of the mazes are solvable. Therefore, our threshold p_0 is equal to 0.2.

0.2.4 Shortest paths

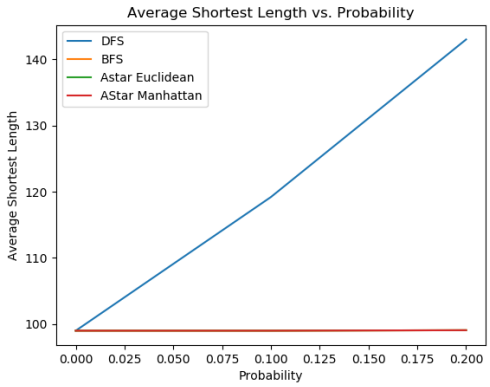
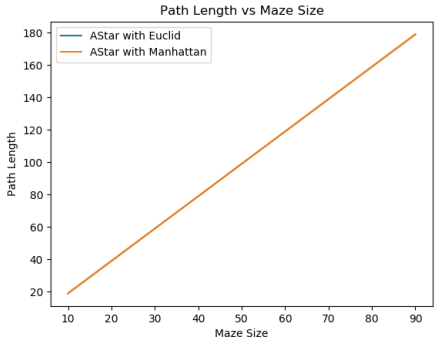


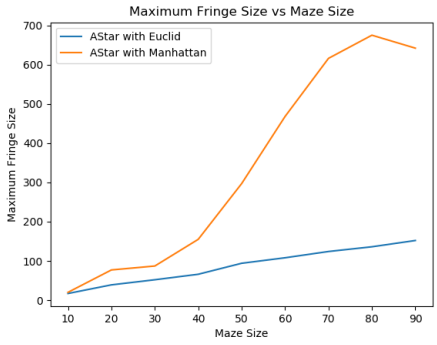
Figure 9: Shortest Path Length vs. Probability

In Figure 10, we compare the average shortest path length between probabilities 0 and $p_0 = 0.2$ for all the algorithms. Except DFS algorithm the other three algorithms have the same shortest path length as these algorithms always look for the optimal path between the start and goal. Hence, comparatively, we can choose to use A^* with a Manhattan heuristic.

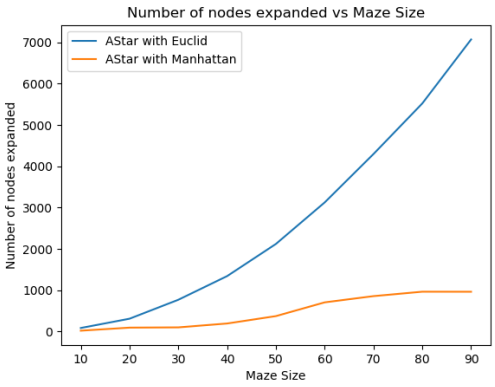
0.2.5 Heuristic comparison for A^*



(a) Path Length vs. Maze Dimension



(b) Maximum Fringe Size vs. Maze Dimension



(c) Number of nodes expanded vs. Maze Dimension

Three metrics are used to compare the performance of the heuristic used along with A^* algorithm. The three metrics are - Path length, Maximum fringe size and Number of nodes expanded.

As expected, the path length for both the heuristics are the same as they always find the optimal path from start to goal. Hence, based on path length no comparison can be made. While looking at the metric, maximum fringe size, it is seen that Manhattan heuristic has a higher maximum fringe size when compared to the Euclidean heuristic. On the other hand, when we look at the number of nodes expanded the Manhattan heuristic is better than the Euclidean heuristic. Hence, if we want to reach the goal node faster the Manhattan heuristic is the better option.

0.2.6 Comparison of BFS and DFS

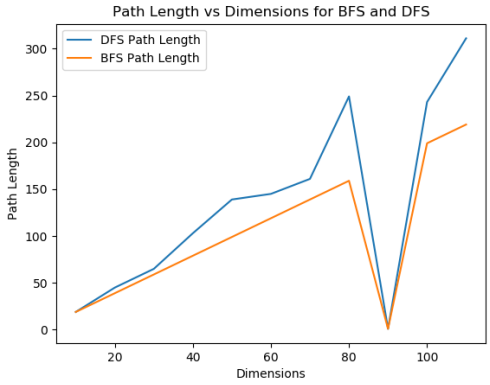


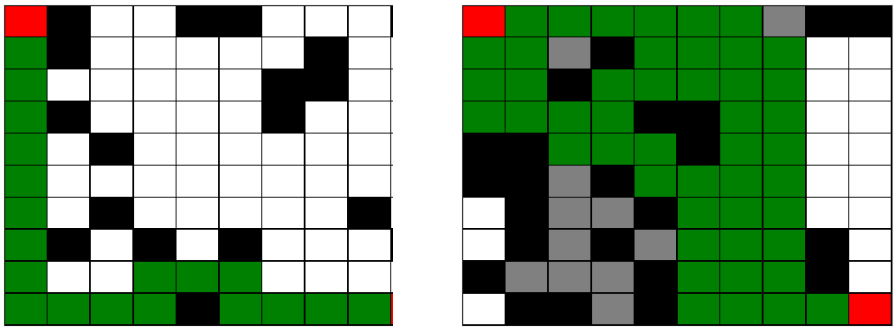
Figure 11: Path Length vs. Maze Dimension for BFS vs. DFS

As seen in Figure 13, the DFS path length is always higher than BFS path length as the maze dimension increases. This occurs as the BFS algorithm always looks for the optimal path from start to end. Hence, when we compare the two algorithms in term of optimal shortest path taken, BFS will always be better. Note that when the maze dimension was 90 there is a drop as there was no path found. Also, when we compare the other two metrics - Maximum fringe size and number of nodes expanded, BFS does better in terms of the former metric and DFS does better in terms of the latter metric. These observations are seen earlier.

0.2.7 Do these algorithms behave as they should?

Yes. The DFS and BFS algorithms should behave the way we see them. The reasoning is based on the metrics seen earlier and how the algorithm is designed.

0.2.8 Can DFS be improved?



(a) With priority of right & down neighbors (b) With priority of left & up neighbors

Figure 12: Path traversed by DFS algorithm

For DFS, the algorithm can be improved by prioritizing the right directions while choosing the order of loading the neighboring rooms into the fringe. Note that as the start is present on the left top corner and the goal is present on the bottom right corner, neighbors - right and down is worth looking into first rather than the neighbors - left and up.

By looking at the path traversed by the DFS algorithm it is clear that when we prioritize right and down it reaches the goal much faster as compared to prioritizing left and up. Hence, DFS can be improved by prioritizing the right neighbors to look at when we know where the goal node is present.

0.3 Generating Hard Mazes

We now go into the details about the hard-maze generating algorithms and how we found some great hard-mazes for our algorithms.

1. The Local Search Algorithm -
    - We picked hill climbing as our local search algorithm to generate harder mazes. Hill climbing is a local search algorithm which looks at its immediate neighbors and searches in the direction of what ever we are trying to maximize.
    - So, in the case of generating harder mazes by using hill climbing, a maze can be made harder by adding a wall in one of the cells. This new maze generated will be a neighbor to the original parent maze. If the path on this maze is larger than the parent maze, then our

algorithm will choose the new maze and advance one step towards the hardest solvable maze. This way all the potential neighbors of a maze are traversed and become potential candidates for hard mazes.

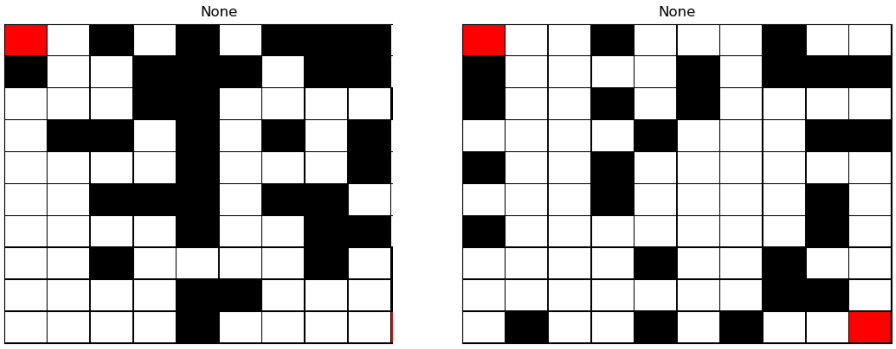
- In this way, we iterate through all n^2, neighbors and try to advance towards the hardest maze. If a child maze is found having a larger metric greater than the original maze, we fix this child maze and search for a harder maze on all its neighbors.
- However, if we are unable to find a hard maze in the neighbors, then we might have reached a local maximum. Post which, we randomly generate another maze and repeat the whole process.
- At the end of our iterations, we would be left with the hardest maze for that particular metric.

2. Termination Conditions -

   - Based on the maze dimension and metric chosen, iterations it took to find the hard maze was different. We mostly tried generating mazes for maze dimension of 10.
   - A termination condition on number of iterations of random start can be applied to know if there existed any harder maze. Number of iterations required to generate harder maze was directly proportional to n. And improvement in metric initially in finding harder maze was often, but we moved towards finding harder maze, the metric increase by only unit or two only after larger iterations. We used a maximum iteration condition to terminate the hill climbing. However, designing a termination condition based on n and iteration count from last improved score could help in reducing the time complexity.
   - Hill climbing was in most of the cases was able to find, the hard maze starting from easier maze. However, if the maze generated is the hardest maze possible is difficult to conclude. Another short coming, we could see was the termination condition for stopping the search, since we could not really understand the global metric maximum.

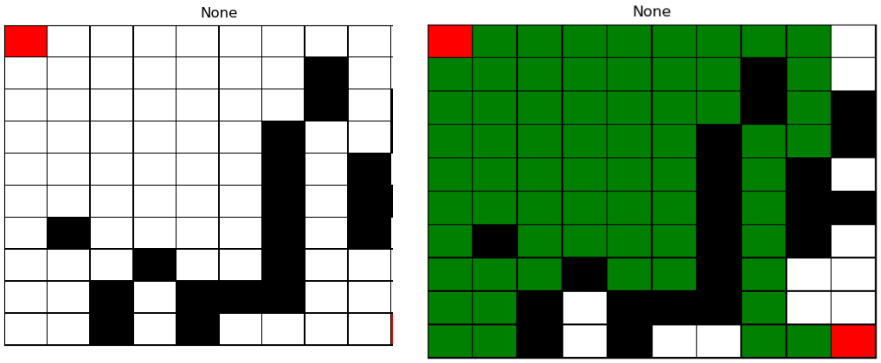
3. Examples of Hard-Mazes Generated -

   - **DFS with Maximal Shortest Path**



(a) Hard maze 1 (b) Hard maze 2

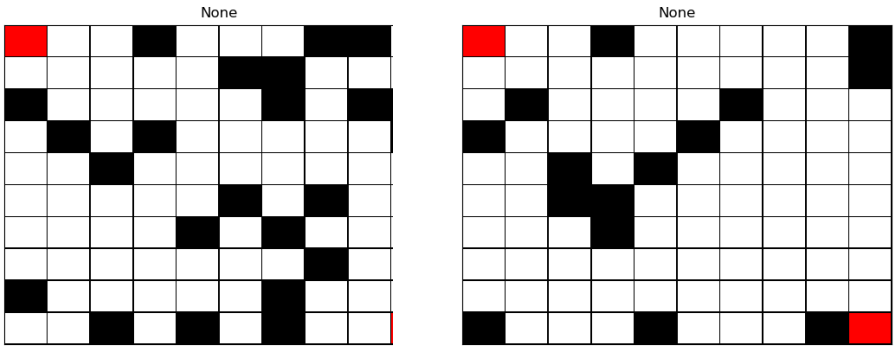
Figure 13: Hard Mazes found after 10 and 60 Iterations



(a) Hardest maze (b) The path through the hardest maze

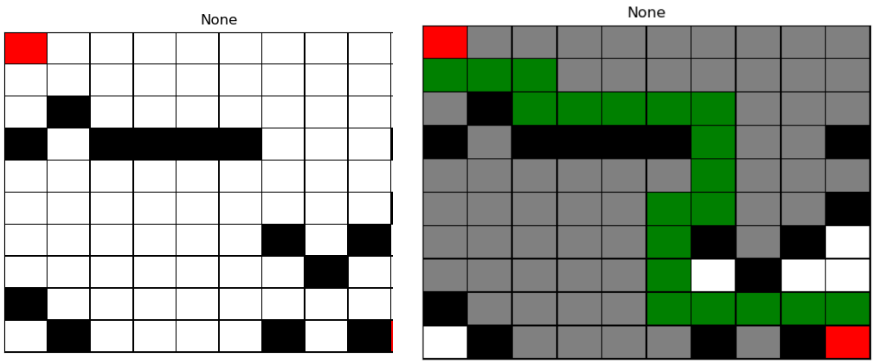
Figure 14: Hardest Maze For DFS based on Maximal Shortest Path

- **DFS with Maximal Fringe Length**



(a) Hard maze 1 (b) Hard maze 2

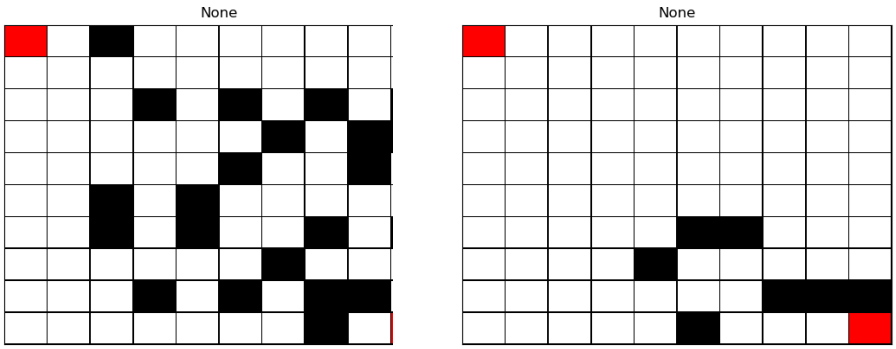
Figure 15: Hard Mazes found after 14 and 25 Iterations



(a) Hardest maze (b) The path through the hardest maze

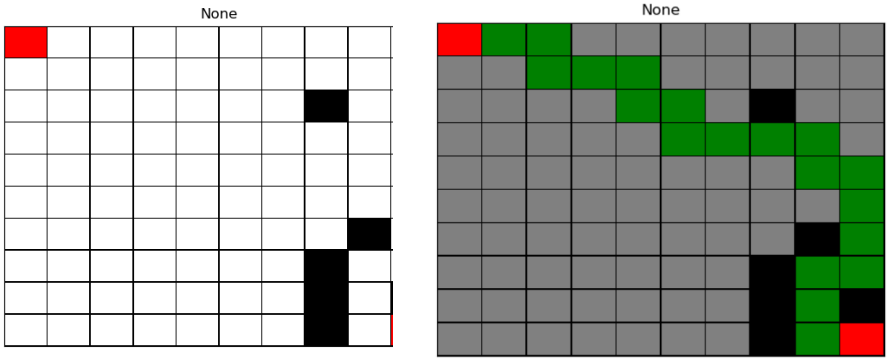
Figure 16: Hardest Maze for DFS based on Maximal Fringe Length

- $A^* - Manhattan$ **with Maximal Nodes Expanded**



(a) Hard maze 1 (b) Hard maze 2

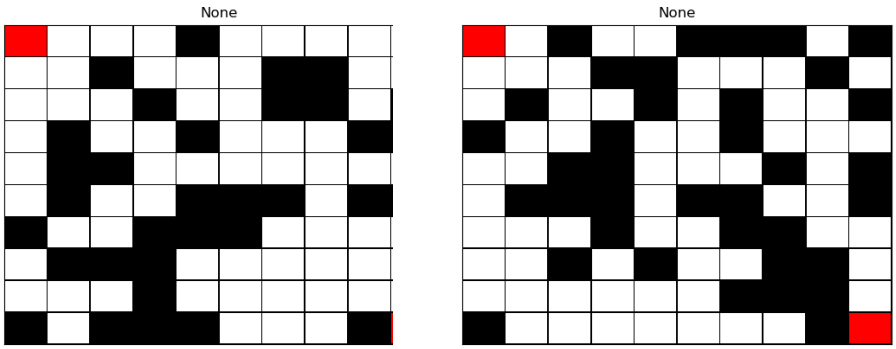
Figure 17: Hard Mazes found after 14 and 184 Iterations



(a) Hardest maze (b) The path through the hardest maze

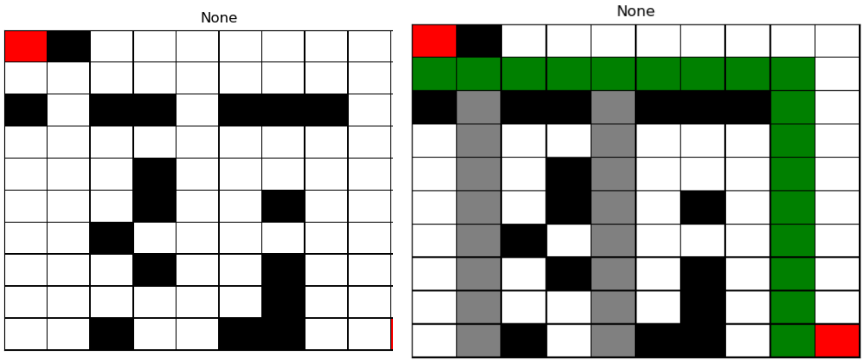
Figure 18: Hardest Maze for $A^* - Manhattan$ based on Maximal Nodes Expanded

- $A^* - Manhattan$ **with Maximal Fringe Length**



(a) Hard maze 1 (b) Hard maze 2

Figure 19: Hard Mazes found after 1 and 8 Iterations



(a) Hardest maze (b) The path through the hardest maze

Figure 20: Hardest Maze for $A^* - Manhattan$ based on Maximal Fringe Length

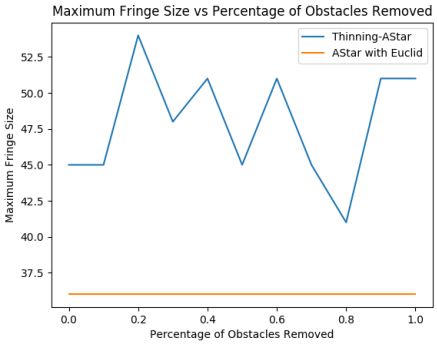
4. As the iteration increases ,we move towards the global maximum of Maximal shortest path, we notice that, algorithms would almost pass through majority of the nodes in the case of harder mazes, the walls are generated in right places and which makes the algorithms to take longer paths/ explore nodes covering most nodes to reach the goal. This is what happens in other harder mazes generated for different metric/algorithms, and they work exactly like expected.

0.4 Thinning A^*

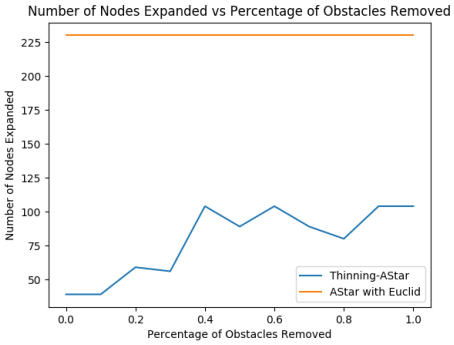
We now compare the Thinning-A-Star algorithm with the A-Star algorithm for the 2 heuristics - Euclid and Manhattan distance. We use the same heuristic in thinning A-Star which we used for the normal A-Star.

1. Thinning A-Star-Euclid vs A-Star-Euclid:

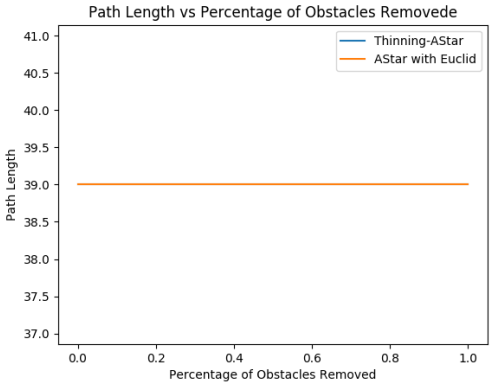
   The path length stays the same for both algorithms which is obvious since both the versions of A-Star algorithms will find the optimal path in the end. The major difference will be in the fringe size and the number of nodes expanded.



(a) Maximum fringe size



(b) Number of nodes expanded



(c) Path length

Figure 21: Comparison of Thinning A-Star and A-Star with Euclid Heuristic

(a) Maximum Fringe Size - From the above graph for the fringe size, we observe that the maximum fringe size is very large for Thinning A-Star than the normal A-Star. We also take into account the fringe

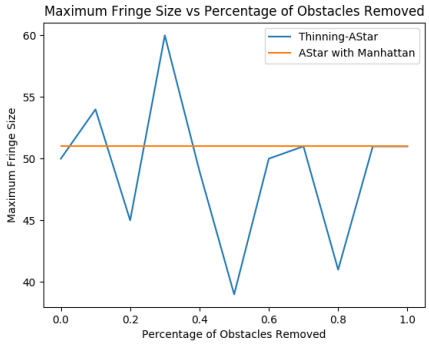
we generate when we run an A-Star for each potential child during the Thinning A-Star. So based on that, the fringe size becomes very large and the memory used during Thinning A-Star-Euclid is more than A-Star-Euclid.

(b) Number of Nodes Expanded - Here the Thinning A-Star algorithm performs a lot better than the normal A-Star.

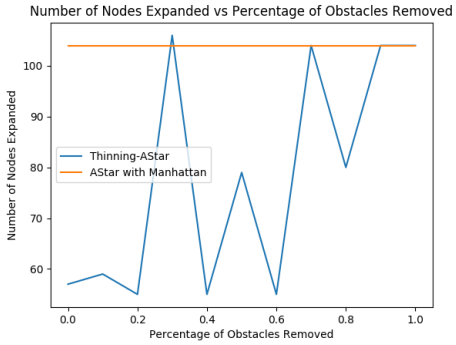
- The number of nodes expanded during the search is very low for Thinning A-Star. However, we only take into account the final fringe which we use for storing the final children and not the temporary fringe during the temporary A-Star on each child/neighbors.
- The reason for this is that Thinning A-Star itself chooses heuristic values by running an A-Star-Euclid on each potential neighbour. So the heuristic value is pretty close to the actual path length from that child to the destination. Hence, it has to expand as few nodes as possible to arrive at the final path length.
- Finally, the Thinning A-Star also expands on a lot of nodes when calculating the heuristics for each neighbour, so in practice it may not be a good choice to use it since the final path is the same.

2. Thinning A-Star-Manhattan vs A-Star-Manhattan:

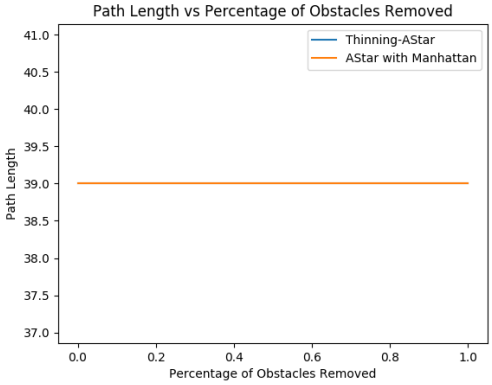
   Again, the path length for the both the algorithms stays the same for the Manhattan heuristic.



(a) Maximum fringe size



(b) Number of nodes expanded



(c) Path length

Figure 22: Comparison of Thinning A-Star and A-Star with Manhattan Heuristic

(a) Maximum Fringe Size - From the above graph for the fringe size, we observe that the maximum fringe size is again large for Thinning A-Star than the normal A-Star. In general, the maximum fringe size stays below that of A-Star-Manhattan.

(b) Number of Nodes Expanded - Here the Thinning A-Star algorithm performs a lot better than the normal A-Star. The number of nodes expanded is less than A-Star-Manhattan. However, Thinning-A-Star-Manhattan will again take more time and computation than normal A-Star because it runs an algorithm for each neighbor to calculate its heuristic.

0.5 How to solve a maze on fire?

The questions asks us to create a structure and solution of a maze which is in essence "dynamic". Till now the mazes we have been running our algorithms on were static, which meant that the algorithm had the map for the maze, it spent some time computing the best possible path to take and then took that path to reach to the goal, if possible. In this question, we have to come up with a solution that will incorporate the dynamic nature of the maze (fire spreading to the tiles at every turn) and try to find the solution before fire blocks the possible solutions. With the advent of fire, there are some situations which should be kept in mind while making an algorithm. Now, any cell can have three states: 'open', 'blocked', or 'on fire'.

Before starting with the algorithm, we made changes to our visualization code by incorporating the probability factor of fire spreading. By applying these rules, we were able to visually see how the fire is spreading through the maze. This helped us see that unlike our pointer (which can only move once per step), fire can spread to at most 2 tiles with probability between 0.5 to 1. Also, this makes the task even more difficult because the fire is spreading more quickly.

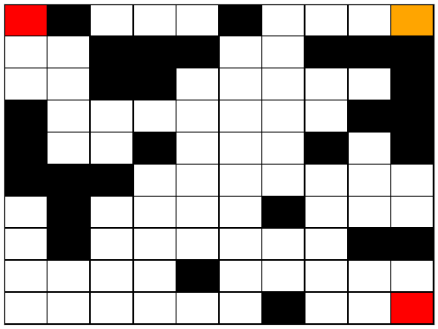


Figure 23: Initialising the Fire Maze

With the codes already deployed, we started thinking along the lines of making a custom heuristic to solve this problem. Since A-Star already uses distance (euclidean or manhattan) as a heuristic for finding paths, we took the same idea and modified our heuristic to include the distance (only euclidean considered) from all the blocks on fire. Calculating distance from all fire blocks does not solve the problem, so we took the fire block which has minimum distance from our current block. We removed the distance from source from our final heuristic because in this scenario of fire, with fire increasing rapidly, it does not make sense to go back a lot of steps - if you go back 2 steps, the fire has spread to at least 2 tiles. Now, we have two distances with us, distance from closest fire,

and distance to destination. We need to come up with a function that includes both to be represented as the final heuristic. The way we finalized the heuristic is as follows:

- Parameters:
  - Distance from closest fire: Maximize this distance (to feel safer)
  - Distance to destination: Minimize this distance (to feel closer to solution)
  - Alpha - Value given to change the importance of fire distance
- Cost function:
  - Minimize (-Alpha*Distance from fire + Distance to destination)
  - We are able to minimize this function since we are adding the negative of distance from fire to destination.
  - Example: Suppose we have two scenario - dist.fire1 = 700, dist.dest1 = 200 and dist.fire2 = 600, dist.dest2 = 300. Our heuristic will choose the first scenario because it offers a safer yet closer path

Now that we have our heuristic, we actually implemented a DFS + Heuristic approach, in which the next child to be explored will be the child having the best heuristic possible at that step. The priority queue prioritizes children with heuristic values. This allows us to expand the block which presents a safer and closer distance to the solution. In some sense this is a greedy algorithm as we only look at neighbors of our current cell and choose the best neighbor greedily.

After running through various iterations, we realized a shortcoming of our algorithm. Since, it chooses its next child greedily (greedy algorithm), there are times when it gets stuck due to a closed block.

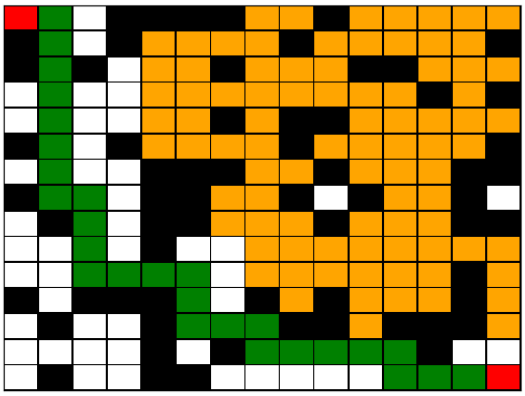


Figure 24: Successful Path found by the Fire Algorithm