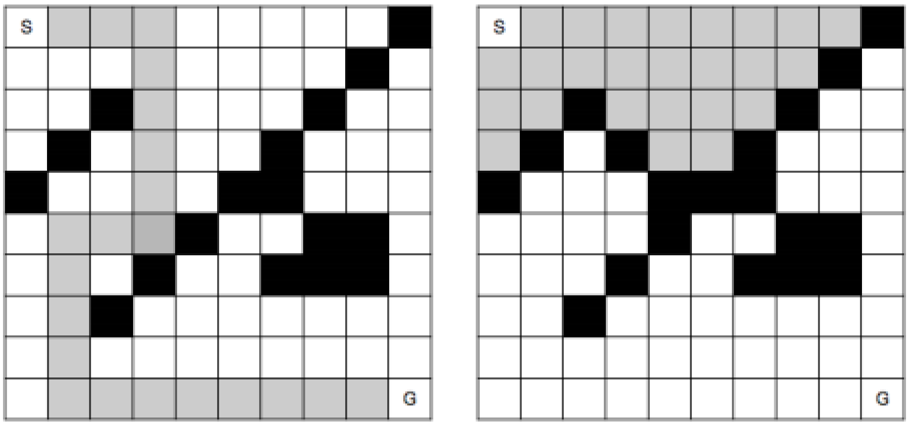
Maze Runner Analysis

CS 440



Muhammad Arif, Cesar Herrera, Jae Weon Kim

Spring 2020

List of Tables

List of Figures

Introduction

This project is intended as an exploration of various search algorithms, both in the traditional application of path planning, and more abstractly in the construction and design of complex objects.

Questions

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

2. For $p = 0.2$ generate a solvable map, and show the paths returned for each algorithm. *Do the results make sense? ASCII printouts are fine, gut good visualizations are a bonus*

3. Given **dim** how does maze-solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple mazes and checking them for solvability. What is the best algorithm to use here? Plot *density vs solvability,* and try to identify as accurately as you can the threshold p_0 where for $p < p_0$, most mazes are solvable , but $p > p_0$, most mazes are not solvable

4. 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?

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

6. Do these algorithms behave as they should?

7. For DFS, can you improve the performance of the algorithm by choosing what order to load the neighboring rooms into the fringe? What neighbors are 'worth' looking at before others? Be thorough and justify yourself.

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

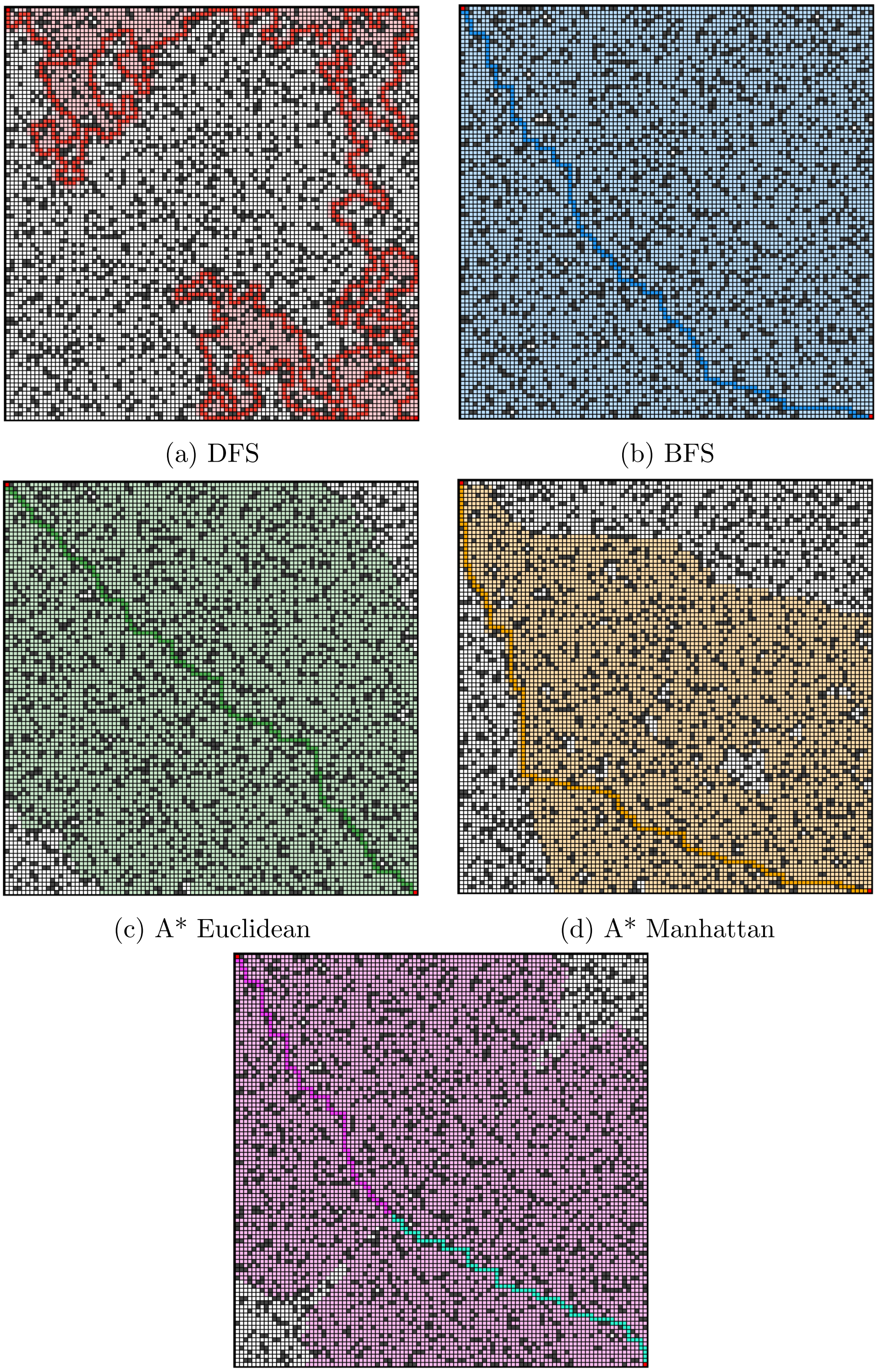
Analysis and Comparison

1) After testing through various map sizes we ultimately decided on a dimension of ***100***. To come to this conclusion we ran our algorithms 1000 iterations on increasing dimension sizes. Through these tests we noticed that solvability remained around the same with increasing map sizes, but the time to solve the mazes increased as map size increased. The results are displayed on Table 1 below.

Dimension	**Time:(seconds)**	**Solvable:(out of 1000)**	**Solvable:(percentage)**
10	1.417	832/1000	83.2%
25	6.091	841/1000	84.1%
50	21.914	840/1000	84.0%
100	76.892	861/1000	86.1%
200	313.148	846/1000	84.6%

Table 1: **Increasing Dimension Size Impact**
p *value used:* **0.2**

2) We generated a **100 x 100** maze with ***p=0.2*** and ran our different algorithms on it. See Figure:1 Paths below



(a) DFS

(b) BFS

(c) A* Euclidean

(d) A* Manhattan

(e) BI-BFS

Figure 1: Paths

3) For any dimension size maze solvability remained relatively the same for a specific $\boldsymbol{p}$ value. Thus, to test out the impact of different $\boldsymbol{p}$ values we iterated ***1000*** times for $\boldsymbol{p}$ values in range of ***0*** to ***1***. Our results are graphed below. See Figure: 2 below. Out of the five algorithms we used DFS as it was the fastest performing algorithm . From the data we can conclude that the threshold $\boldsymbol{p_0}$ is around ***0.3***, because when $\boldsymbol{p}$ <***0.3*** most mazes are solvable and when $\boldsymbol{p}$ >***0.3*** most if not all are unsolvable.

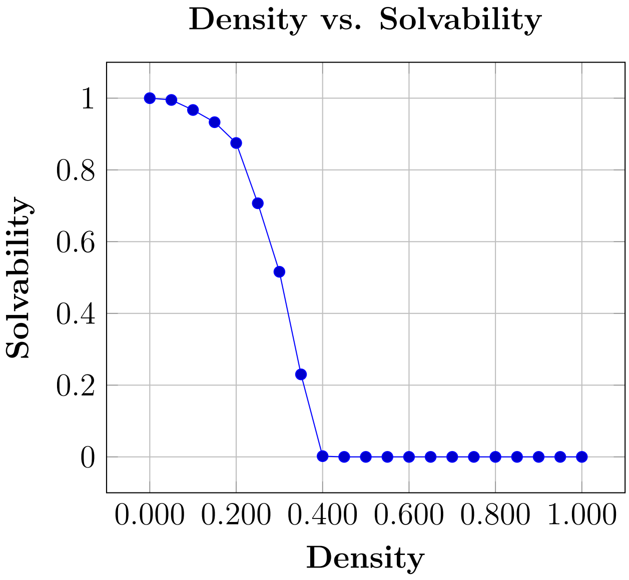


Figure 2: Density vs. Solvability

4) Intuitively, to estimate the average or expected length of the shortest path from start to goal we went for BFS as the algorithm is intended to find the shortest path to the goal node. Thus, our results are shown in Figure: 3 below. ***(Iterated 1000 times.)***

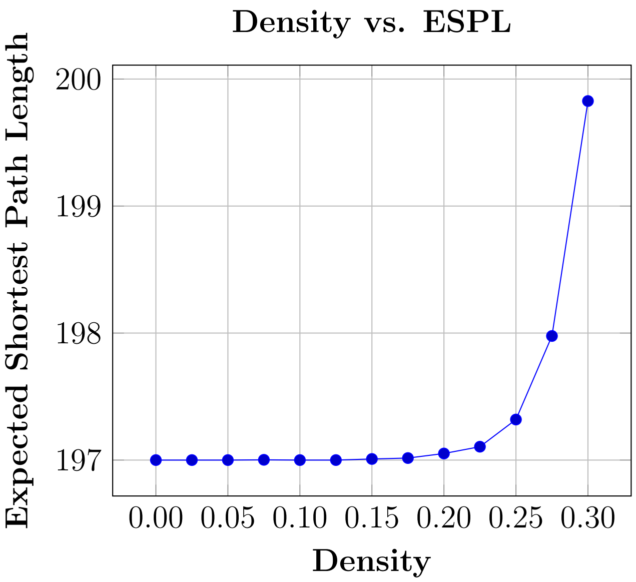


Figure 3: Density vs. Expected Shortest Path Length: BFS

5) Depending on what you define as ***"better"***, one heuristic can be called better than the other. One way to define ***"better"*** can be that we want the least number of cells visited in order to find the path from the starting cell to goal cell. Thus, this is a comparison on which one is ***"faster"***, both computationally and time wise because it theoretically visits less cells if the algorithm is implemented correctly. in Figure: 4 below.

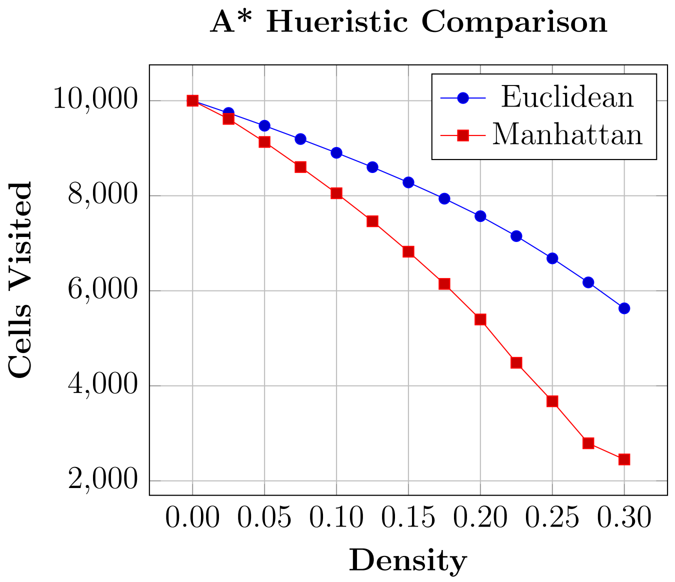


Figure 4: A* Hueristic Comparison: Euclidean vs. Manhattan

6) In analyzing our results, the algorithms implemented do behave as they should. ***See Figure:1 Paths above for visualization.***

7)For ***DFS***, you can drastically improve the difference of the algorithm by choosing what order to load the neighboring rooms into the fringe. For example, our first implemenatation of ***DFS*** our algorithm prioritized going up and to the right, however in the second improved DFS, prioritization of the down and right directions actually resulted in shorter path lengths and less number of visited cells. Thus, in order to quantify this we graphed the average path length for a given density for *1000* iterations. See Figure 5 below.

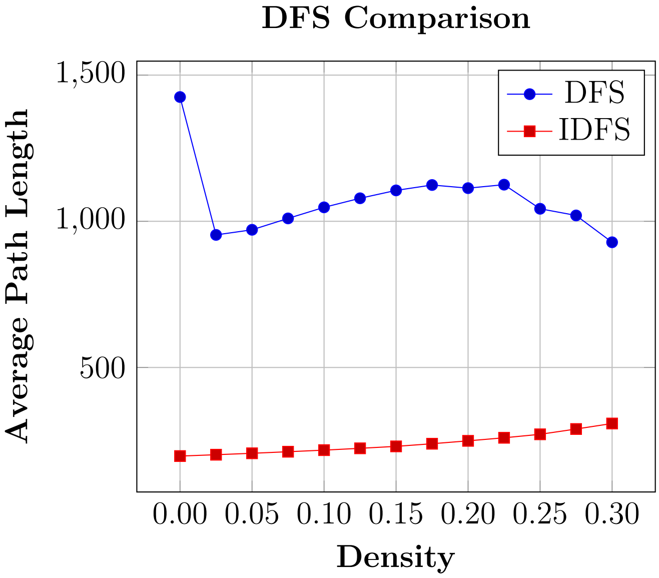
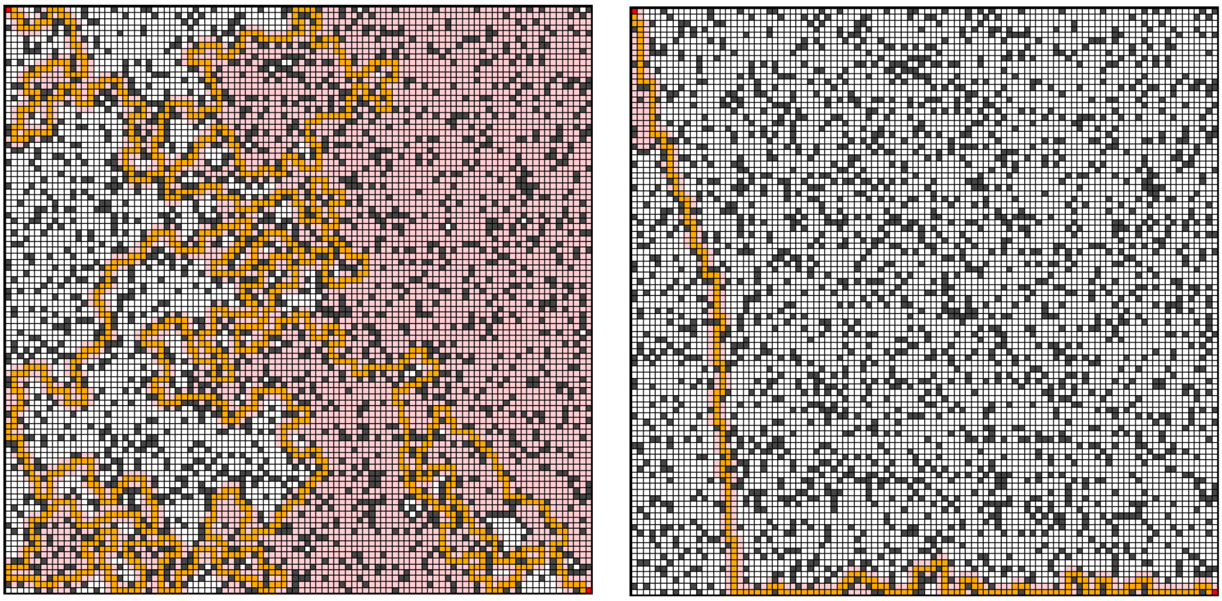


Figure 5: DFS vs IDFS(Improved DFS)

Furthermore, to visualize this difference you can take a look at the following images comparing***DFS*** vs the***improved DFS***.

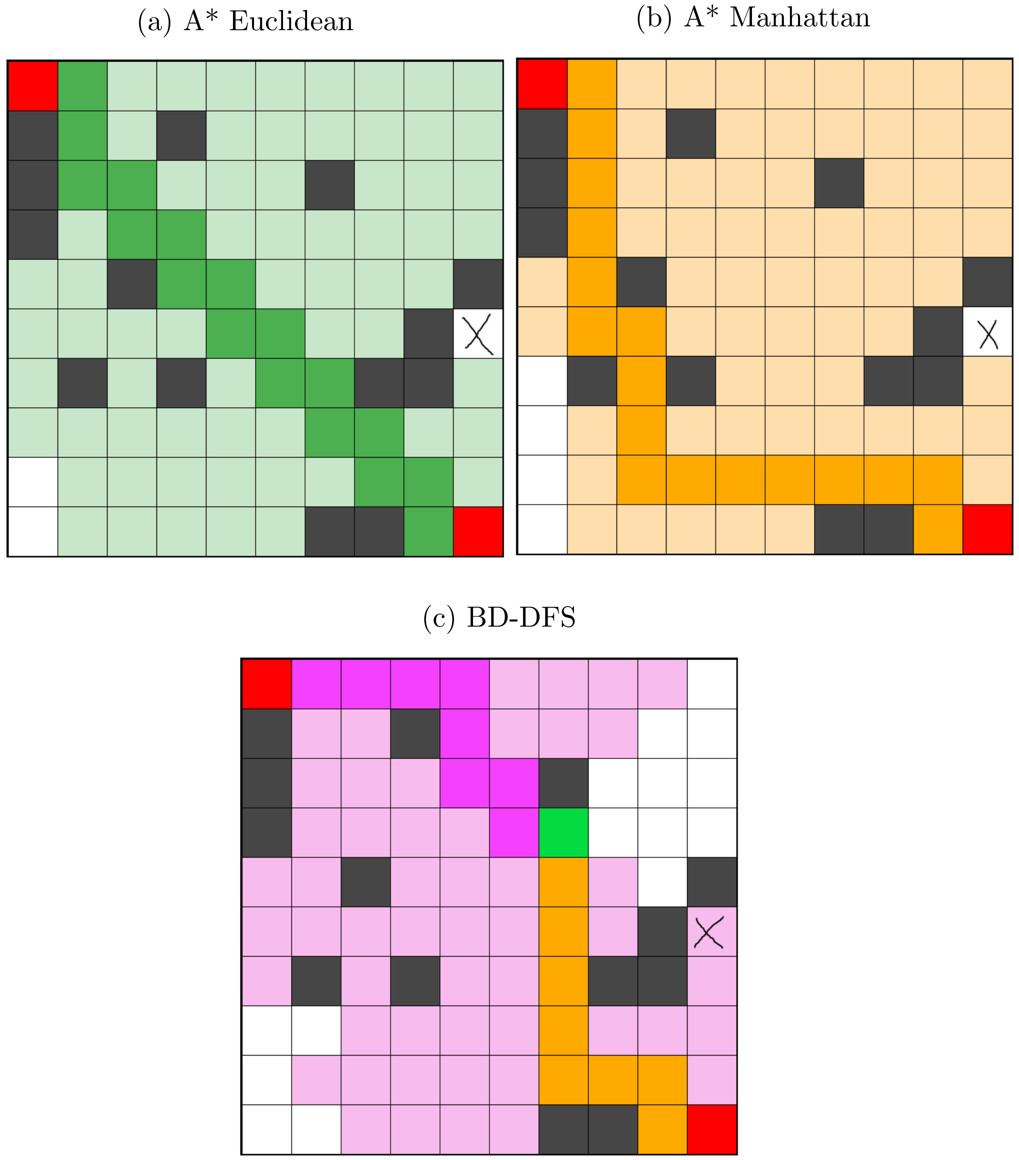


(a) DFS

(b) DFS

Figure 6: DFS vs IDFS
Dimension:*100 x 100* ***p***:*0.20*

8) On the same map there are times that Bidirectional Breath First Search (***BD-BFS***) expands that **_A*_** doesn't. To make the visualization easier, instead of using a dimension of 100 by 100 like in our previous questions for this we can examine a 10 by 10 maze. See Figure: 7 below for the paths given by the three algorithms.



(a) A* Euclidean

(b) A* Manhattan

(c) BD-DFS

Figure 7: A* vs BD-DFS
Dimension:*10 x 10* ***p***:*0.20*

As you can see we have marked an x on the cell in which both the ***euclidean*** and ***manhattan A**** did not visit, while ***BD-BFS*** did. The reason for this is because of the nature of breath-first search. Since, breath-first search works on the premise of discovering cells layer by layer, it "expands" outwards, giving it more of a chance to explore cells that ***A**** doesn't. Given that it also starts from both the start and goal cells, it will only come to an end once it finds the intersecting cells. On the other hand, the way that the ***A**** algorithm uses it's heuristics in combination with the priority queue by adding the distance to the current node and the estimated distance to the destination from said current node, it gives higher preference to the directions in which the estimated distance left is less. Hence, that cell explored by ***BD-BFS*** and not the ***A**** algorithm was due to the estimated cost of exploring it being higher than that of the ones that were explored.

Generating Hard Mazes

The local search algorithm that we decided to implement was hill climbing, as it is designed to find the optimal cost albeit a greedy approach. In order for our search algorithms to use the created mazes, we are representing the mazes as a 10 x 10 grid, with the probability $\boldsymbol{p}$ = 0.2. The design choices that we had to make in order to apply this search algorithm to our problem was finding a way to compute the cost of what a 'hard maze' was. In order to proceed from there we also had to find a way to store the previous maze generated and then compare with the new one to see if was 'harder'.

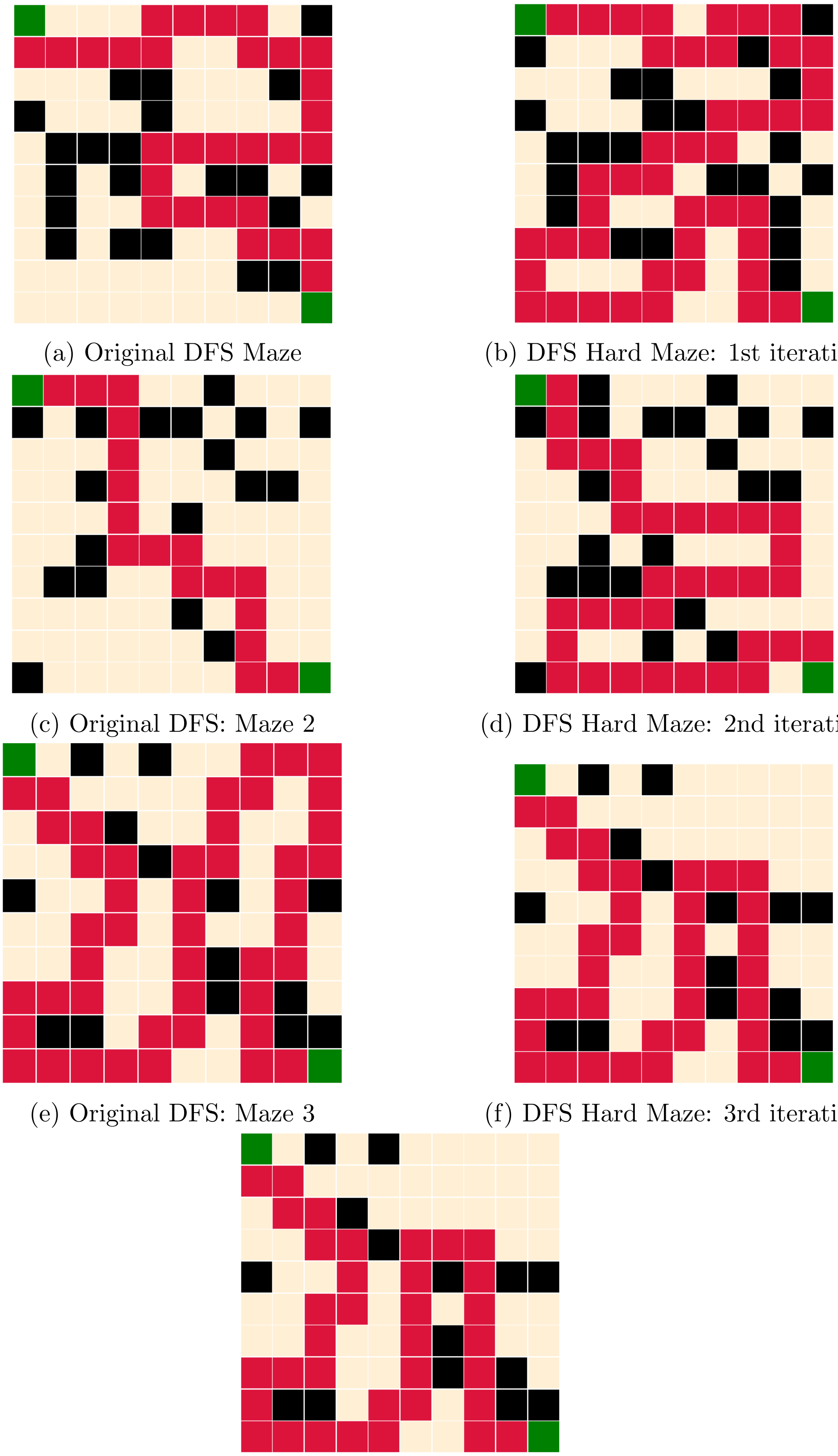
Unlike the problem of solving a maze, for which the 'goal' is well-defined, it is difficult to know if you have constructed the 'hardest' maze. The termination conditions that we decided to apply here to generate hard if not the hardest maze included the following:

1. Generate the original maze
2. Pass a constant parameter to our hill climbing algorithm for the max number of hill climbing searches to be done which eventually terminates the loop. (*for our tests below we set it to a maximum of 100*)
3. Pass another parameter to our hill climbing algorithm for the number of iterations for maze generation to happen.

Some of the shortcomings/advantages that we anticipate from our approach are that since hill climbing is a greedy approach, it's advantage is that it will find a local optimum for the cost metric. However, it's advantage is also it's shortcoming as we anticipate that the local optimum might not be the actual hardest maze that you can generate with the given maze.

Below are our attempts to find the hardest mazes for the following paired metrics

- DFS with Maximal Fringe Size (***See Figures: 8,9 for DFS*** & **10,11 for IDFS** *for mazes generated and plot of cost*)
- A*- Manhattan with Maximal Nodes Expanded (***See Figures: 12*** & **13** *for mazes generated and plot of cost*)



(a) Original DFS Maze

(b) DFS Hard Maze: 1st iteration

(c) Original DFS: Maze 2

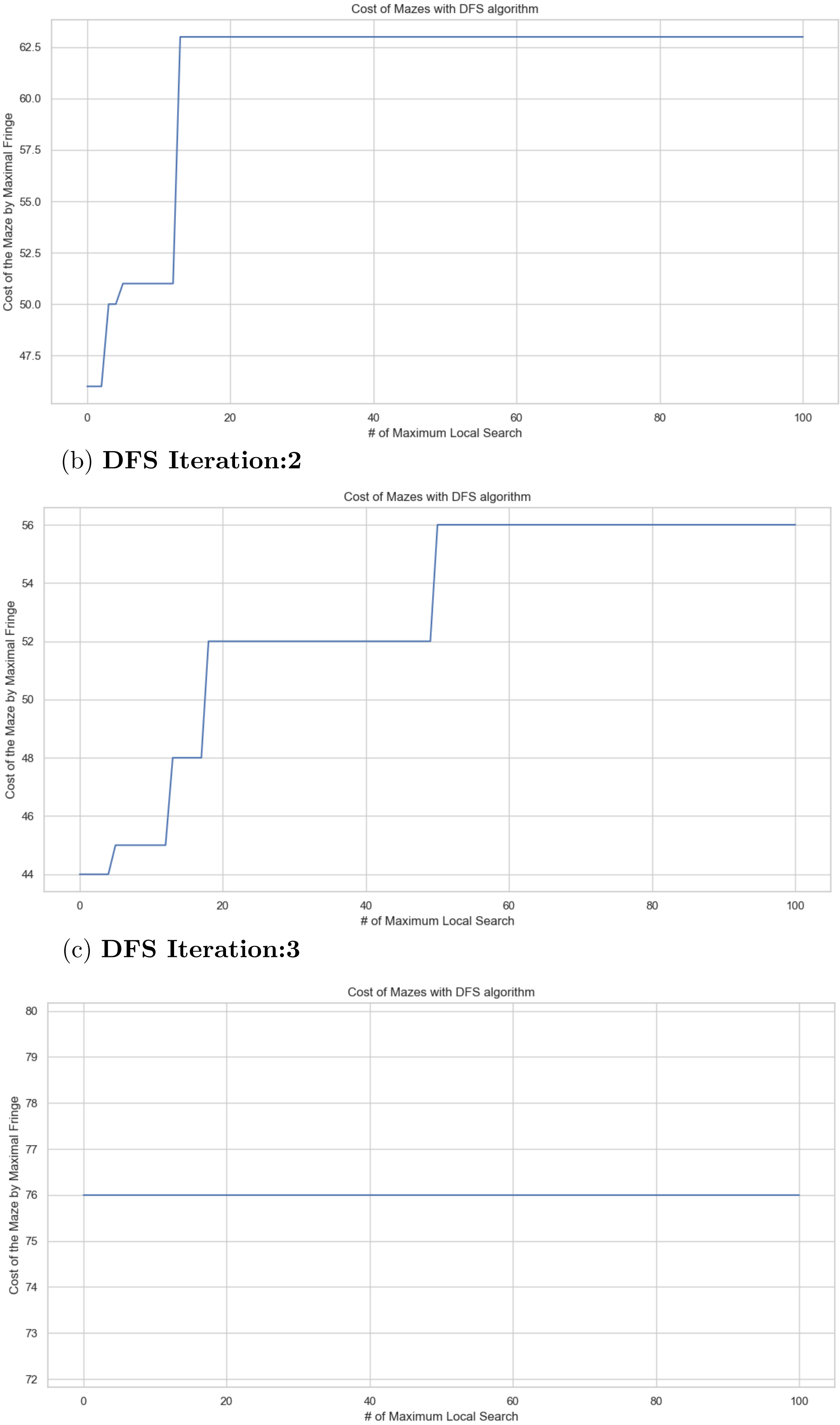
(d) DFS Hard Maze: 2nd iteration

(e) Original DFS: Maze 3

(f) DFS Hard Maze: 3rd iteration

(g) DFS Hard Maze: Final Hardest Maze

Figure 8: Hard Maze Generation: DFS

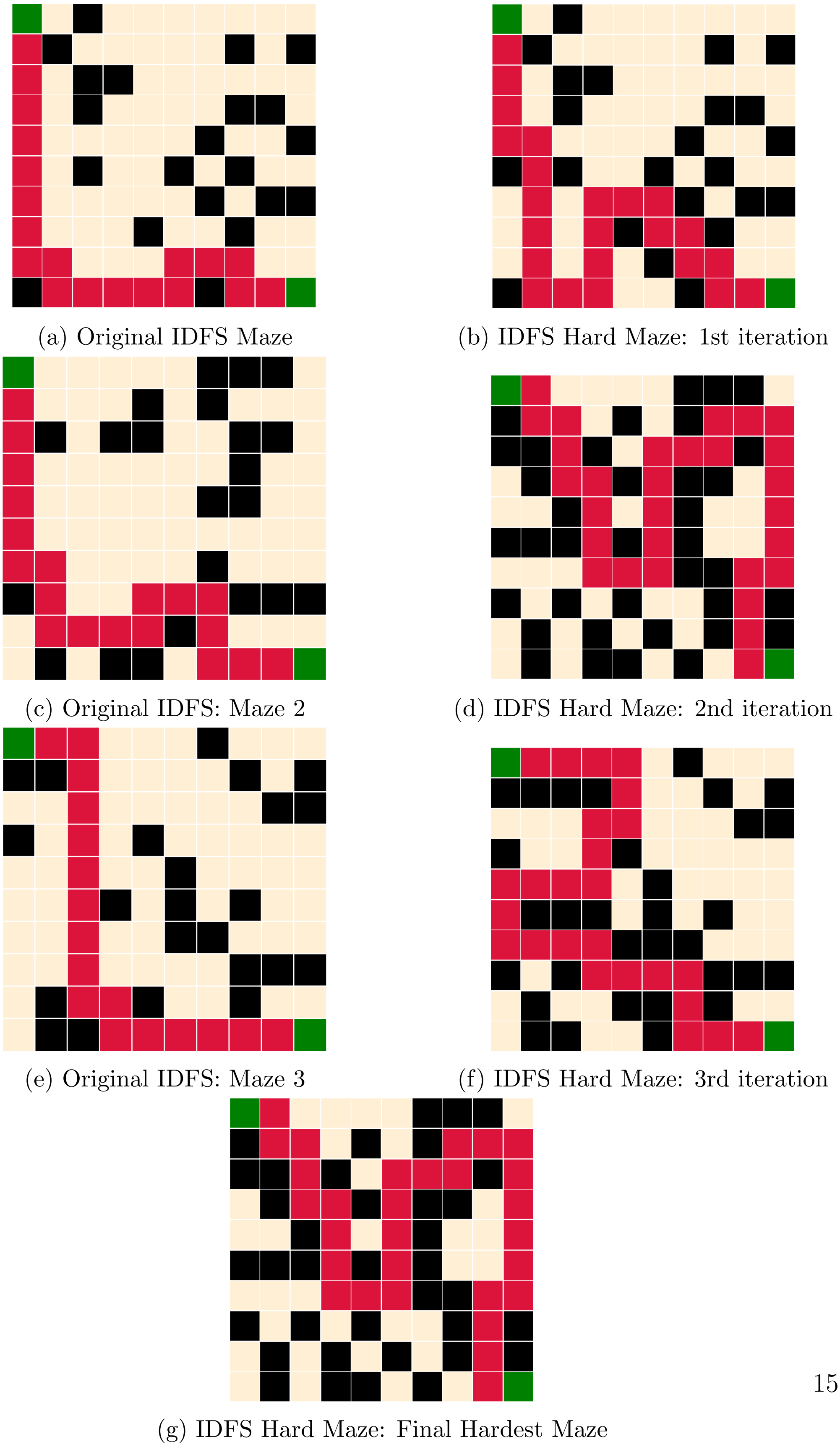


(a) **DFS Iteration:1**

(b) **DFS Iteration:2**

(c) **DFS Iteration:3**

Figure 9: **Data**: DFS



(a) Original IDFS Maze

(b) IDFS Hard Maze: 1st iteration

(c) Original IDFS: Maze 2

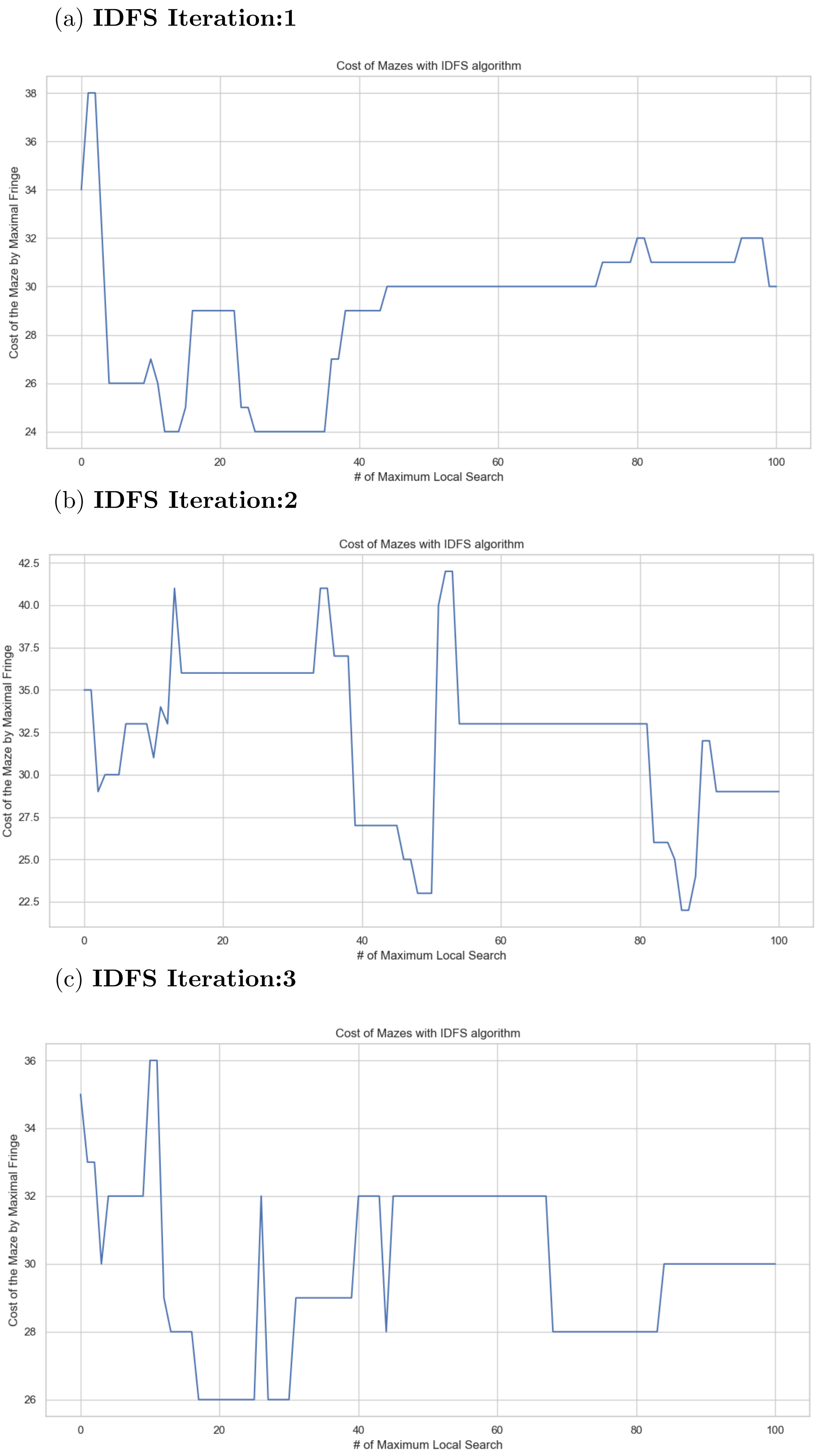
(d) IDFS Hard Maze: 2nd iteration

(e) Original IDFS: Maze 3

(f) IDFS Hard Maze: 3rd iteration

(g) IDFS Hard Maze: Final Hardest Maze

Figure 10: Hard Maze Generation: IDFS

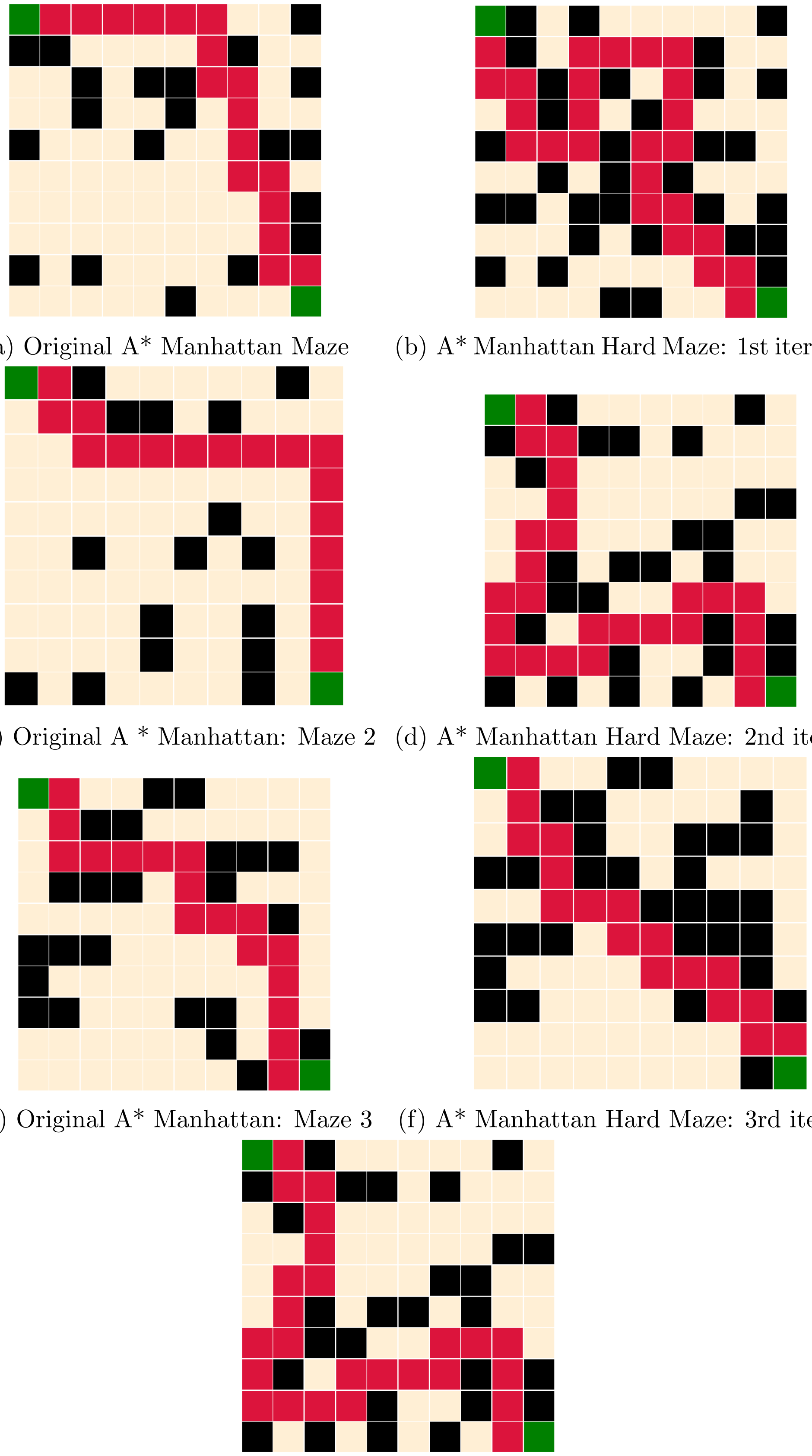


(a) **IDFS Iteration:1**

(b) **IDFS Iteration:2**

(c) **IDFS Iteration:3**

Figure 11: **Data**: IDFS



(a) Original A* Manhattan Maze

(b) A* Manhattan Hard Maze: 1st iteration

(c) Original A * Manhattan: Maze 2

(d) A* Manhattan Hard Maze: 2nd iteration

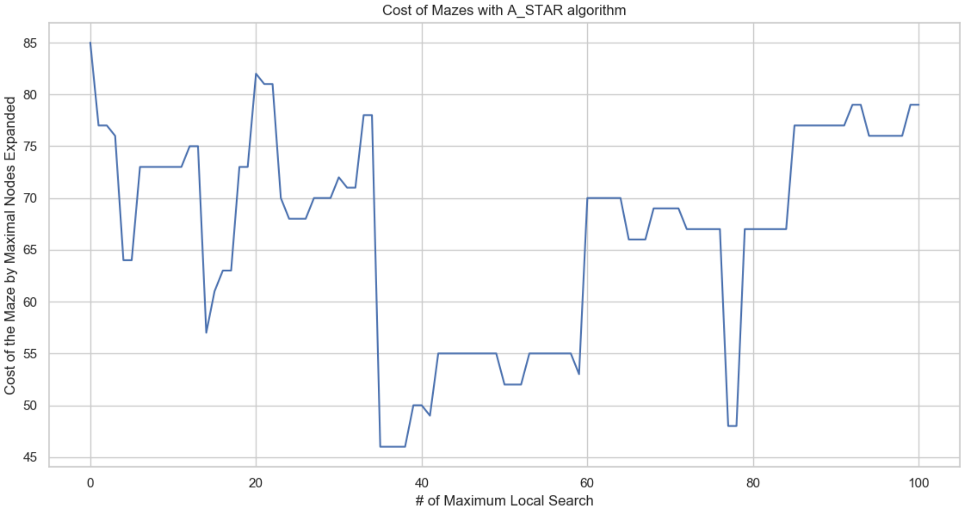
(e) Original A* Manhattan: Maze 3

(f) A* Manhattan Hard Maze: 3rd iteration

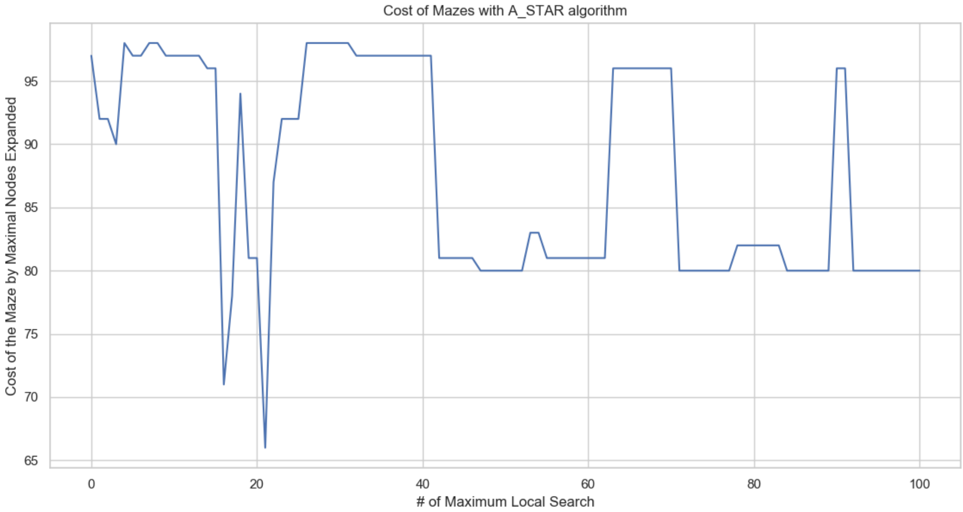
(g) A* Manhattan Hard Maze: Final Hardest Maze

Figure 12: Hard Maze Generation: A* Manhattan

(a) **A* Manhattan Iteration:1**



(b) **A* Manhattan Iteration:2**



(c) **A* Manhattan Iteration:3**

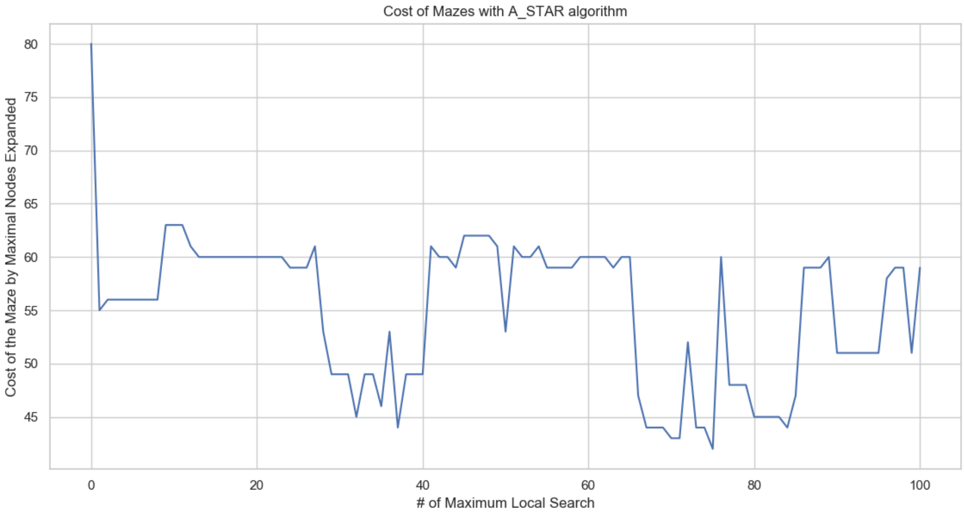


Figure 13: **Data**: A* Manhattan

Overall, our results do agree with our intuition as the cost for each iteration sometimes was not the optimum.

What If The Maze Were On Fire?

Approaches Taken:
The fire is spreading based on a probability that is decided priorly, and the result will be computationally extensive to predict the rate of the fire that is being spread beforehand. However, the result will be various based on the different approaches to the optimal distance that is computed by the subtraction between max distance and min distance. Hence, among the three different strategies, the best approach for our search algorithm would be a greedy approach to predict the neighboring cells which are priorly computed to the current maze running point in order to avoid intersecting with the potential fire spaces as the runner extends its move.

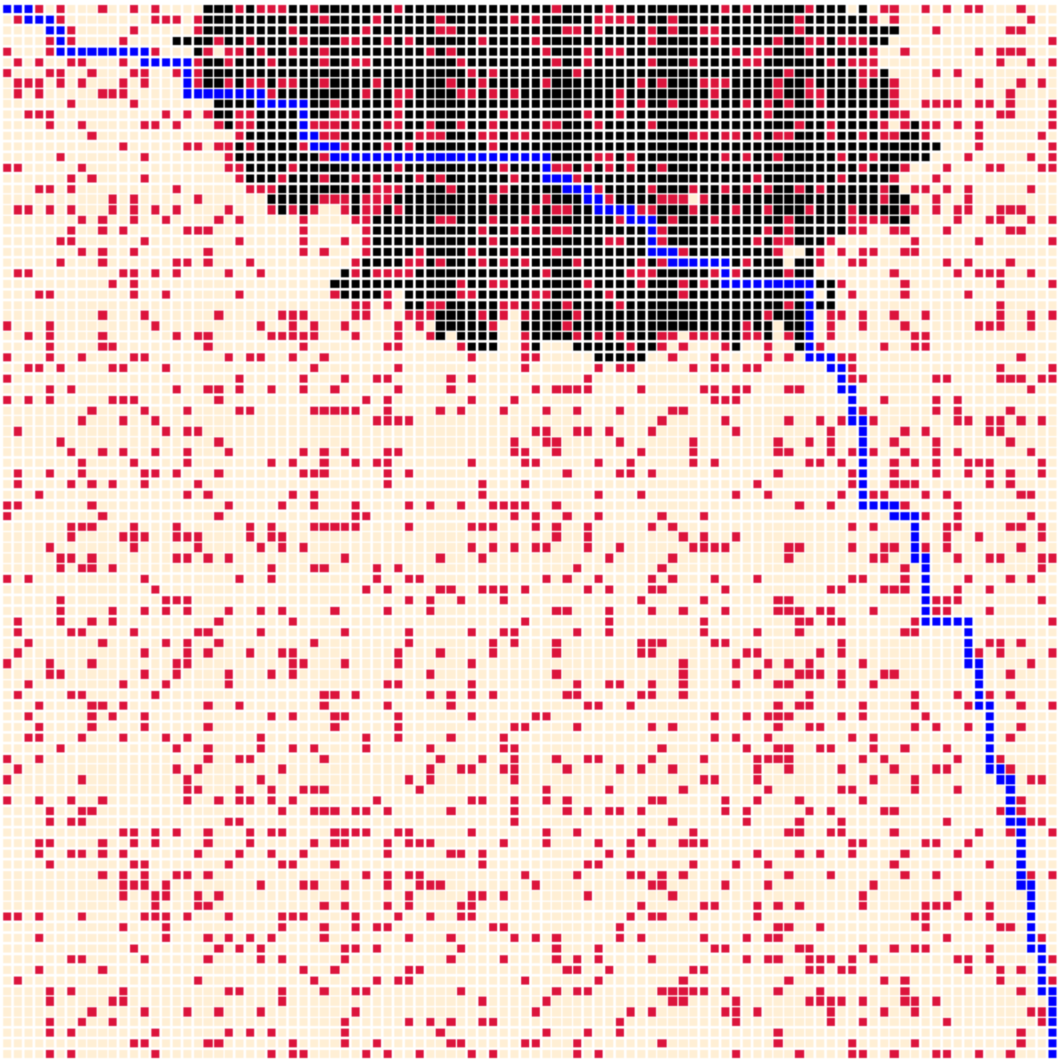


Figure 14: ***Maze Runner on Fire Strategy One:***
Shortest path from left to lower right.
Fire Starts on cell: *(54, 5)*
Time taken for strategy: *10.10969066619873047 seconds*

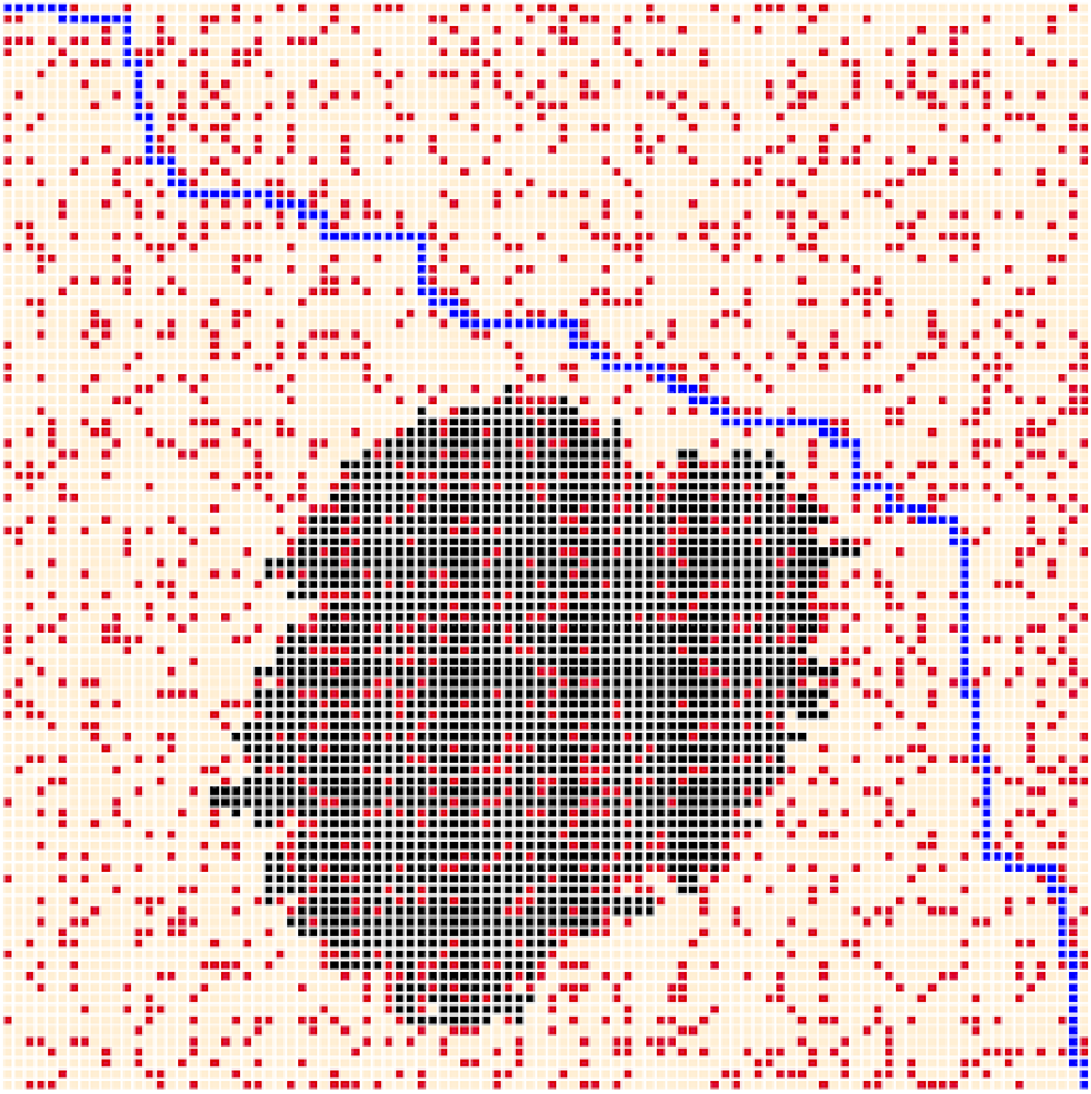


Figure 15: ***Maze Runner on Fire Strategy Two:*** *Recomputing every distance by subtracting minimum distance from maximum distance.*
Fire Starts on cell: *(62, 43)*
Time taken for strategy: *0.13666892051696777 seconds*

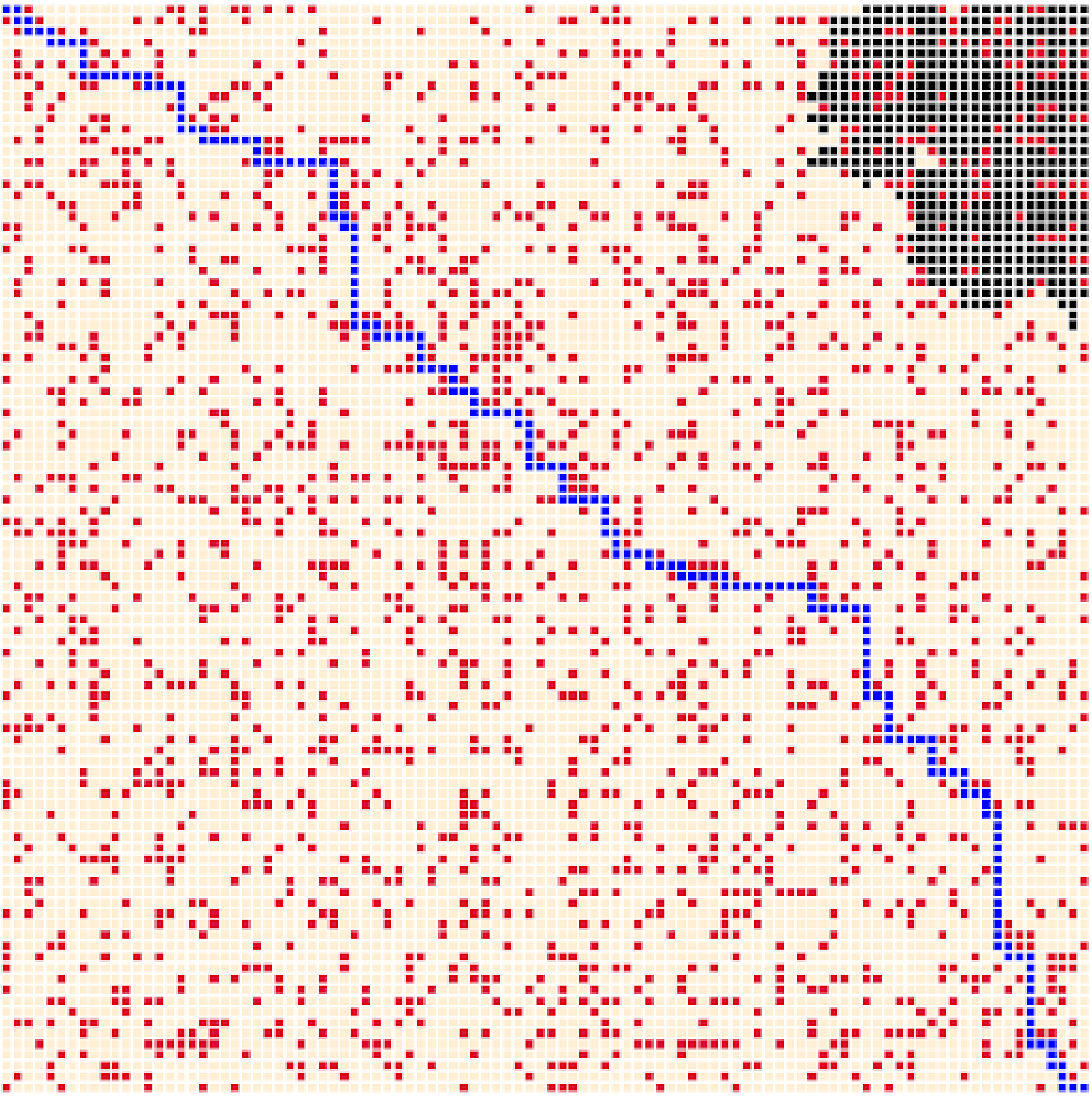


Figure 16: ***Maze Runner on Fire Strategy Three:***
Applying maximum distance from new fire cell.
Fire Starts on cell: *(97, 34)*
Time taken for strategy: *0.2124311923980713 seconds*

Approach to Strategy Three: In general, to get the optimal distance between the runner and fire is subtracting the manhattan distance from goal by distance from the nearest fire cell. On top of that, strategy three should avoid the possible moves that the maze runner approaches to the possible fire cells no matter where the fire starts at. Hence, if we continuously stay

away from the fire cells and implement the approach for maximizing the minimum distance from the new fire cells, we can apply this algorithm by iterating loops for computing the maximum min distance from the fire cell to the maze runner coordinate.
The figure for strategy one (Figure: 14) was almost impossible to get the plot due to the high repetition for the maze runner collapse with the fire that was being spread.

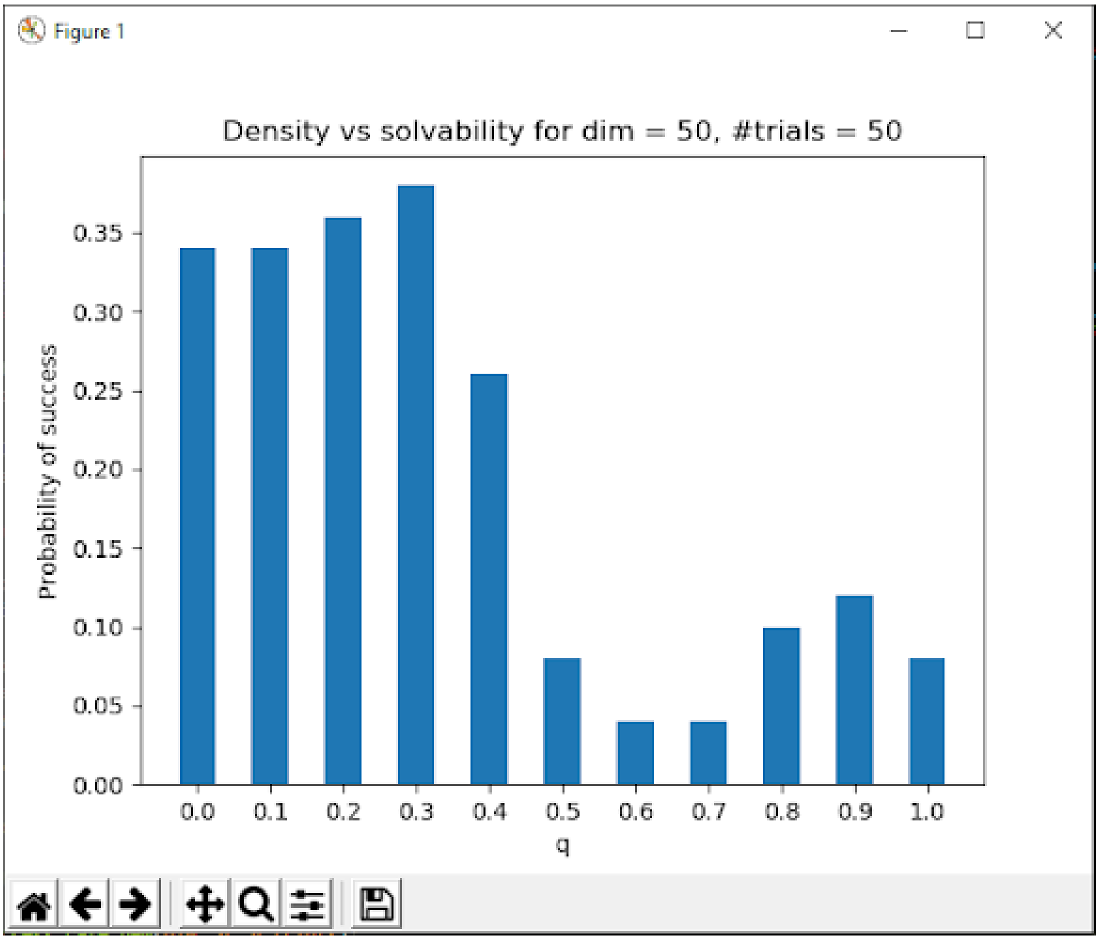


Figure 17: ***Retrieval of Data by implementing Maze on Fire Strategy Two***

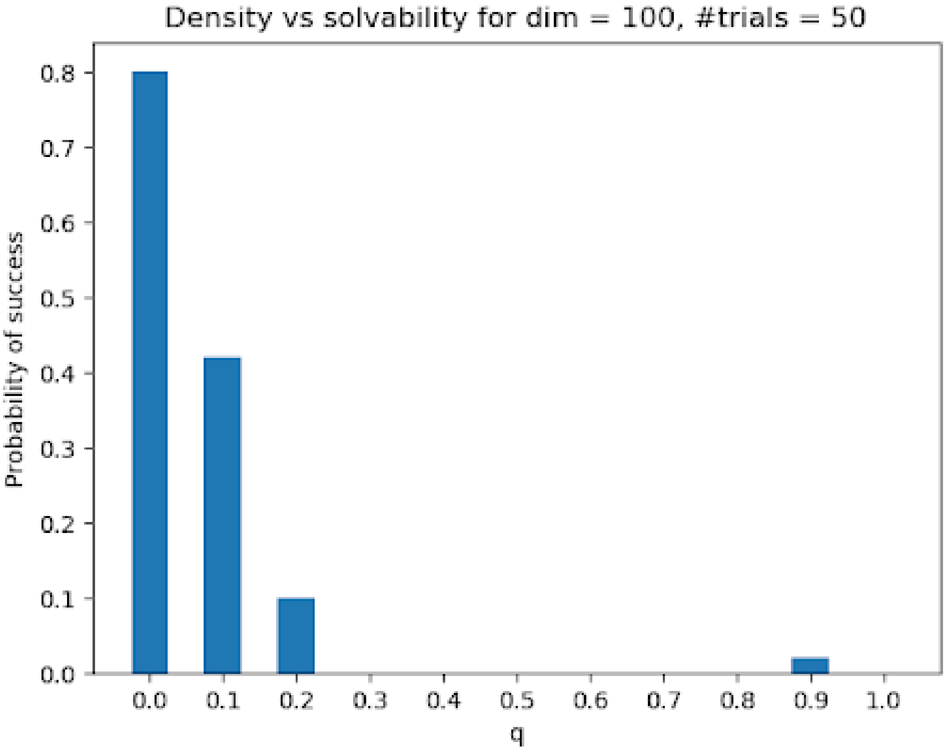


Figure 18: ***Retrieval of Data by implementing Maze on Fire Strategy Three***

According to the two plots above (*Figure: 17 and Figure: 18*), the result was quite interesting due to the range of the probability of success and the range of flammability, q. In Figure: 17, the distribution for the probability of success was separated into two groups, one with less than 0.5 for q-value and another one for more or equal to 0.5 for q-value. This represents that the success rate for any possible q-value can be found more than once due to the approach with the fire runner with maximum distance from all cells on fire.

On the other hand, in Figure: 18 indicates the probability of success is exponentially decreasing while the q-value is increasing. But, the success rate for both 0.0 and 0.1 is much higher than the plot in In Figure: 18.

According to the two plots above, the result was quite interesting due to the range of the probability of success and the range of flammability, q. In Figure: 17, the distribution for the probability of success was separated into two groups, one with less than 0.5 for q-value and another one for more or equal to 0.5 for q-value. This represents that the success rate for any possible q-value can be found more than once due to the approach with the fire runner with maximum distance from all cells on fire.

On the other hand, in Figure: 18 indicates the probability of success is exponentially decreasing while the q-value is increasing. But, the success rate for both 0.0 and 0.1 is much higher than the plot in 18.

Old Harder

We used an evolutionary algorithm for this problem. This class of algorithms can select for any variables that are of interest for us. We want to find the longest shortest path. This would be most computationally intensive for both metrics of maze difficulty. We measure the shortest distance using A* Manhattan search, because that gives the shortest path on a maze. We cannot move diagonally, so a Manhattan search is necessary.

A Make a generic maze: We calculate the shortest solution for the maze

B Inherit: The maze moves five steps, and saves the nodes it visits.

C Mutation: We shuffle the maze randomly. We find the shortest solution. If the new solution is greater than the old one, we save the new solution.

D We continue C and D until we reach the goal state.

Make generic maze: We calculate the shortest solution for the maze. Inherit: The maze moves five steps, and saves the nodes it visits. Mutation: We shuffle the maze randomly. We find the shortest solution. If the new new solution is greater than the old one, we save the new solution. We continue C and D until we reach the goal state.

This process would be most difficult for A*-manhattan metric, because the maze would produce the shortest longest path. In the case of a DFS, this process would also be very challenging. The shortest-longest path would force DFS, on average, to have the highest fringe size.

We decided this process would terminate based on iterations. We applied the process 300 times. We are assuming that this number of iterations will get us in range of the global optimum. We do of course not know if this is the case. We do not know how far we are from the global optimum. We could have improved our algorithm with more iterations. Moreover, we could more aggressively introduce mutations. This would allow us to explore the space of possible solutions more; however it would also decrease the speed at which we arrived at any optimum. We could also improve our algorithm if we had a better idea of what the global optimum is.

Our results do agree with our intuition. We find mazes with longer shortest solutions after applying our algorithm.

In the table 2 below we included our results from some trials.

Number of Iterations	**Shortest-Longest Path**
0	1.417
100	6.091
200	21.914
300	76.892

Table 2: **Hard Maze Results**

Contributions

Jae: All code for part one, three and four, and report for part four
Cesar: Analysis for part 2, and report for part 3
Muhammad: All of old part three