**CS 520: Assignment 1 – Path Planning and Search Algorithms**

Group members:

|  |  |  |
| --- | --- | --- |
| Name | netID | Work |
| Yan GU | yg369 | A\*/Experiments of part1/Write-up of part1 |
| Yunfan LI | yl1269 | Visualization/Write-up of part1/Organization of write-up |
| Xueyu WU | xw318 | Random map generation/DFS/GA algorithm/Experiments of part2 |
| Junlin LU | jl2364 | BFS/Write-up of part2 |

**Part 1**

**Generating Environments:**

In this part, we set up a function named ***maze\_generate***with two variables: dim for dimension of mazes and ***p*** for probability of a cell being occupied. We leave all obstacles with 1 in our maze arraies (showing as black blocks in visualizations). The main goal for path planning is to find a route from upper left to lower right corners. In the following quesitions, we will discuss the most practical value of dimension and probability to make mazes more interesting and feasible. And then we are going to analyze advantages and disadvantages in different search algorithems.

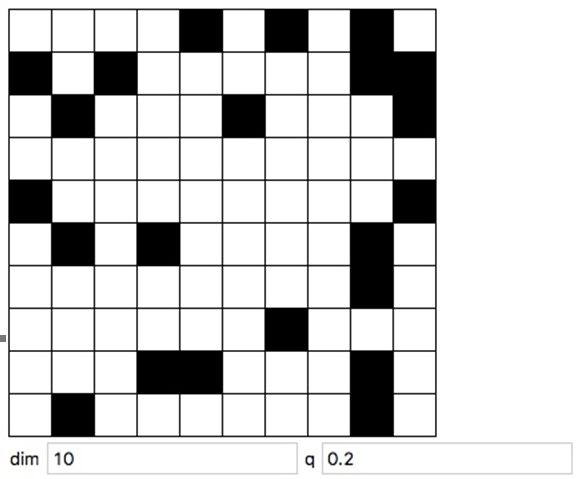


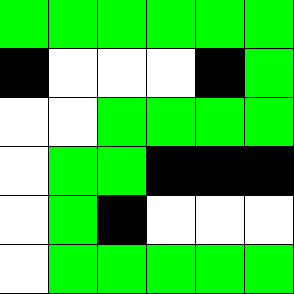
Figure 1-1 Randomly generated with dim=10 and p=0.2

**DFS:**

Depth-first search is a basic and simple algorithem for graph search, and this idea is really easy to implement using stack. It starts at the beginning and goes as far as it can(push in) down a certain path, then backtracks(pop off) until it find an unexplored path, and do the same thing again until it finds the end(top is the end cell) or the entire graph has been explored(stack is empty).

DFS is often used in complex graph search algorithems since it has less time and space complexity than BFS. Solution can be found without much more search and memory. While it does not guarantee that the found path is the shortest. Because DFS has a certain order in searching direction(like right-down-left-up), it will happen that DFS goes some detours between begin and end.Below is a simple example of this.

DFS path:



Shortest path:

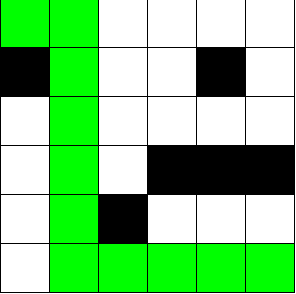


Figure 1-2 Comparison of DFS path and shortest path

**BFS:**

The main idea of breath-first search in a maze is to explore from current location to four direction each time until it reaches the exit of the maze.

Firstly, we set an array to indicate the next possible step: [[-1, 0], [0, 1], [1, 0], [0, -1]]. We need several arrays, one is to store the visited path, two for x-coordinate and y-coordinate, one for current steps’ last step (father nodes) and one for the coordinate of the nodes that are expanded.

Secondly, from upper left corner, we explore its next steps. According to what the maze is designed, if the next step is not the lower right corner and is marked 0, it means we can take this step. We store the x-coordinate and y-coordinate of this step, record this node and its father. We repeat this process until the next step reaches the lower right corner. Take a 3 x 3 maze for example (see Figure 1-3 ), as the steps indicated above, it expands 5 levels of nodes and the path is [0, 0] -> [1, 0] -> [1, 1] -> [1, 2] -> [2, 2].

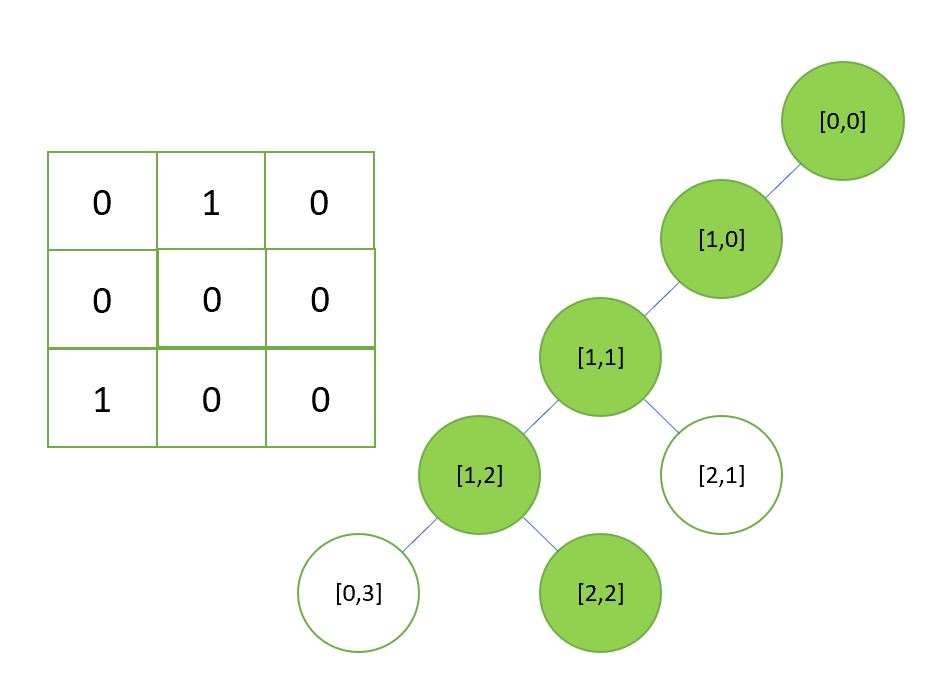


Figure 1-3 Example of BFS

At last, we return its shortest path (The first path that reaches lower right corner is the shortest path) and the nodes that are expanded by backtracking the father nodes.

**A\* :**

Figure 1-4 shows a classic example of A\* ALG. We use a closed set to maintain the nodes we have already traveled. And when we arrive a node, we will discover its neighbor and store them in a min-heap called *open set* so we can always pop the neighbor with minimum ***f*.**

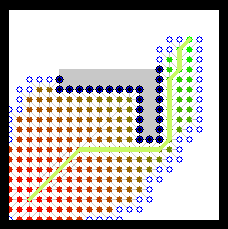


Figure 1-4 Example of A\* algorithm

But we found that there may be a tie when there are some nodes in the *open set* have same ***f*** value, by default, we will pop the first node in the *open set*. For example, using the Manhattan Distance, the following picture states a tie. In a 5\*5 grid, when we arrive node *A*, we can easily know that node *B* and *C* is the first and second node in the *open set*, but ***f****B*= 1+(5-1)+(5-2) = 8, ***f****C*= 2+(5-2)+(5-2) = 8.

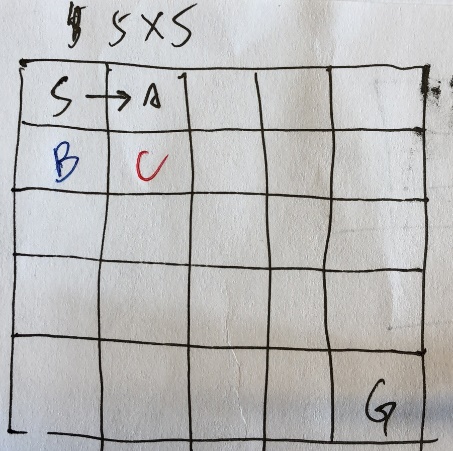


Figure 1-5 Demonstration of tie-breaking method

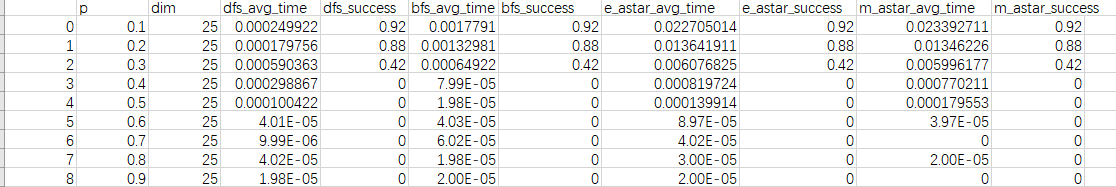
Figure 1-5 is a simple demonstration of the tie-breaking method we used. Obviously, C is better than B, but B may be pop in front of C, because we can never directly know which node is better in a larger grid, so we need a tie breaker. We choose a strategy that when several nodes have same ***f***, we always choose a node with higher ***g*** value since we think it may be closer to the goal. Using this tie-breaking method, we can expand fewer nodes than usual.

**Questions:**

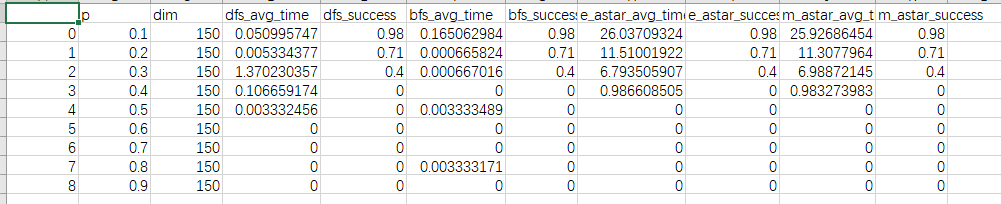
1. We test for each p = 0.1, 0.2, …, 0.9, and in each step, we test for dim = 20, 25, 30, …, 200. We found that when p = 0.1, the time cost is biggest compared to other p value. And the probability for finding a path is almost the same between different dim. I put some of the test data in the submission. \_avg\_time is the time for each finding process in seconds. \_success is the probability for succeeding in finding a path.

Here are some examples, based on them, we choose dim = 160 for the following test of **Part 1**.

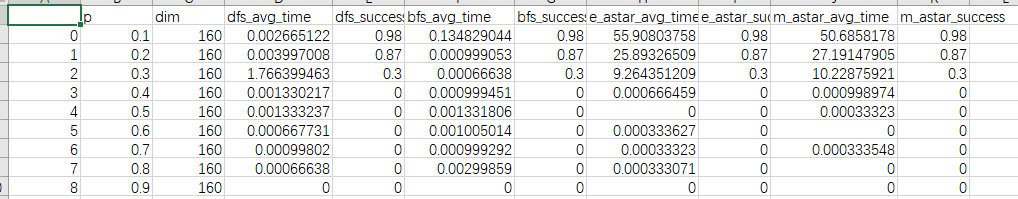
**Dim = 25**



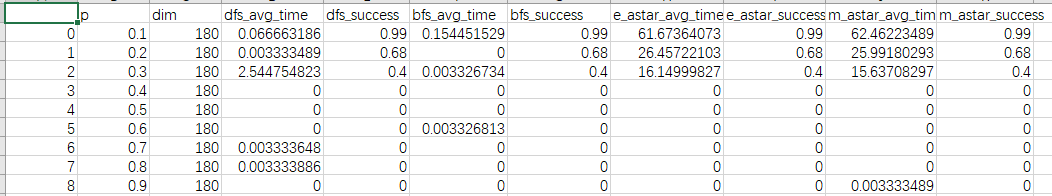
**Dim = 150**



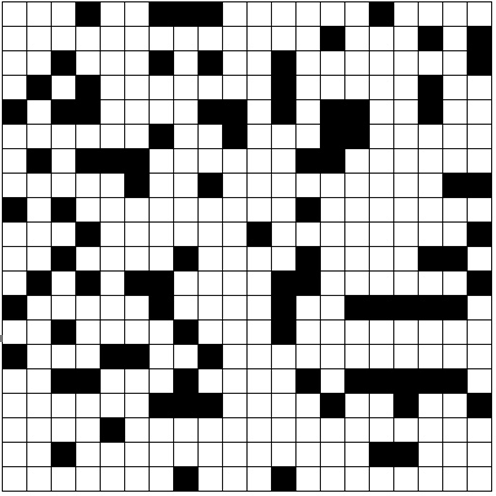
**Dim = 160**



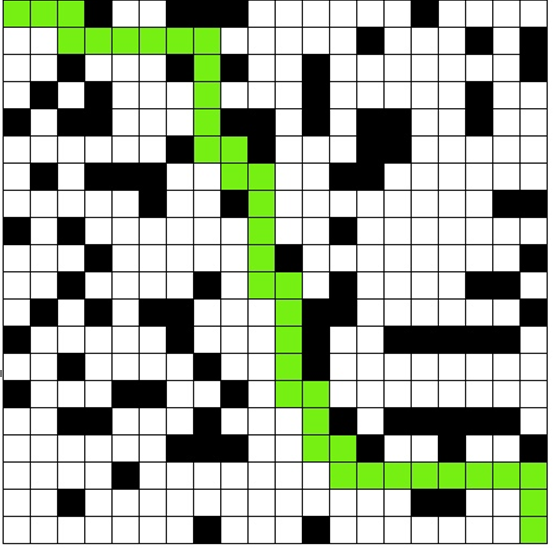
**Dim = 180**



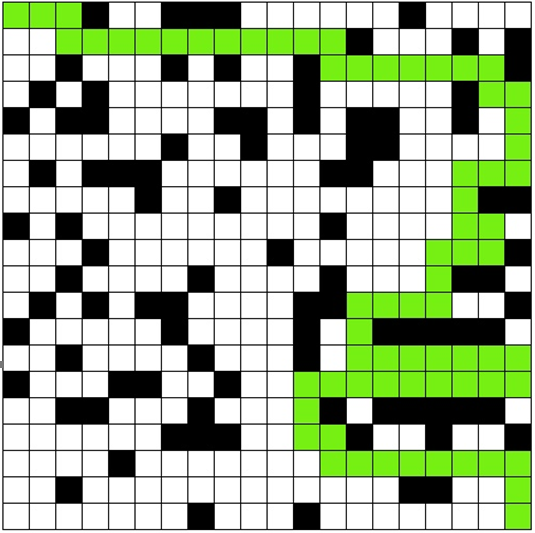
1. Genarated map:



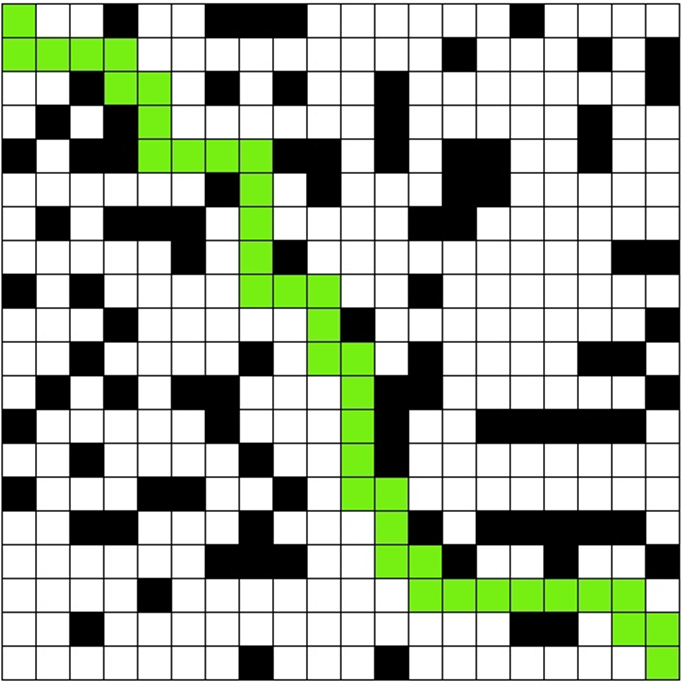
Path by BFS:



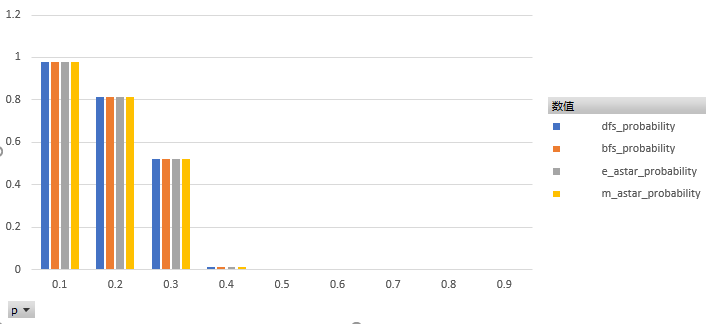
Path by DFS:

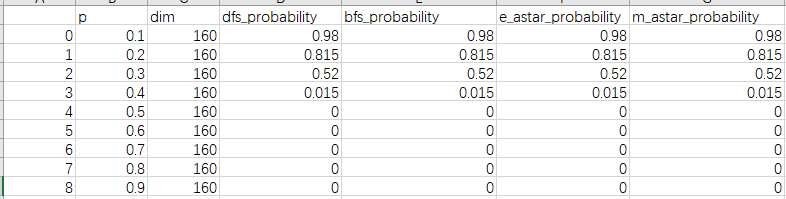


Path by A\*(path for Manhattan and Euclidean are the same in this case):



1. For each p, we use 1000 random mazes to test, the probability for finding a path shows below.

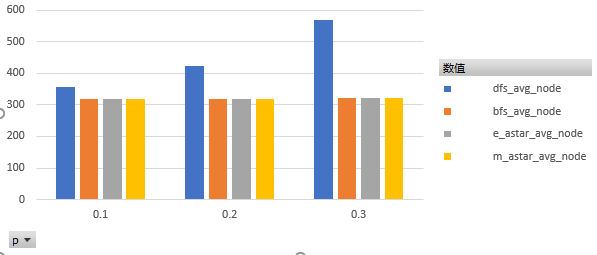


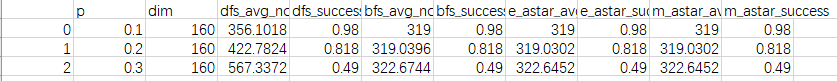


So for all four ALG, we choose p0 = 0.3

1. We test 1000 random mazes, from the data we can easily know that the BFS and two A\* have the same shortest path and the expected length of the shortest path for DFS is larger than the other three. And BFS is slightly larger than two A\*. So in this question, we can conclude that the two A\* are the most useful ALG.

\_avg\_node represents the average length of path, \_success represents the probability of finding a path.





1. Use the data we already have in Question 4, the length of path generated by DFS is obviously larger than two A\*, and the lengths of path generated by two A\* are the same.

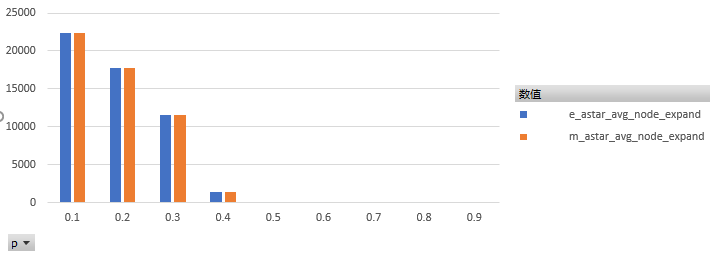
And as the p getting bigger, the gap between DFS and A\* is getting much bigger.

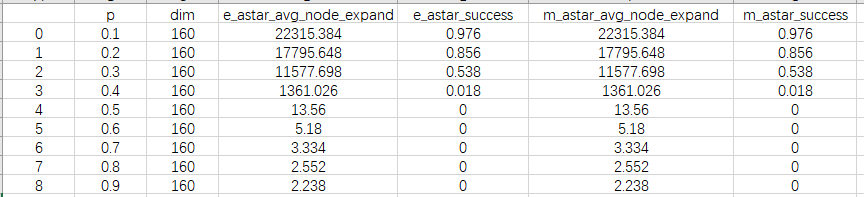
1. Because we use a tie-breaking strategy beforehand, so the total number of expanded nodes will be the same for either heuristic since the influence caused by different heuristic in choosing next step is dropped off by the tie breaker. The tie breaker will force two algorithm to do the same *right* thing, that is, to get closer to the goal as far as possible.

So for all p, there is no difference.

\_avg\_node\_expand represents the average number of total nodes expanded.

\_success represents the probability of finding a path.



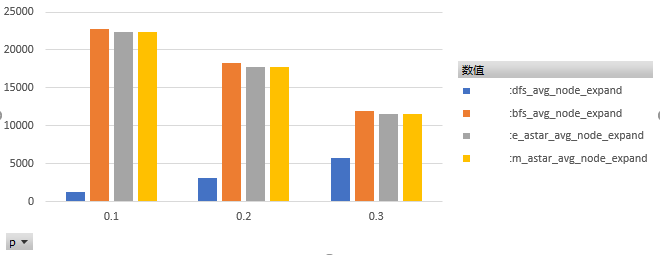


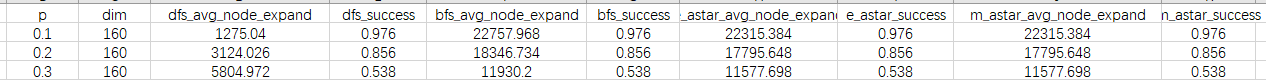
1. The DFS expanded the fewer nodes than other three, because it finds the first path from start to the goal regardless whether the path is the shortest. But BFS and two A\* aim to find the shortest path, so they will traverse a larger part of grid than DFS.

Data shows below.

\_avg\_node\_expand represents the average number of total nodes expanded.

\_success represents the probability of finding a path.





Bonus 1)

**Part 2**

1. We use genetic algorithm to generate a hard maze. The basic idea is that we generate a population with a lot of normal mazes. Each time we calculate the fitness of mazes, that means the harder the maze is, the higher the fitness is. Generally, we want to choose hard mazes to produce new mazes, so it is wise to select mazes by roulette according to their fitness. Within new population consisted by selected hard mazes, they reproduce a number of new mazes with their original characters by exchanging some parts of them. In addition, we sometimes randomly change the value of cells in a maze, like mutation in the nature, which may bring good genes into this population. And then we do selection again. After repeating this process for enough times, we can get a bunch of relatively hard mazes in the population.

In part 1, we choose dim that is greater than 100, but in this part, we try to generate a maze that is as hard as possible, thus we need to do enough times of reproduction and selection until it reaches a highest value (if this value exists). Therefore, as much as we want to make a huge maze, we can only afford to set the dim at the value of 25 as it takes time to generate the “hardest” generation. But all in all, if there is a point where we reach the highest value, this algorithm can be proven successful no matter what the dim is set.

Firstly, we calculate the *fitness* of mazes. The fitness is either the total length of the shortest path, total number of the nodes expanded, or the total maximum size of the fringe of all three algorithms (DFS, BFS, A\*), depending on what the *goal* is. Then we try to get the biggest discrepancy by subtracting the smallest one. For example:

If our goal is node, we add nodes that are expanded by DFS, BFS and A\* of each maze. Among those mazes, we have mazes that expand nodes in the number of 100, 101, 102, 103. Although their discrepancy are small, certainly 103 is a little better than others. So if we want to maximize the chance for the 103 to be selected, we minus the smallest number for each number and calculate their cumulation probability, which is 3/(0+1+2+3), then we will have 50% chance to select this relatively harder maze rather than 103/(100+101+102+103), which is about 25%.

Secondly, we do the *selection*. For those mazes that are harder, we put them into the population. Then we *crossover* them two by two, meaning that we move the rows and columns to combine them. To avoid getting local optimal result, we use *mutation*, which is adding new characters by editing cells in mazes.

By doing 1000 times of generation, we can gradually reach a steady peak that will not change. We can say that we get a hard maze.

1. From the Figure 2-1 to 2-3 below, we can see that after hundreds of generations of selection, the maze sort of reaches a peak. The following generations don’t change significantly. If the paths, nodes or fringes vary less than 1% in 200 consecutive generations, we conclude that this may be the hardest maze that we have been looking for.

However, there exists a mutation that could affect the result. For example, after 500 generations, there could be a mutation that generates a much harder maze, then the whole process could reach a new peak, see Figure 2-1. What’s more, mutation could happen in early time but its characteristic may not pass to next generation. Thus, the hardest maze may already appear in early time, see Figure 2-3.

Figure 2-1 The change of shortest path in 1000 generations

Figure 2-2 The change of nodes expanded in 1000 generations

Figure 2-3 The change of maximum fringe in 1000 generations

As mentioned before, we may not select the hardest maze in one generation into next population due to probability, so we might not reach another higher peak. All we get is a local optimal result. I do not think it is just the shortcoming of genetic algorithm, but it is the main problem of local search. However, we have to admit that mutation and crossover do bring a lot of better results for us. And for time and space complexity, it is an easy way to gain a relatively good answer, especially for mazes since they are made of 0 and 1, we do not have to translate genotype to phenotype, and they are matrix which are easy to crossover.

Goal: path

If the goal is to generate the maximum length of the shortest path, the lengths paths of each algorithm after 1000 generations are as Table 2-1.

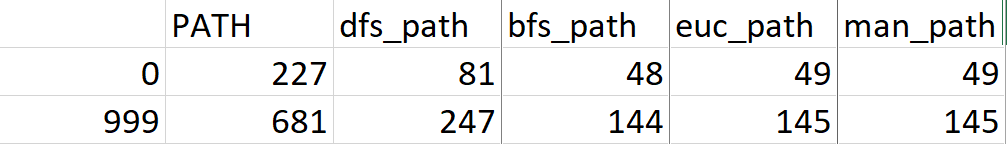


Table 2-1 The shortest path of each algorithm after 1000 generations

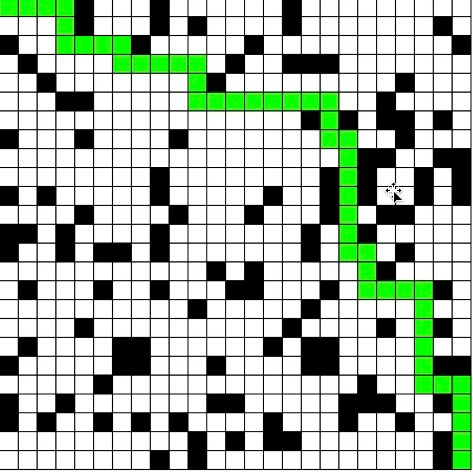


Figure 2-4 Generated map after 0 generations (path/BFS)

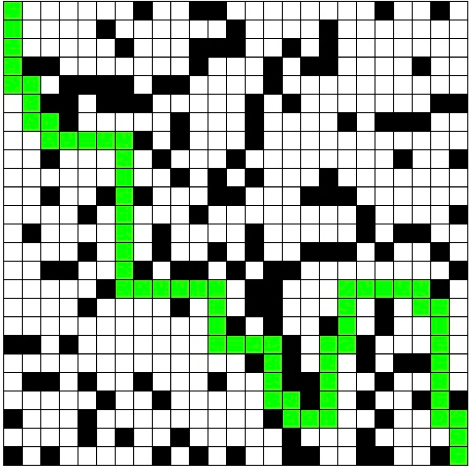


Figure 2-5 Generated map after 100 generations (path/BFS)

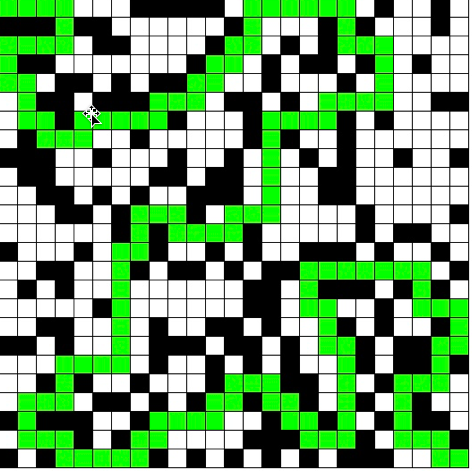


Figure 2-6 Generated map after 1000 generations (path/BFS)

Similarly,

Goal: node

If the goal is to generate the maximum amount of the nodes expanded, the lengths paths of each algorithm after 1000 generations are as Table 2-2.

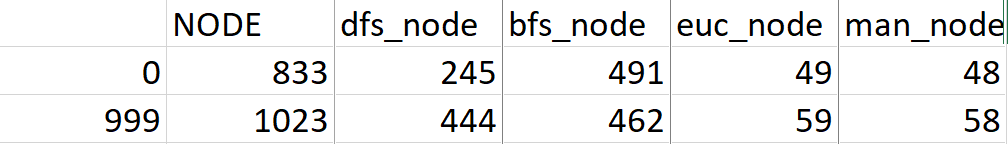


Table 2-2: The nodes expanded of each algorithm after 1000 generations

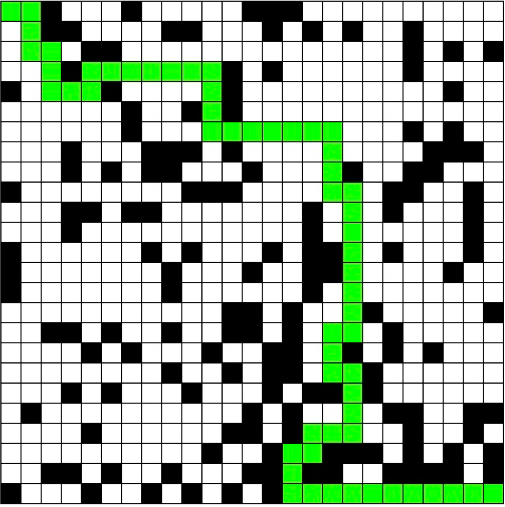


Figure 2-7 Generated map after 1000 generations (node/BFS)

Goal: Fringe

If the goal is to generate the maximum size of the fringe, the lengths of paths of each algorithm after 1000 generations are as Table 2-3.

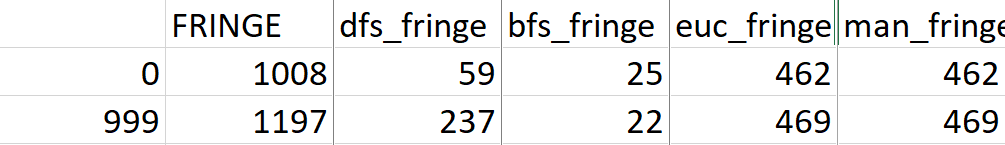


Table 2-3: The maximum fringe of each algorithm after 1000 generations

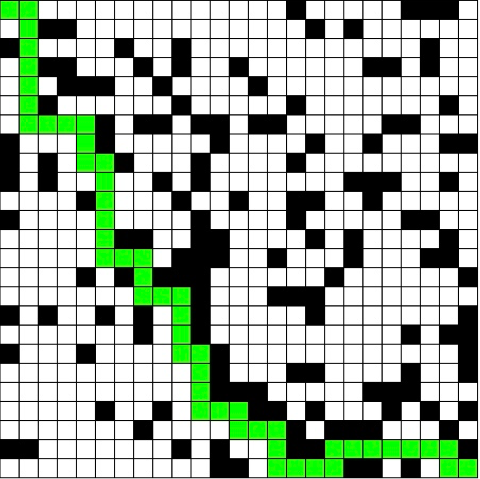


Figure 2-8 Generated map after 1000 generations (fringe/BFS)

From the tables above, we can see that if we use DFS to solve a maze, it is very easy to make our maze more difficult to solve. But for BFS and A\*, using genetic algorithm to generate a hard maze for them to solve is not very sufficient, especially when our goal is to expand more nodes or generate a larger fringe. After thinking and discussion, I think since BFS and A\* using queue to search path, they can easily decide to not go to that way by search all feasible cells in their neighbors although there are some blocks. However, DFS goes as deep as it can, so it is more likely to go some detours or dead ends.