Zekun Zhang, Ziqi Wang, Richa Rai

Artificial Intelligence
September 26, 2019

Assignment 1

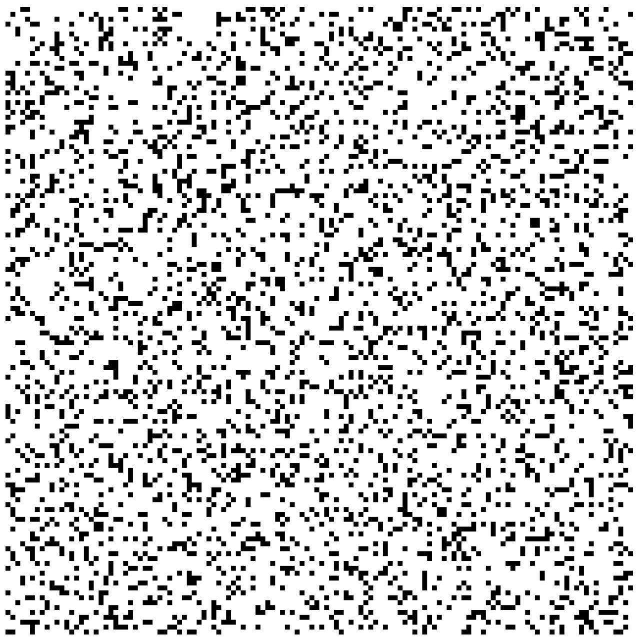


Figure 1: A sample maze with dim = 128 and p = 0.2.

Part 1

Given an input dimension and an occupancy probability p, a maze is generated by inserting values drawn from a uniformly distributed random variable X $U0,1$, at each cell. A value of 1 means the cell is occupied and thus cannot be traversed and a value of 0 means the cell is free space and traversable. An example is presented in figure 1. Visualization is made by expanding each cell to a 16x16 pixel tile, assign black color to occupied cells and white color to empty cells. Cells that are parts of solution attempts are displayed in gray.

Part 2

Bullet a

I chose dim = 128. It provides a reasonable total execution time and a reasonably large map. The complexity of DFS and BFS algorithms is in the order of $O(|V| + |E|)$. In our case if we consider our maze as a graph, it is a 4-way connected case, where $|V|$ $|E|$ and thus the running time of the search is $O(2n) = O(n)$. This is confirmed by the code where for n = 64 time is about 0.16 seconds and for n = 128 about 0.38 seconds. The bottleneck in the code is the initialization of values and the accessing of data structures in Python. For this reason even though the actual running time is small it takes about 1.4 seconds for the program to halt.

Algorithms	Success Ratio	p_0
BFS	$\sim$ 0.65	0.3
DFS	$\sim$0.66	0.26
A* Man	$\sim$0.66	0.26
A* Euc	$\sim$0.66	0.26
BiBFS	$\sim$0.66	0.37

Table 1: Density p_0 for acceptable solvability, per algorithm.

Bullet b

The results for a solvable map with p = 0.2 are displayed in figure 2. The direction for search is UP-LEFT-RIGHT-DO WN, unless stated otherwise. Generally, the algorithms behave as expected. That is, BFS branches out in ripples from the Origin(0,0) and attempts to traverse the maze in expanding rings from the start. DFS, goes directly to the goal; this happens as the chose search direction aims the algorithm directly to the goal (end,end). DFS attempts to go as down as possible and then as right as possible, which coincidentally, is its goal! A* behaves in between BFS and DFS, with the manhattan Heuristic outperforming the Euclidean significantly. This is expected as the manhattan heuristic is the natural distance metric for out problem, and A* depends heavily on an accurate distance heuristic. Finally, **Bi-Directional BFS** exhibits a a symmetric, mirrored path from its start points origin and goal. The ends meet at a single point in the middle left and the algorithm exits.

Bullet c

If we select as a cutoof point for "most mazes are solvable" a success probability of 66%, we can identify the probability of occupancy for reasonable success p_0, for each algorithm, in table 1.

Bullet d

Bullet e

Yes as as its displayed in the figure below, a heuristic weight given from the manhattan distance far outperforms A* using euclidean distance. This is because in out problem space manhattan fits naturally to our problem; we can only move up, down, left or right, on a square connected grid with homogeneous weights. In contrast, euclidean distance considers a straight line from a position to the goal, so diagonal elements are deemed closer than in the manhattan case, as we consider that we can actually move diagonally; but in this maze, we cannot. This is showcased in figure 4, where it is evident that A* with euclidean distance branches out much more than A* with manhattan.

Bullet f

They do. They navigate the maze, trying to go to nodes that are optimally close to our origin but are also getting closer to the goal, according to our heuristic distance. As the manhattan dist is the better metric for the discirite maze traversal, when A* uses it it converges to the solution faster.

Bullet g

Yes, as discussed before, providing a search direction that inherently points towards the goal will lead to vastly improved results. This is because going in depth towards the general direction of the goal will shorten the running time of the path search, as we are not interested in discovering the entire maze but escaping! So, considering nodes that are away from the goal is detrimental to the efficiency of DFS in this scenario. This is showcased in figure 5.

Bullet h

Yes, Bi-Directional BFS (or BiBFS) will consider cells further to the size than A* using manhattan distance. The reason for this is that BiBFS just branches out in expanding circles from the start point with no sense of direction. In sharp contrast A* leverages its heuristic distance from the goal and its minimum distance plaining from the origin to move optimally towards the goal at each step. 6.

Part 3

In this part we explore techniques to generate increasingly harder mazes.2 techniques were finished:

1. Random Walk
2. Recursive Maze hardening

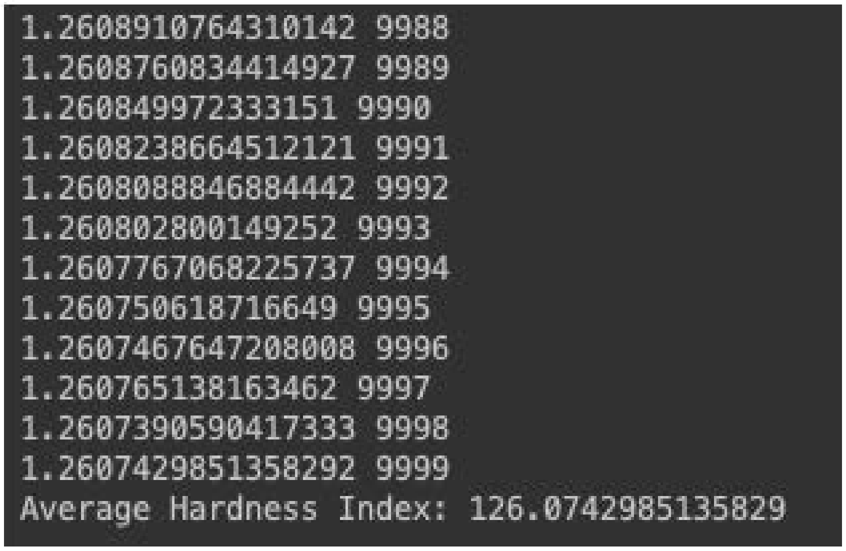
Random Walk is as described in the assignment description. The results for an iterative process of 1000 trials are displayed in figure 7.
Recursive Split Hardening is a technique where the maze is, at each step, split by two perpendicular lines at random locations. Then a number of 'doors' are opened; that is at 3-4 random locations along the split lines for this step, the wall is removed to create a passage. The process is continued until the areas to split are too small, i.e their size is less than 2x2. The resulting harder map is also displayed at figure 7.

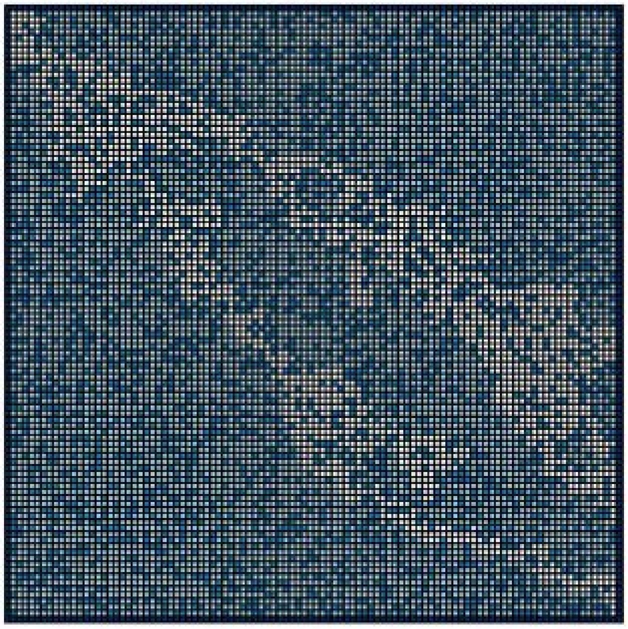
Idea 1:
Recursive Split Hardening is a technique where the maze is, at each step, split by two perpendicular lines at random locations. Then a number of 'doors' are opened; that is at 3-4 random locations along the split lines for this step, the wall is removed to create a passage. The process is continued until the areas to split are too small, i.e their size is less than 2x2. The resulting harder map is also displayed in figure 7.

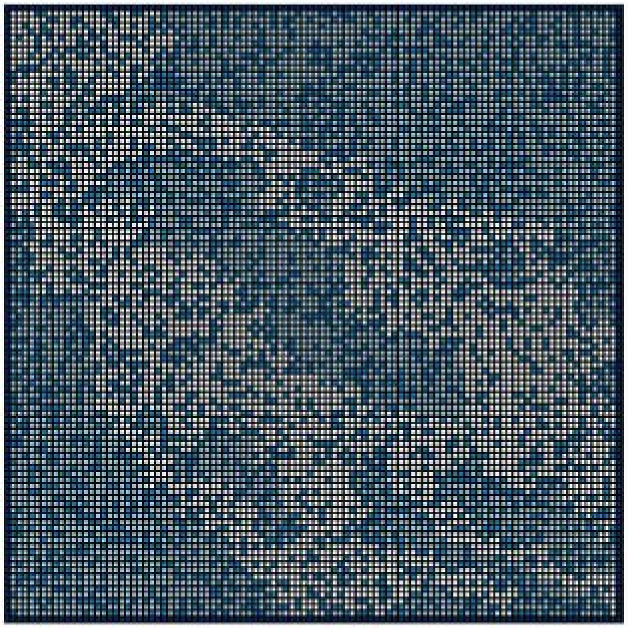
Idea 2:
Hill Climbing Algorithm is a classical local search algorithm that is trying to find out a better state near the current state. And in this case, we utilize Maximum Fringe Size in DFS and Maximum Nodes Expanded in A* - Manhattan as parameters into this algorithm. The greater the values of these two parameters mean that the maze would be harder. So, in our code, by comparing the values of these two parameters and put them and their corresponding maze into a priority queue and trigger our ‘flip’ function over and over again.

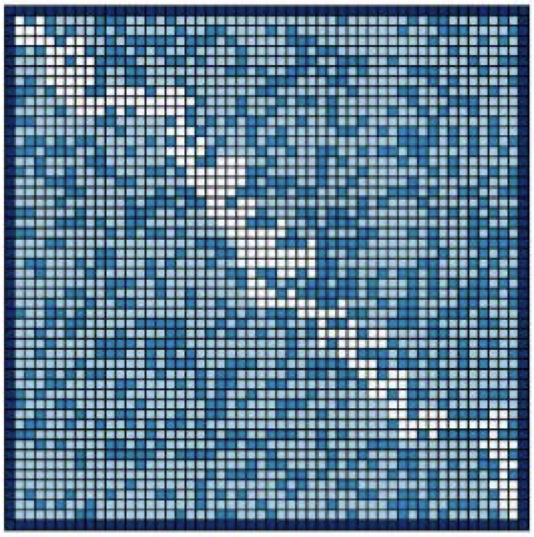
Hill Climbing is trying to help us to find greater values and moving to the harder state recursively. When there are no harder cases in anther word, the priority queue is empty which means we’ve already got to the local maximum. Then we jump out of the search recursion and the maze we get is kind of the hardest one we can get by the local search algorithm. As we all know that the most annoying drawback of this algorithm is that we can only find out the local maximum.

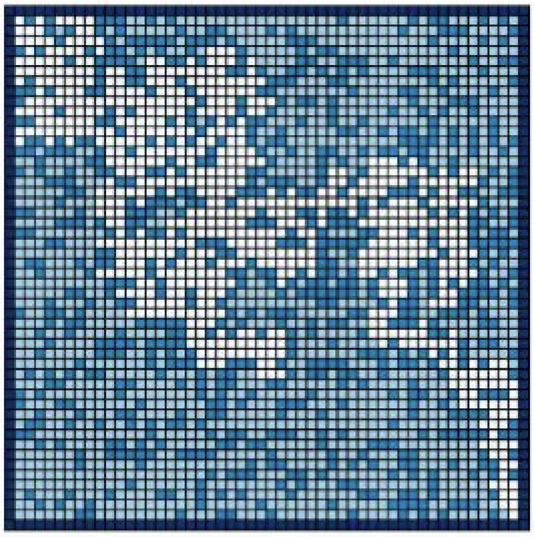
By running a maze-drawing function, we can clearly see that after the recursive maze hardening, the path we expanded becomes more complex which means the maze becomes harder. And we still use an index named ‘hardness’ -- the cells we need to expand in the new maze comparing to the old maze. We test a 10 * 10 maze for 10,000 times and the average value of the hardness index is near 126 which means we increase the hardness by 26%.


```
1.2608910764310142 9988
1.2608760834414927 9989
1.260849972333151 9990
1.2608238664512121 9991
1.2608088846884442 9992
1.260802800149252 9993
1.2607767068225737 9994
1.260750618716649 9995
1.2607467647208008 9996
1.260765138163462 9997
1.2607390590417333 9998
1.2607429851358292 9999
Average Hardness Index: 126.0742985135829
```


Maze: 100 * 100 path with A* before


Maze: 100 * 100 path with A* after


Maze: 50 * 50 path with A* before


Maze: 50 * 50 path with A* after

Part 4

In this section, we use A* search using manhattan distance as its heuristic function to find the shortest path of the random generated maze. in this part, we set the p0 for obstacle is 0.5.the mark of obstacle is 2, fire spot is 5, and using 3 to represent the node we has been to.

Idea of improvement

The success rate of the functions are as below:
dimension: 20, possibility of the obstacle :0 .5

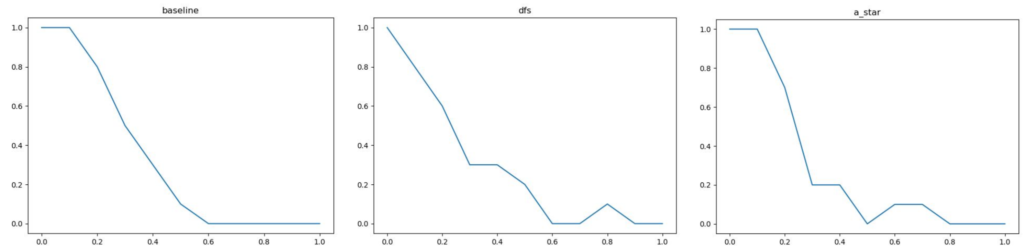
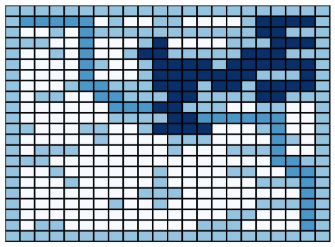


figure:the results of baseline, dfs and A*

In this section, we use three functions to test the possibility of solving a maze which is on fire.

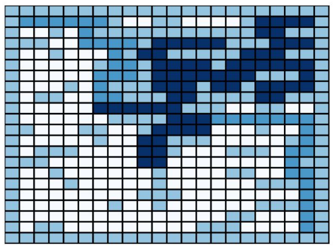
Baseline

When fire spreading speed is 0.5, the path or robot and fire is below.



dfs

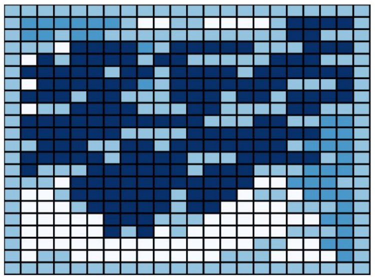
When fire spreading speed is 0.5, the path or robot and fire is below.



This method is based on dfs, we use dfs to simulate the robot who tries to find a way out, if he detects there is fire or obstacle on his way, he simply turned to another direction.

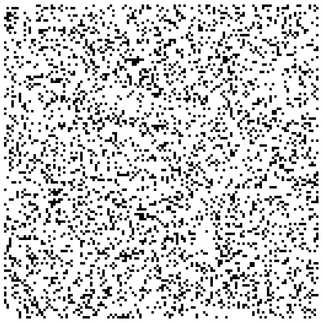
A*

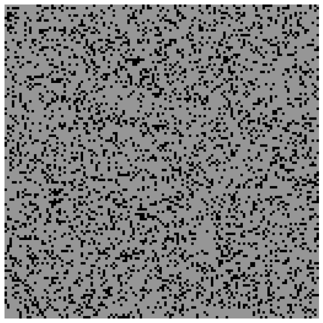
When fire spreading speed is 0.5, the path or robot and fire is below.

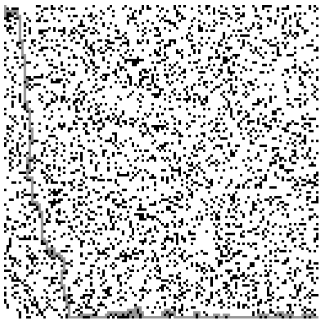


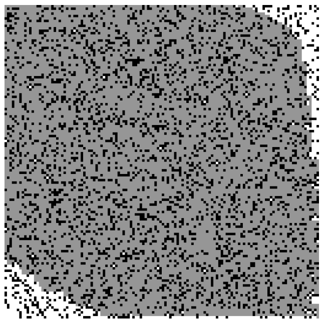
The difference between this method and the original method is that we change the heuristic function, the original function is manhattan distance or euclidean distance. But in the method, we will add some weights depends on the distance between the fire spot and the robot. If the fire is adjacent to the robot, the robot won't step on it and it won't burn our robot at the next turn, so the danger weight is assigned to zero. Then, if the fire is one block away from the robot, it is the most dangerous situation because the fire and robot step towards each other, the robot will burn itself, so we assign the weight to be 2. The last situation is the fire is 2 block away from the robot, it is kind of dangerous so we assign the weight between the first and second situation, and it will be 1.

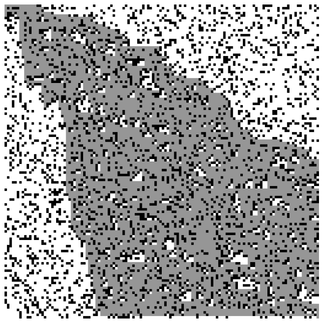
We ran the three different algorithms on 100 different solvable maze, test the possibility of the fire spreading between 0 to 1.0, and the figure above are the results. As you can see, when the spreading speed is too low or too fast, the performance of the three different algorithm is similar. But when the possibility is around 0.5, A* and dfs works better than the baseline.


(a) Original maze.


(b) BFS.


(c) DFS.


(d) A* with Euclidean Distance.


(e) A* with Manhattan Distance.

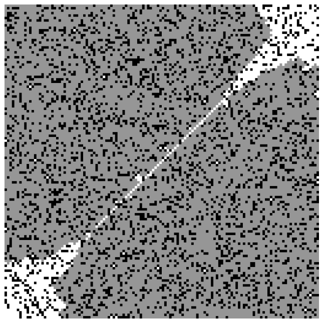
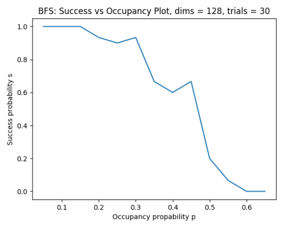
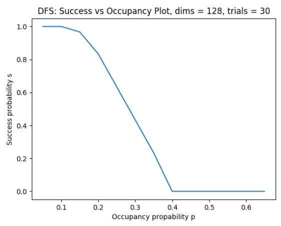

(f) Bi-Directional BFS.

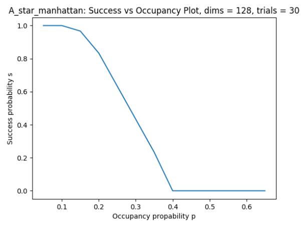
Figure 2: Bullet 2: All the Str ategies for a solvable map.



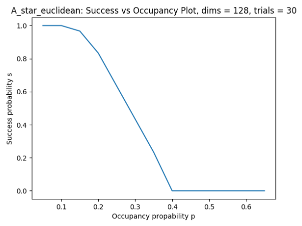
(a) BFS.



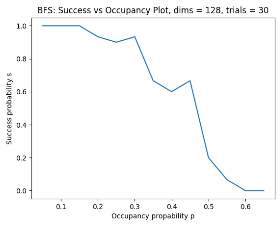
(b) DFS.



(c) A* with manhattan distance.

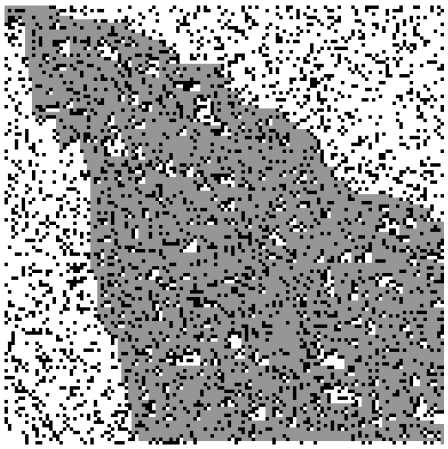


(d) A* with Euclidean Distance.

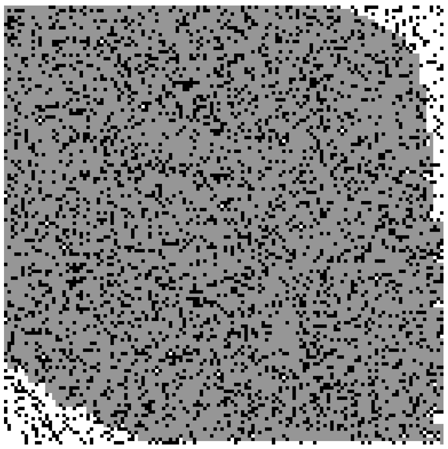


(e) Bi-Directional BFS.

Figure 3: Bullet C(3). Success vs occupancy probability curves.

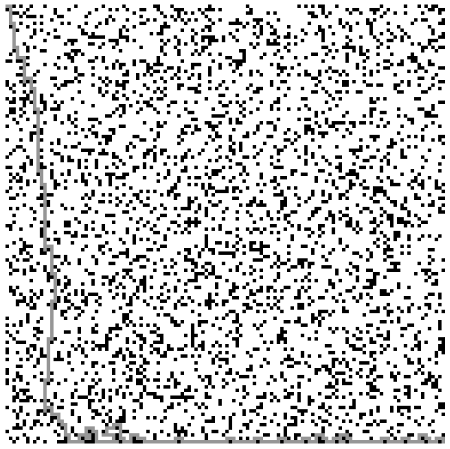


(a) DFS prioritizing goal direction.



(b) DFS prioritizing direction away from goal.

Figure 4: Bullet e(5): A∗ euclidean vs manhattan performance comparison.


(a) DFS prioritizing goal direction.

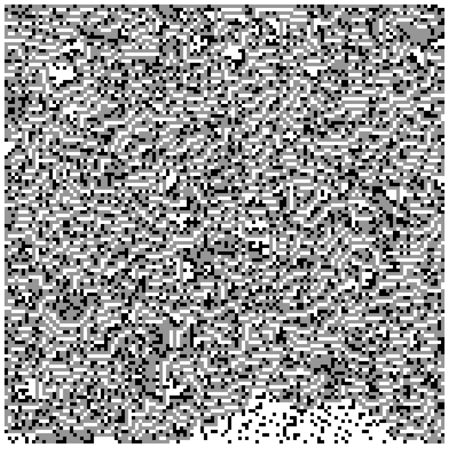
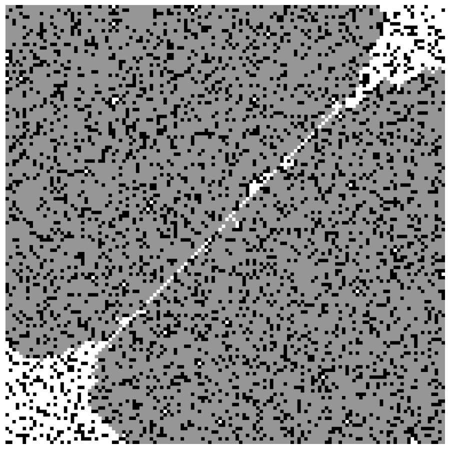

(b) DFS prioritizing direction away from goal.

Figure 5: Bullet g(7): DFS search direction comparison.


(a) Bi-Directional BFS.

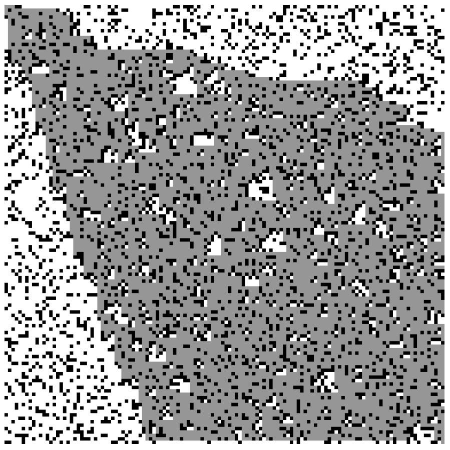
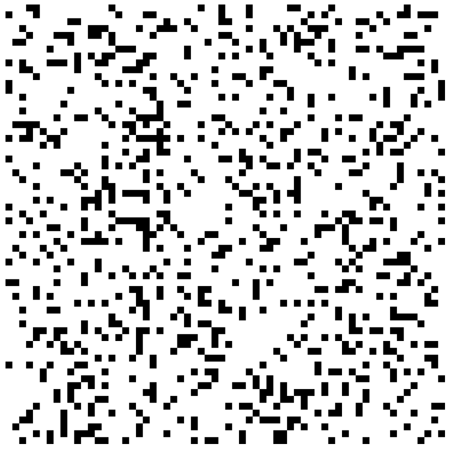
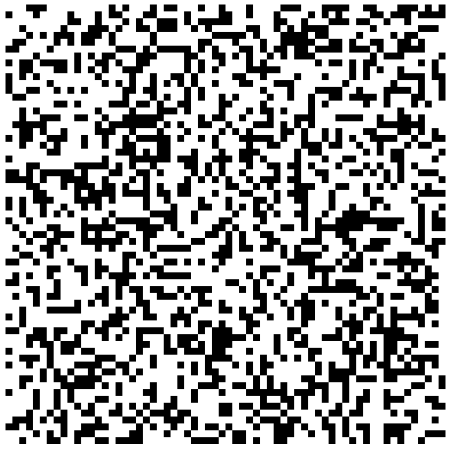
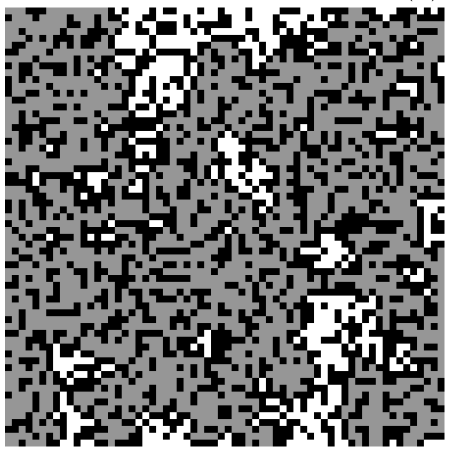

(b) A* with manhattan distance.

Figure 6: Bullet h(8): Bi-Directional BFS vs A* with manhattan distance.


(a) A sample maze.


(b) Same maze after 1000 iterations of harderning.


(c) DFS traversal of the hardest random walk maze.

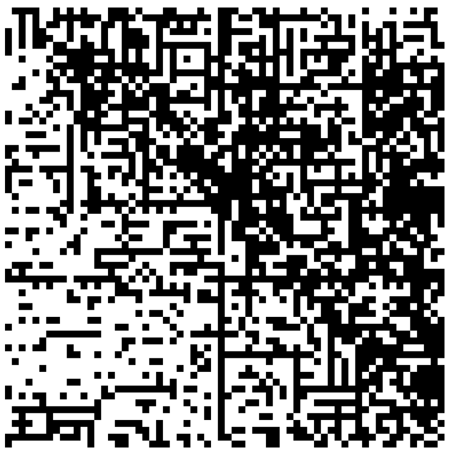

(d) Recursive Split maze hardening.

Figure 7: A size by side by side display of randomwalk and recursive hardening along with the solutions.

The team members of this group are Zekun Zhang, Ziqi Wang, and Richa Rai. The three members discussed how to best implement the concepts in code and compared their different ideas. They then worked together to compile the project and discuss the analysis components.