Maze Runner

Rroup Members: Yaxin Zhu(yz956), Xiaochen Gao(xg142), Jiawei Wu(jw1308)

1 Enviornments and Algortihm

1.1 Generating Environments

The maze is sorted in a d*d matrix(d represents the dimension of the maze), if the value is set to 1, it means that there is a wall at this point. We generate the maze using np.random.binomial() function in python.

Also we implement a function based on matplotlib to visualize the maze and the path we take to solve the maze.

1.2 Path Planning

1.2.1 Depth-First Search and Breadth-First Search

In DFS and BFS we have a data structure to store visited nodes in the maze grid, then visit them from the start node following given rule that only neighbor nodes on left/right and up/down directions can be visited next. The difference is, we use stack in DFS, which means last visited nodes will go out first, and in BFS we use a queue, which means first visited nodes will go out first.

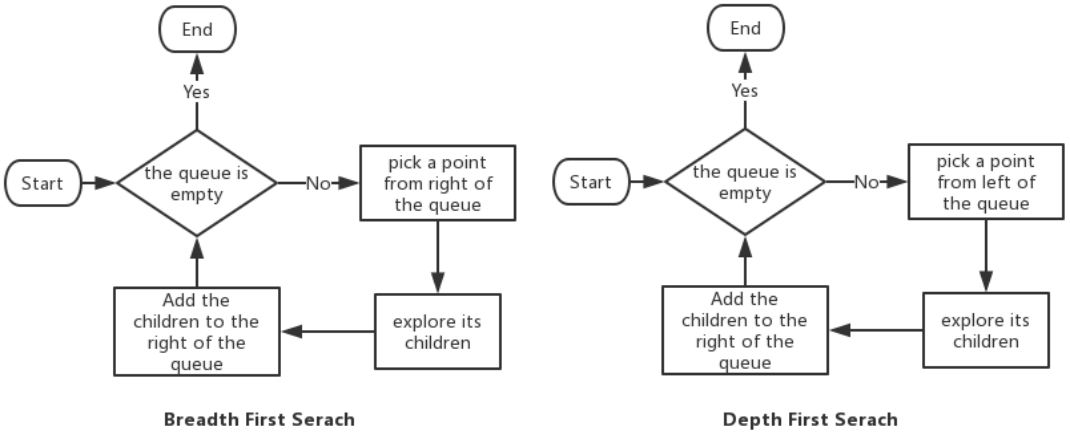

Fig 1.1 BFS and DFS

In complete DFS or BFS execution we equally go through all nodes. For children connected by the parent node their visiting order are just randomly determined. Although we gave it an order in implementation, it's a random path equally to all other paths not given by the algorithm.

1.2.2 A* (Euclidean Distance And Manhattan Distance)

Unlike DFS and BFS, A* Algorithm merge the information of nodes when performing search by introducing the heuristic function. (Calculating Euclidean Distance or Manhattan Distance).

We use the PriorityQueue provided by Python to implement the algorithm.

1.2.3 Bi-Directional Breadth-First Search

Different from the traditional searching strategy that the Previous Algorithms, Bi-Directional Breadth-First Search(BD-BFS) chooses to search from both sides of the solution tree. This searching strategy can not only find the shortest path like BFS, but Reduce the running time by cutting the searching into two parts. In this case, we simply start searching from both the starting point and goal point. The flowchart of BD_BFS we implemented is shown as follows.

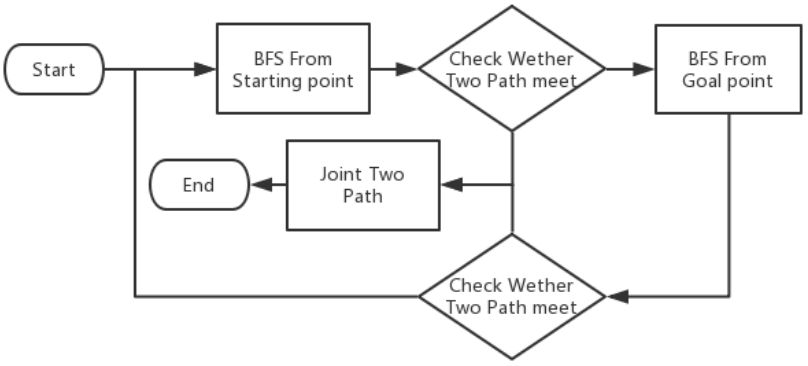

Fig 1.2 BD-DFS

There are two ways to check whether two paths meet, the first way is to use Intersection() function in Python. However, the running time of Intersection() is increasing tremendously as the size of maze increases, resulting in BD_BFS running much slower than the traditional searching algorithm. The second way is to use a matrix to mark whether a point in the maze is already visited. So when we expand the neighbor of a point and find it already visited, we can say that the two paths meet at this point.

In addition, when we add a node into fringe, we determine whether it is reasonable to join the node (for example, obstacles, boundaries, and points that have been reached by the path), which increases the efficiency of our algorithm and avoids loops. This operation reduces the number of child nodes and greatly reduces the max fringe size, which is very obvious on some algorithms with many extension nodes(BFS, A*EU).

2 Analysis and Comparison

2.1 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?

We implemented the four algorithms DFS, BFS, A*EU, A*MH, and BDBFS and run them on the maze we generated, and recorded the time the algorithm was running. The time records for these algorithms are as follows, and the time representation is in seconds.

Algorithm	Dim=10	Dim=50	Dim=100	Dim=250	Dim=500	Dim=1000
DFS (p=0)	0.0003	0.006	0.03	0.15	0.58	2.27
DFS (p=0.1)	0.0004	0.005	0.024	0.127	0.57	2.24
DFS (p=0.2)	0.0004	0.007	0.021	0.169	0.67	2.33
DFS (p=0.3)	0.0004	0.006	0.022	0.135	0.61	2.22
BFS (p=0)	0.0007	0.012	0.052	0.368	1.56	6.77
BFS (p=0.1)	0.0005	0.017	0.058	0.357	1.52	6.06
BFS (p=0.2)	0.0005	0.012	0.044	0.321	1.28	5.42
BFS (p=0.3)	0.0005	0.01	0.041	0.29	1.16	4.86
A*EU (p=0)	0.004	0.066	0.258	1.54	6.66	27.9
A*EU (p=0.1)	0.0018	0.054	0.219	1.34	5.76	26.26
A*EU (p=0.2)	0.0016	0.047	0.21	1.21	5.05	22.03
A*EU (p=0.3)	0.0024	0.046	0.23	0.99	4.3	18.63
A*MH (p=0)	0.0007	0.007	0.025	0.18	0.62	2.38
A*MH (p=0.1)	0.0007	0.01	0.03	0.28	0.95	4.38
A*MH (p=0.2)	0.0007	0.01	0.04	0.23	1.1	3.85

Algorithm	Dim=10	Dim=50	Dim=100	Dim=250	Dim=500	Dim=1000
A*MH (p=0.3)	0.0003	0.01	0.05	0.24	1.6	4.5
BDBFS (p=0)	0.0002	0.007	0.029	0.204	1.01	4.43
BDBFS (p=0.1)	0.0002	0.005	0.028	0.174	0.83	3.78
BDBFS (p=0.2)	0.0002	0.005	0.024	0.146	0.78	3.22
BDBFS (p=0.3)	0.0002	0.004	0.015	0.107	0.547	2.42

Form 1.1 running time of the algorithms

From the table above we can see that when the dim is 100, the running time of each algorithm under all p values is less than 1 second, which allows us to run the program many times in a short time. Moreover, this dim is not too small to make the experimental results uninteresting. So we think 100 x 100 is a suitable size of maze.

2.2 For p=0.2, generate a solvable map, and show the paths returned for each algorithm.

When p=0.2, dim=100, we have run five search algorithms on the solvable maze, and the resulting path is as follows, where the red line represents the path, the black grids represent the obstacles, and the gray grids represent the expanded nodes.

Depth First Search:

Max Fringe Size: 207
Path Length: 235
Total Expanded Nodes: 236

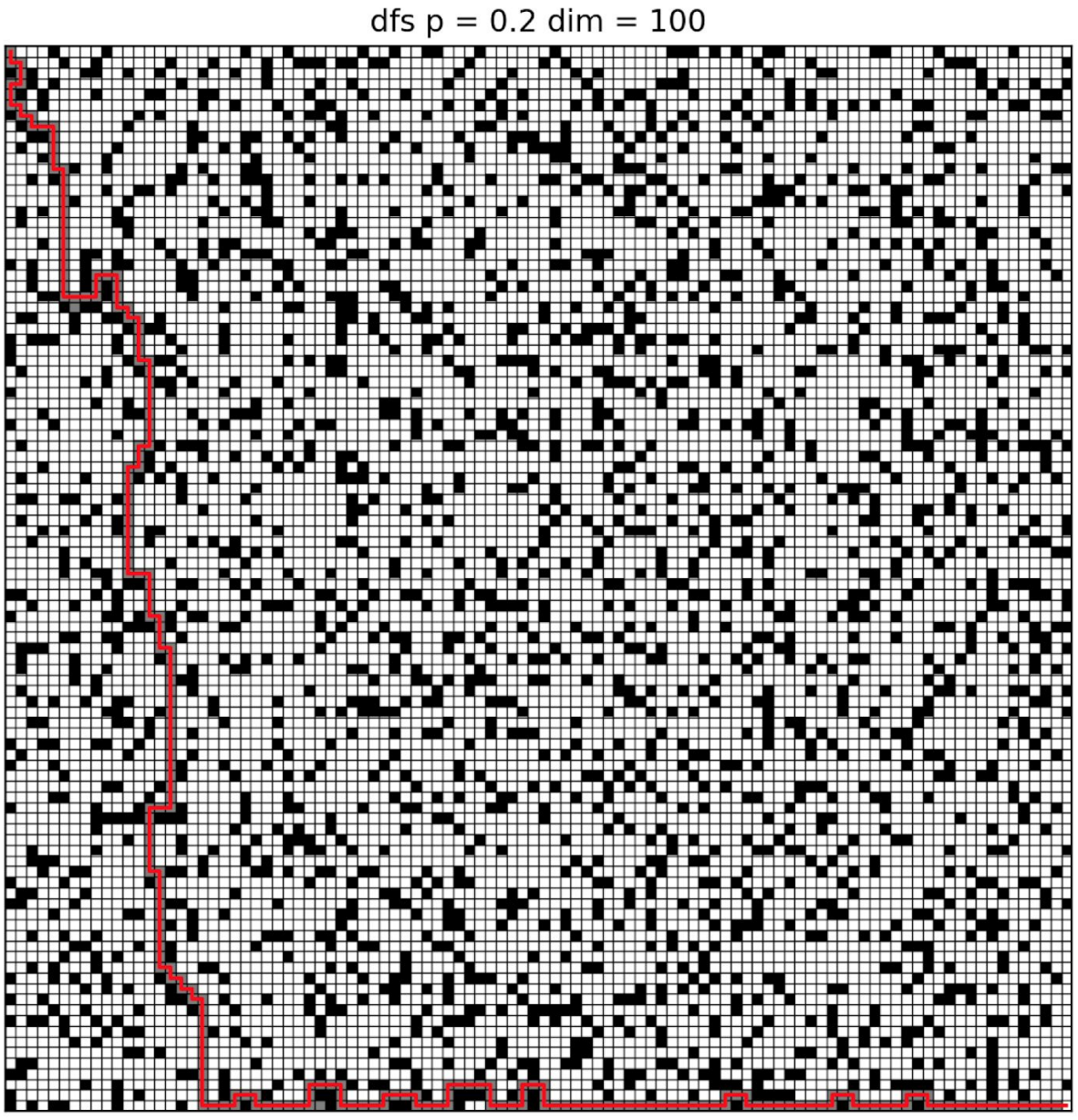

Fig 2.1 path returned by DFS

Breadth First Search:

Max Fringe Size: 102
Path Length: 199
Total Expanded Nodes: 7966

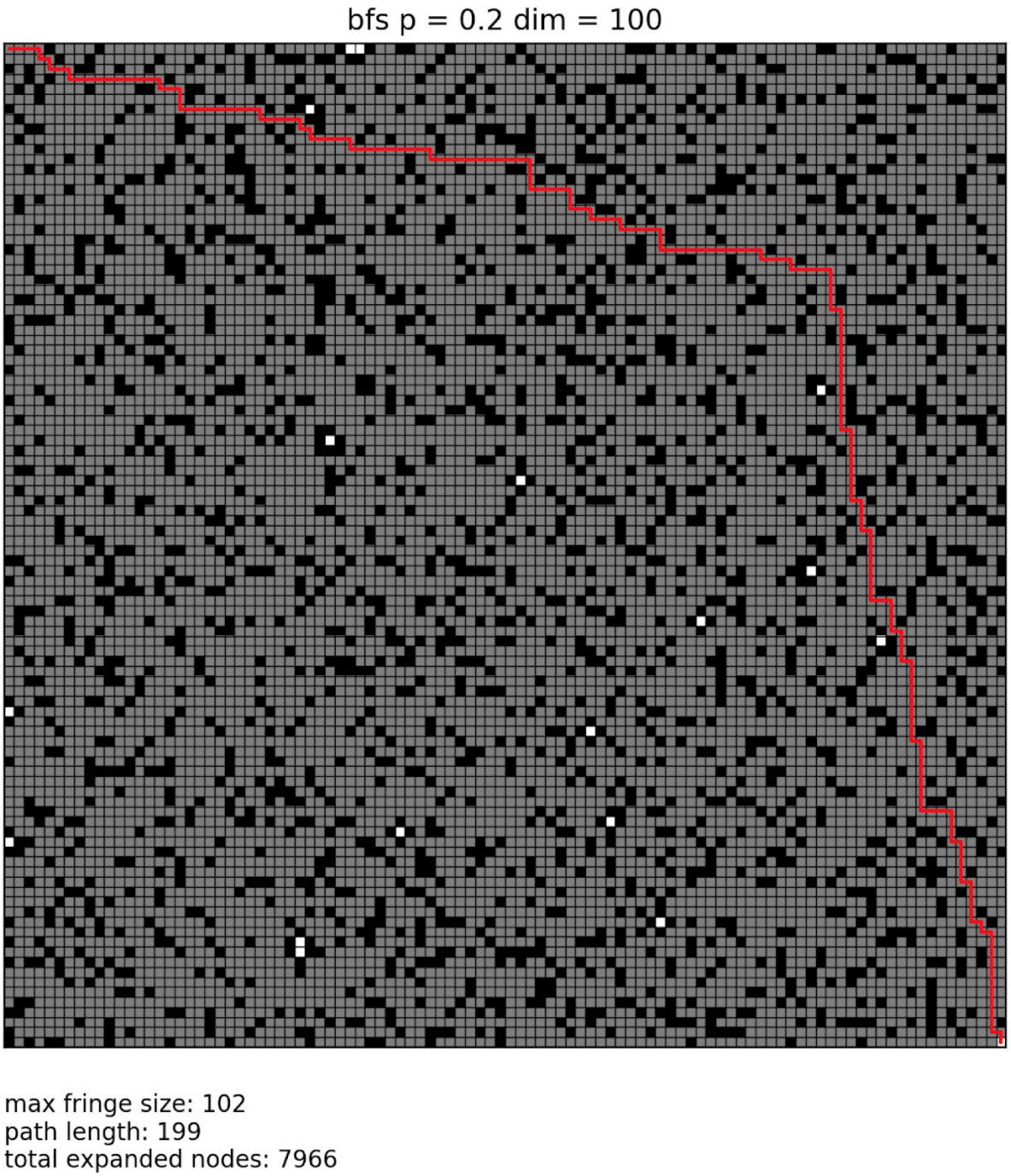

Fig 2.2 path returned by BFS

A* Search (Euclidean Distance)

Max Fringe Size: 161
Path Length: 199
Total Expanded Nodes: 7277

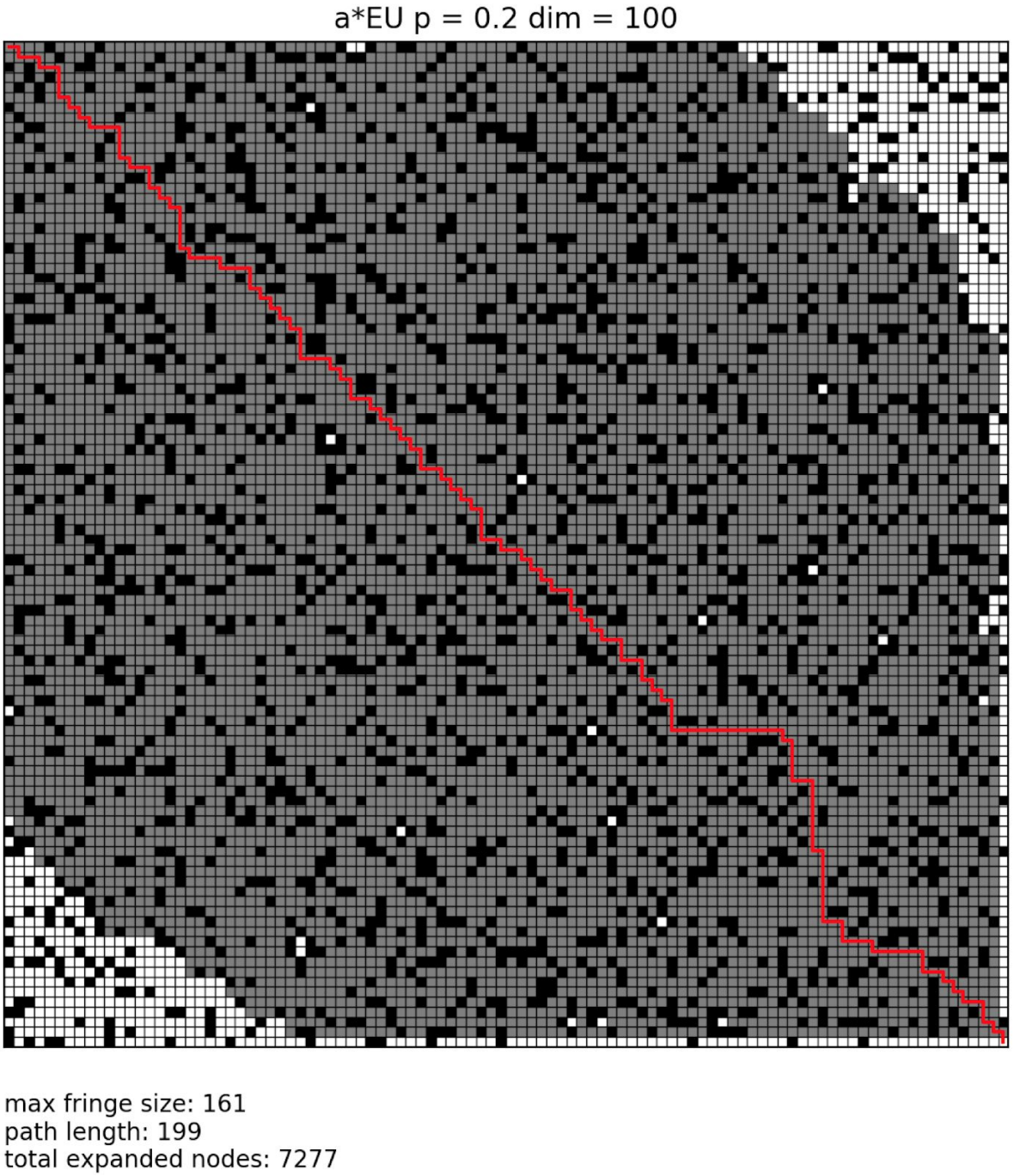

Fig 2.3 path returned by A*(Euclidean Distance)

A* Search (Manhattan Distance)

Max Fringe Size: 499
Path Length: 199
Total Expanded Nodes: 976

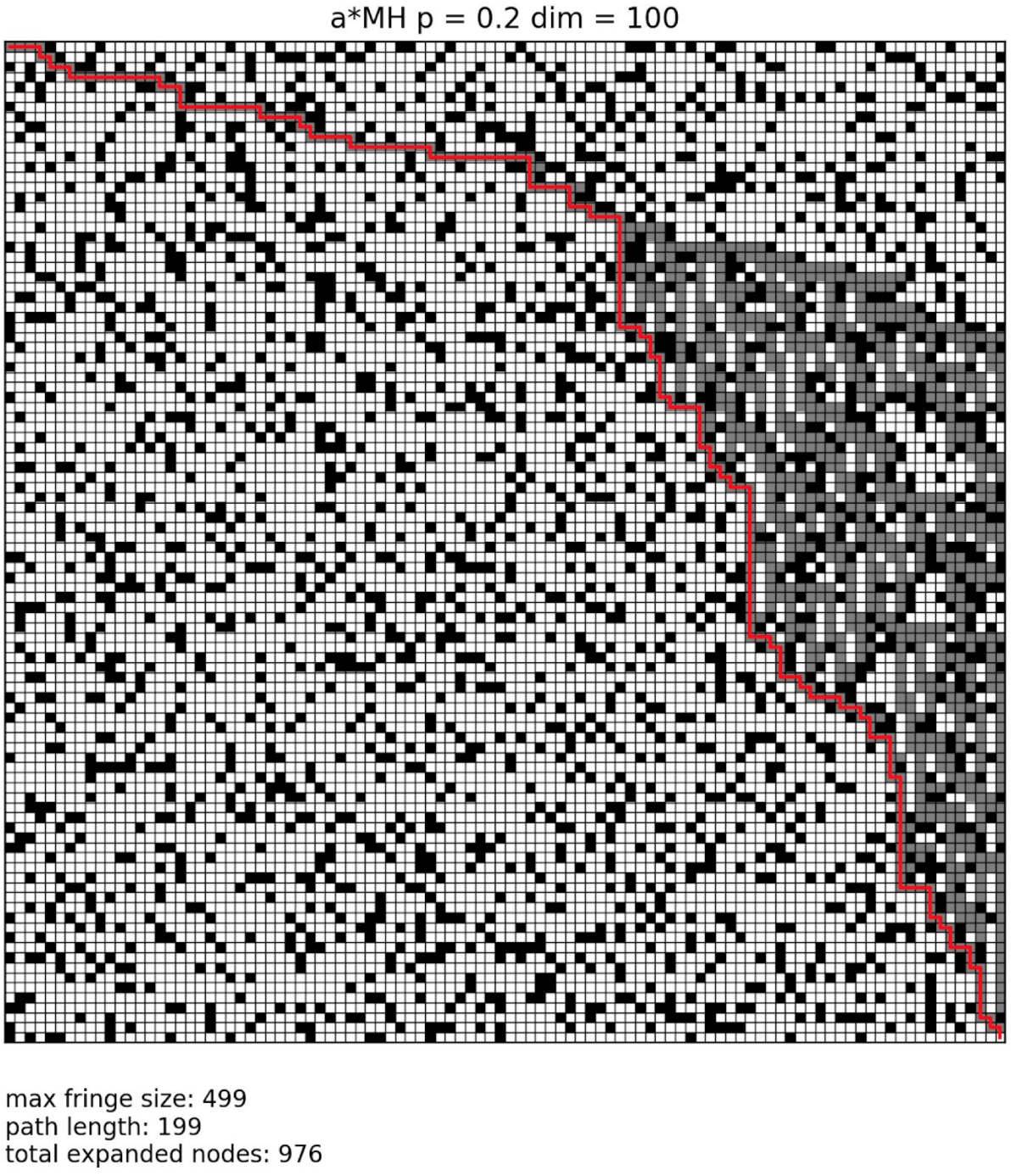

Fig 2.4 path returned by A*(Manhattan Distance)

Bi-Directional Breadth First Search:

Max Fringe Size: Not recorded
Path Length: 199
Total Expanded Nodes: 7286

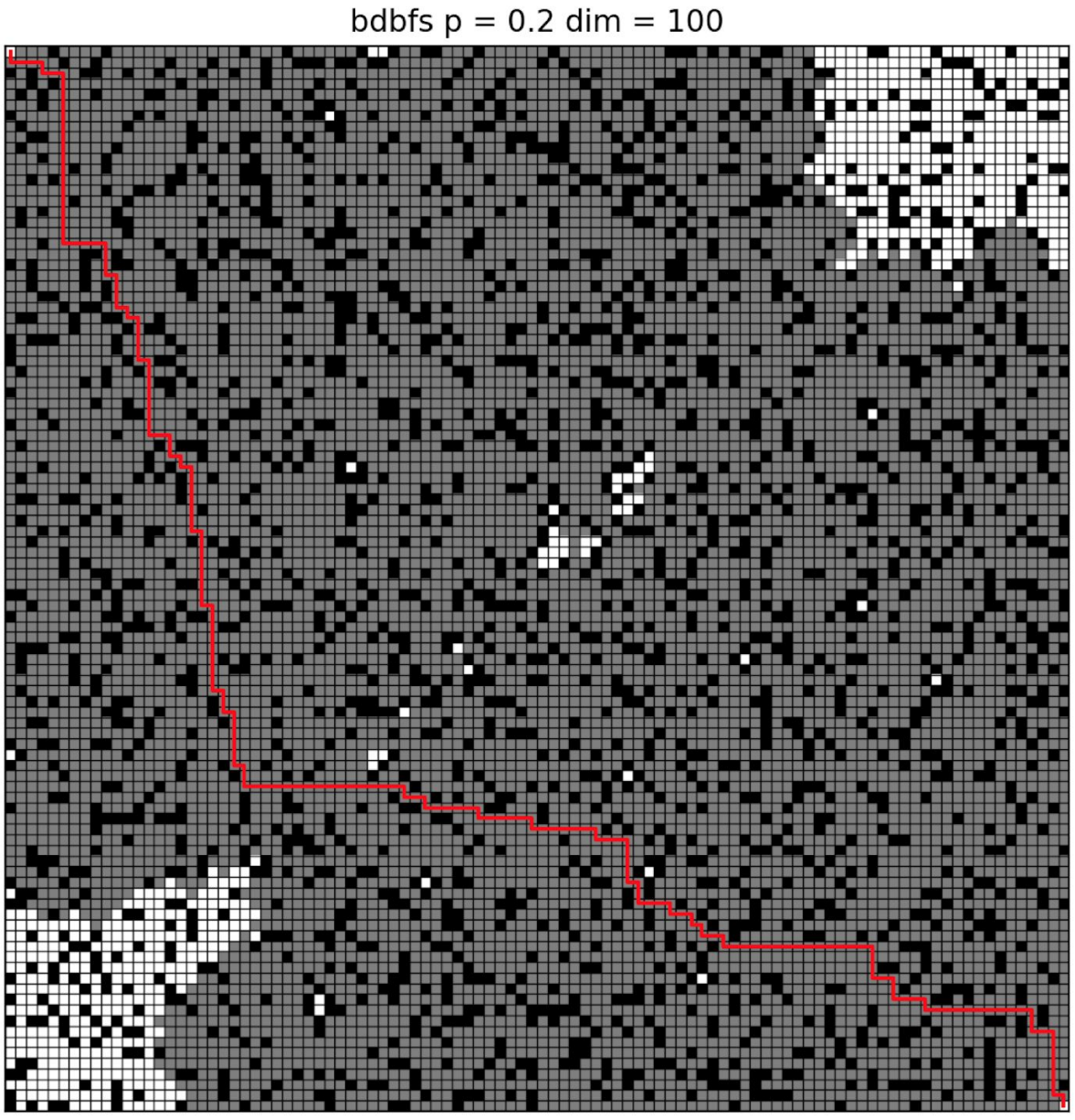

Fig 2.5 path returned by BD-BFS

From the above results, we can see that, except for DFS, other algorithms give the shortest path, but because there is more than one shortest path, the paths obtained by different algorithms are not the same. At the same time, we can observe that DFS expands fewer points than BFS, which is in line with expectations. Moreover, A*MH also expands less points than A*EU, because the heuristic function of A*MH is closer to the actual cost than A*EU under the conditions of this problem, so it is more efficient. In summary, we conclude that our algorithms behave as they should.

2.3 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 p0 where for p < p0, most mazes are solvable, but p > p0, most mazes are not solvable.

For given dim=100, we plot the maze solvable rate versus p-value based on 1000 experiments. From the figure as follows, we can see that when p is close to 0.4, the solvability is almost zero. Therefore, we can say that the value of p0 is 0.4. When p is less than 0.4, many mazes have solutions, and when p is greater than 0.4, most of the mazes have no solution. In order to get results faster, we chose to use the DFS algorithm when calculating the solvable rate, because in most cases DFS can find the path in the maze fastest without traversing most of the points in the maze compared to other search solutions.

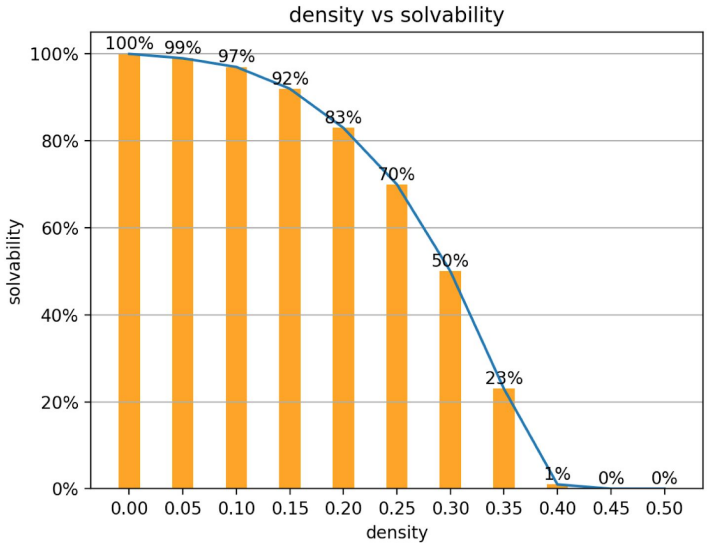

Fig 2.6 density vs. solvability

Bonus: In addition, we also draw a three-dimensional image of p, dim versus solvability. From the image we can still see that regardless of the value of dim, the survival rate is close to 0 when p is equal to 0.4, so we know that p0 value is not affected by dim.

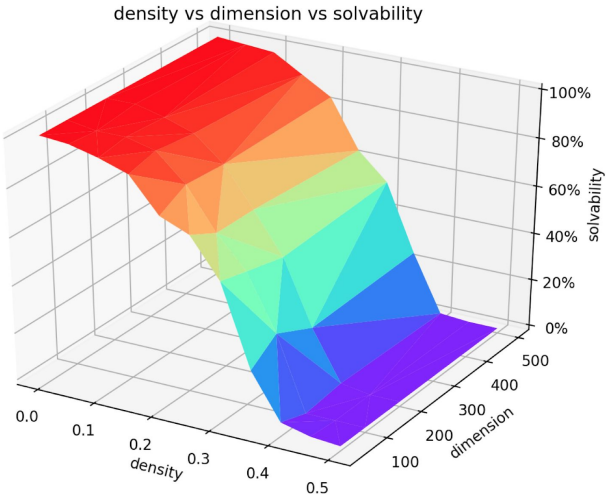

Fig 2.7 density vs. solvability

2.4 For p in [0,p0] 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?

For the four algorithms BFS, BDBFS, A*MH, and A*EU, they all get the shortest path, but because A*MH expands less nodes, we can get the results faster. So we use A*MH to calculate the average shortest path. When dim=100 and p is less than p0, we draw the curve of density vs path length as follows

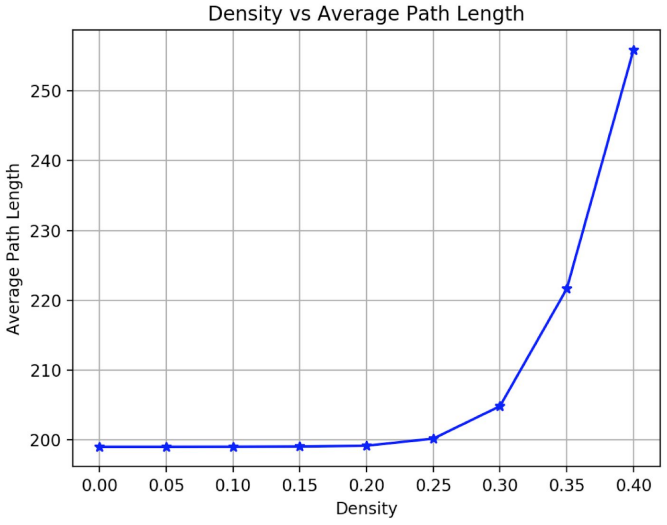

Fig 2.8 density vs. average path length

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

For a path search algorithm, we can measure it in three ways: path length, max fringe size, total expanded nodes. For A*MH and A*EU, because their heuristic functions are both less than the actual cost, then they could get the shortest path in the graph, so the path generated by them have the same length. However, since the heuristic function estimate of MH is larger than the estimate of EU and is closer to the actual cost, A*MH will expand fewer points and find the path faster. For the last property----max fringe size, as we said in the first part, A*EU extends more points than A*MH, reducing the storage of many useless nodes in fringe, so A*EU tends to have a smaller max fringe size, but as we can see from the data below, the gap between the two algorithms on this property is often small, usually in the same order of magnitude. Instead, the difference between the two algorithms in total expanded nodes is very obvious, and gradually increases as the dimension of the maze increases. In summary, we can say that A*MH has better performance than the A*EU in the maze runner.

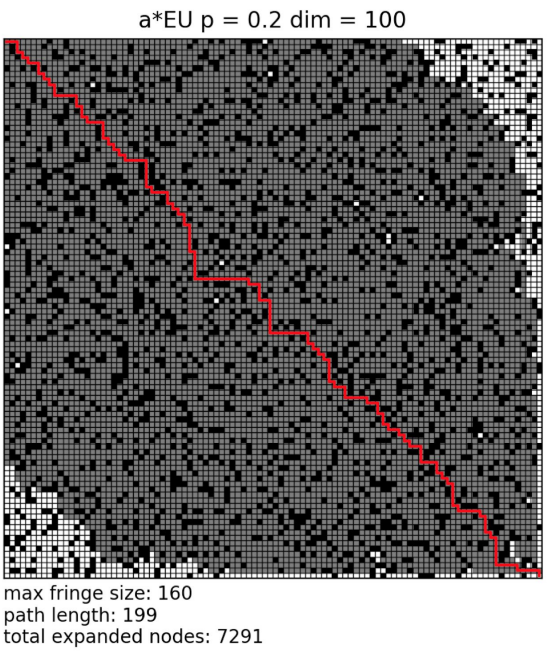

Fig 2.9 Result of a*EU

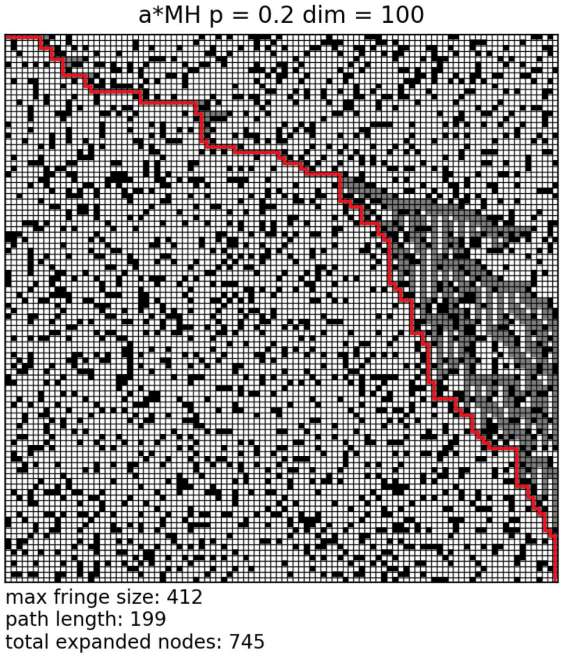

Fig 2.10 Result of a*MH

2.6 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.

For DFS, we indeed can improve the performance by changing the loading order. In this question, because the position of the end point will always be in the lower right corner of the maze, no matter where we are now, if we first load the neighbors to the right or down, there is always a greater chance of reaching the end. So when calling DFS, the left or lower neighbors are more worth looking at first than the upper or right neighbors. For the same maze, the results before and after the improvement are as follows:

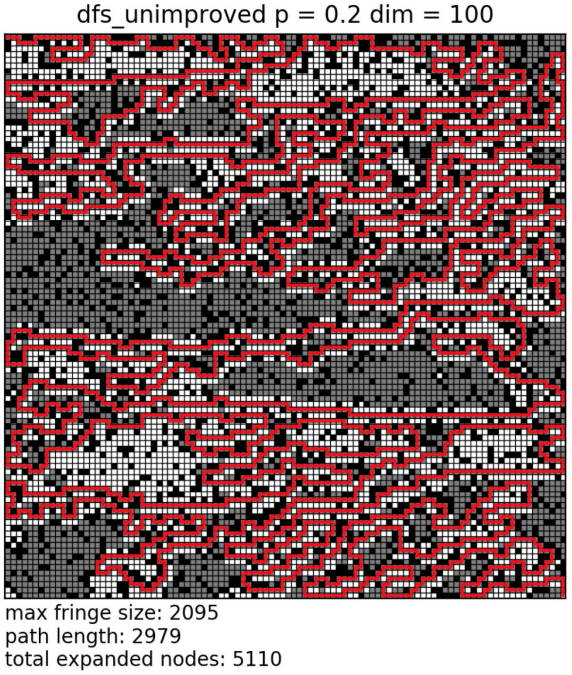

Fig 2.11 Before Improvement

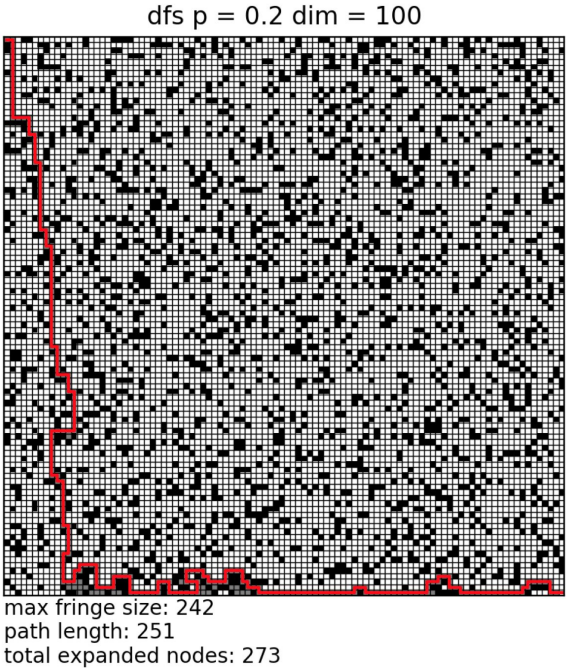

Fig 2.12 After Improvement

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

Yes, on the same maze, there will be many nodes that BDBFS expands but A* does not expand, because BDBFS is still BFS in nature, it will expand all nodes in each layer of the tree. On the other hand, when heuristic function is close enough to the actual cost, A* can arrive at the end point with very few nodes expanded. Let's take A*MH as an example (because it is more efficient than A*EU) to show the difference between A* and BDBFS on the same map.

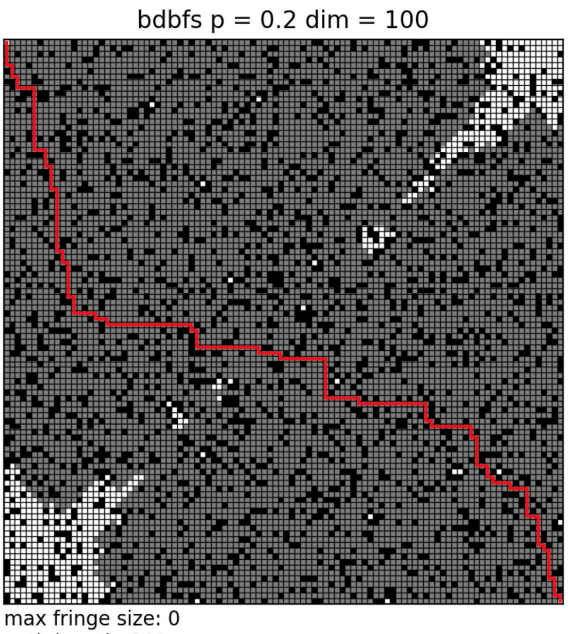

Fig 2.13 node expanded for BD-BFS

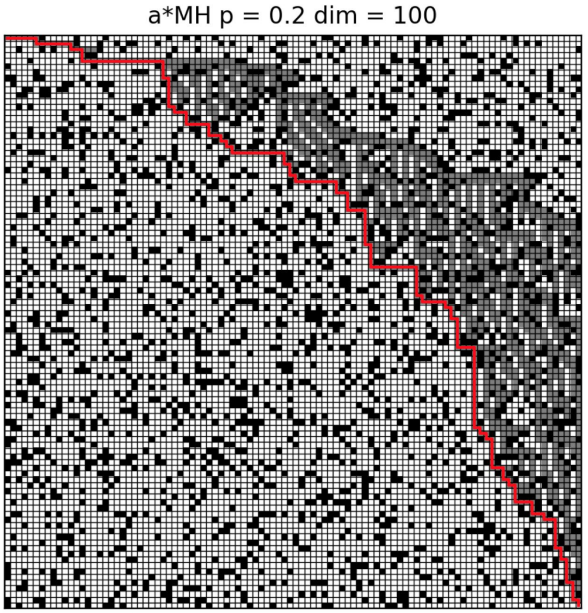

Fig 2.14 node expanded for A*(MH)

3 Generating Hard Maze

In order to gain a better view we implement three local search algorithms including: simulated annealing, beam search and genetic algorithm.

3.1 local search algorithms

In the part, we used three local search algorithms to calculate the hard maze and compared the advantages and disadvantages between them.

For the Simulated Annealing and Beam search algorithm, we use a similar state change pattern. At each time when we change the state, with a certain probability we randomly add or remove an obstacle in the maze. Considering adding operation increases the difficulty of a maze, we do not fairly add or remove but increase the acceptance rate of remove operation as the number of obstructions increases. Besides completely random walk, we add obstructions specifically on the path of the former maze with a probability to give the model some heuristic information. The difference is that in the simulated annealing algorithm, we only change one maze at a time.

In beam search, we generate m random mazes at a time and make k changes for each maze. Then all the generated sub-mazes are scored, leaving only m better sub-mazes for the next iteration. And when some of the parent maze can't find a better child, we record it as a local optimum. The algorithm terminates when all the parent mazes no longer produce a sub-maze.

The genetic algorithm is delighted by the concept of “survival of the fittest” brought by *Theory of Evolution*. Unlike simulated annealing and beam search, genetic algorithm does not focus on a specific solution, it generates a set of solutions and believe that the best solution can be generated from them. The flowchart of genetic algorithm is shown in the following, which is quite a simple one.

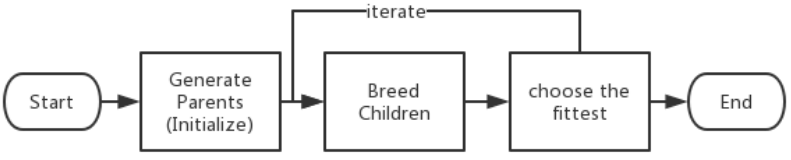

Fig 3.1 genetic algorithm

Firstly we generate a set of solution randomly as parents. In order to increase the complexity of parents, we use different *p* (from 0.3 to 0.4) each time we run the generate function.

Secondly, we pick up two solutions randomly and generate a new solution. The strategy we use is to chop the solution two by two into four parts and combine them into one. After generating a certain number of children, we keep the fittest by calculate the complexity of all solutions and delete the simple ones.

After that we simply iterate the second process until we believe that we get the best solution.

3.2 Answer to the Questions

In addition, the evaluation criteria for maze difficulty can be measured in many ways. We chose three different aspects----Maximal Fringe Size based on DFS, Maximal Nodes Expanded based on A*MH and Maximal Path Length based on BFS.

When dim=50, the experimental results are as follows:
Simulated Annealing:

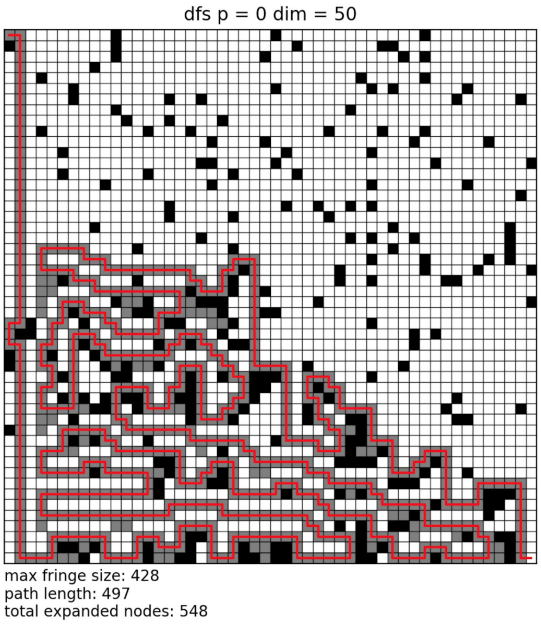

Fig 3.2 Maximal Fringe Size

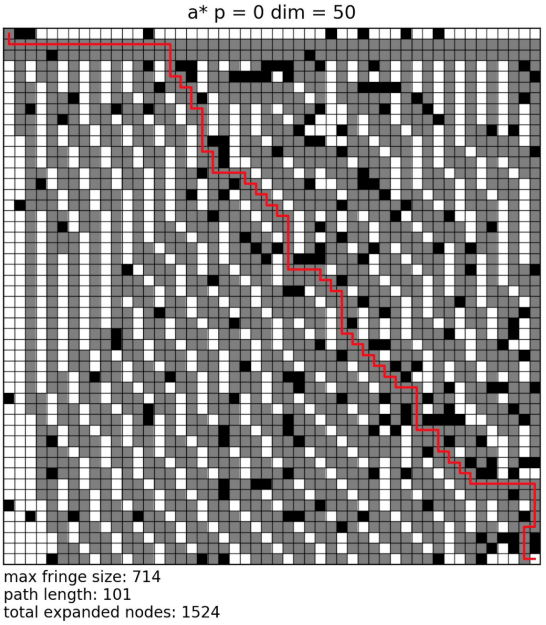

Fig 3.3 Maximal Nodes Expanded

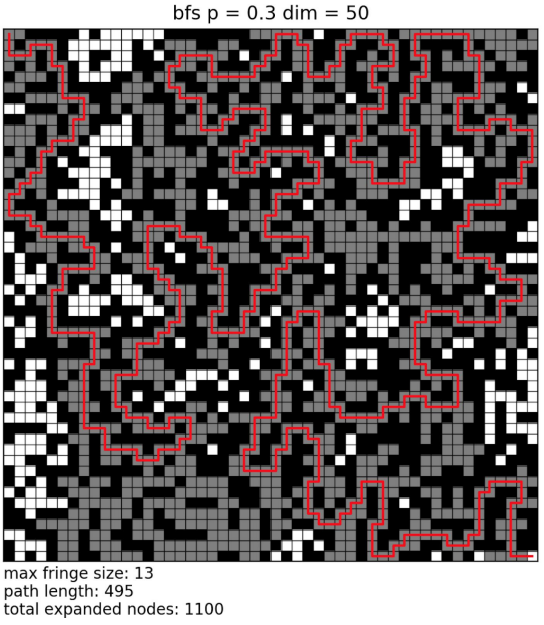

Fig 3.4 Maximal Path Length

Beam Search:

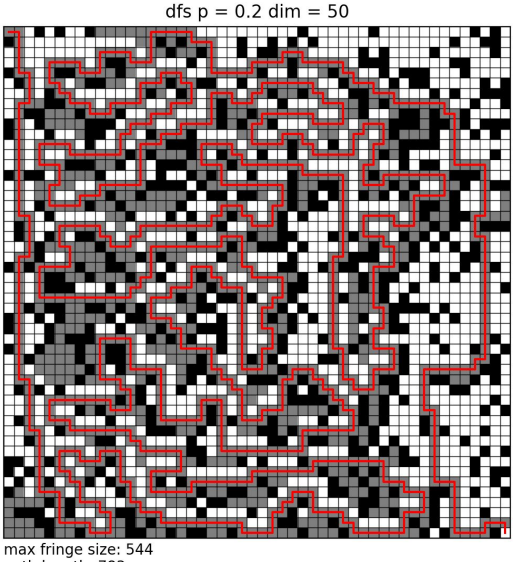

Fig 3.5 Maximal Fringe Size

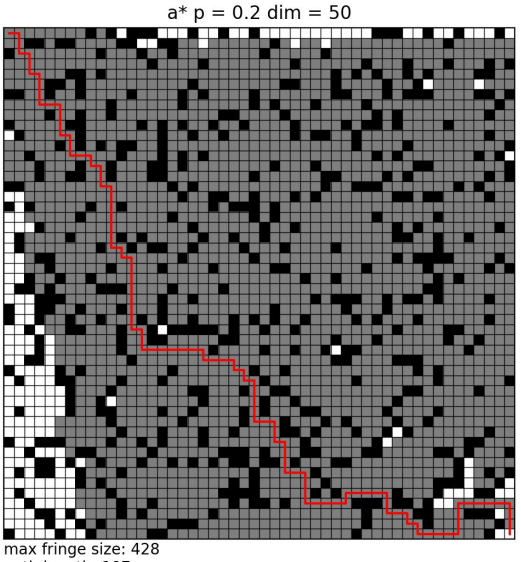

Fig 3.6 Maximal Nodes Expanded

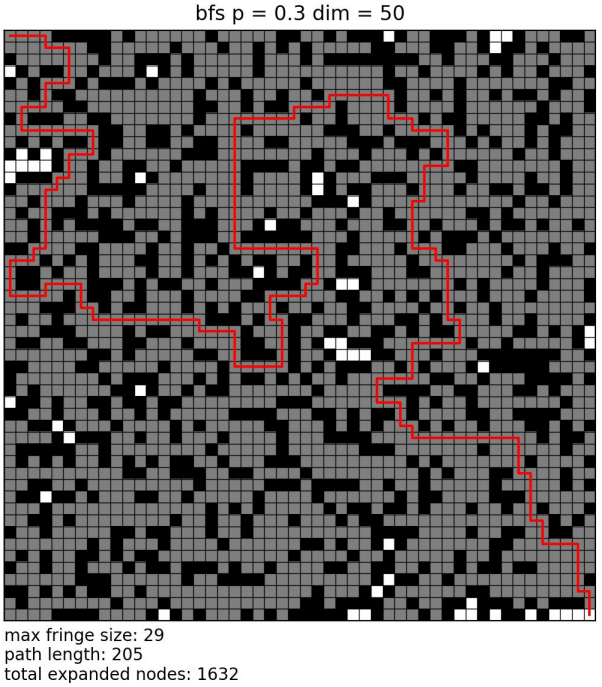

Fig 3.7 Maximal Path Length

Genetic Search:

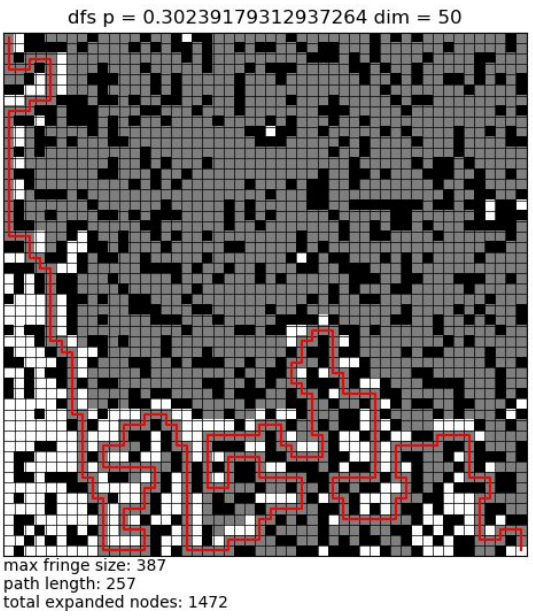

Fig 3.8 Maximal Fringe Size

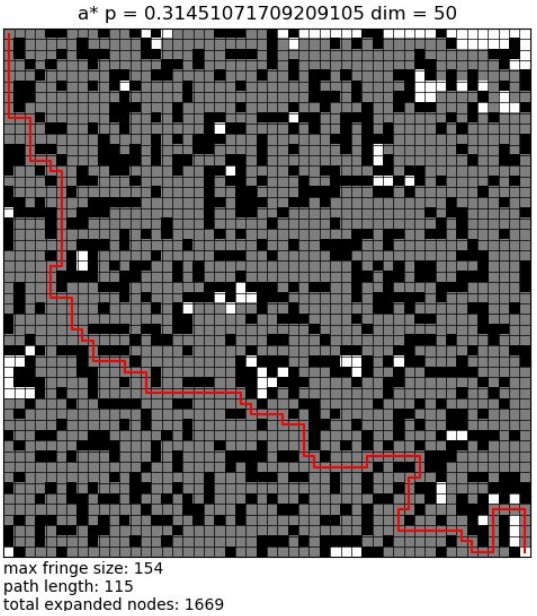

Fig 3.9 Maximal Nodes Expanded

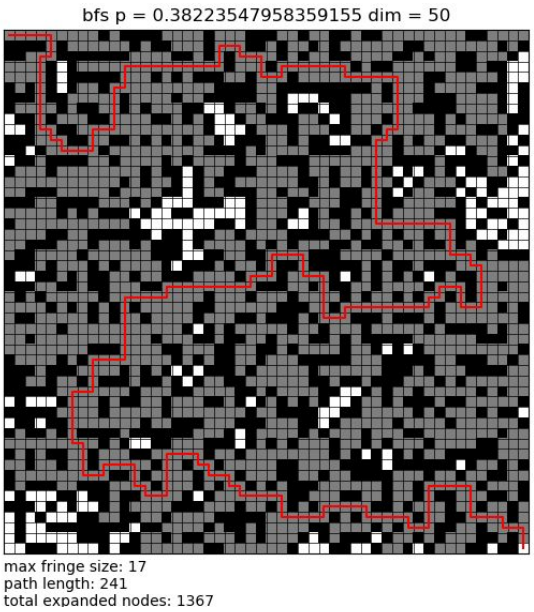

Fig 3.10 Maximal Path Length

Compared with other algorithms, Simulated Annealing has the fastest running speed and can achieve better results under three different fitness functions. Compared with SA, Beam Search is much slower, and it works very badly when using the shortest path of BFS as a fitness function (this is because, for a large number of maze, BFS has the same path length which makes Beam Search hard to move forward). In addition, BS and SA often fall into a local optimal situation, but because Beam Search implements multiple searches at the same time, the probability of discarding the local optimal result is greater than that of SA, so we get the difference results between Figure 3.2 and Figure 3.5. For Genetic Algorithms, it is the most efficient algorithm to avoid local optimization, but his problem is that the iteration speed is too slow, which makes us unable to get satisfactory results in a short time.

Regardless of the impact of the applied local search algorithm, there are some striking differences between the experimental results and intuition of DFS and A*MH algorithms. Because of the nature of the DFS algorithm, even if the number of obstacles in the maze is small, it may form a difficult situation for DFS algorithm(DFS often “moves forward” in the direction away from the end point). And the A*MH algorithm may expand a lot of nodes due to a small detour in the path(This is because many nodes in the maze have the same weight. A detour will seriously affect the search path of the algorithm). However, for the BFS algorithm, we have a satisfactory result since it’s committed to finding a shortest path.

Although we didn't make it “the hardest” because of the experimental time, we believe that if we increase the number of iterations of the algorithm, our algorithm will get better results.

4 What if the maze were on fire

4.1 Algorithms

In the section we implemented MazeState extended from Maze class to describe the state of our maze at every step. The algorithms we used are listed as follows and experiments, analysis and comparisons are made in the latter part.

4.1.1 Baselines

As baseline, we simply generate a maze with its shortest path using the algorithms we implemented in the second part. And then we do the simulation and run the shortest path regardless of the fire. After repeating the experiments for many times we can compute the average success rate if we choose a certain algorithm.

4.1.2 Weighted A* Algorithm

To take consideration of both survival rate and end node information, we choose to search for a path at each step when the state of maze updates, from the robot’s current position to the end node, using a* method. The heuristic function is a combination(simple weighted arithmetic mean) of survival rate and the manhattan distance/euclidean distance. After a path is determined, the robot takes the next step from start node, or say, current node to update maze state.

4.1.3 Timeless Methods(Simple Attempts)

However, this kind of function does too much redundant computation, since we only need to take one step each time, and the whole path is soon useless due to the fire state change. Although too simple and immature, not as good as a result, we made some attempts based on this consideration. We do not search for a precise and valid path to the end node, but just determine the direction by the weight mentioned above without validity determination.

It didn't perform well when we directly determine the next step by weighted arithmetic mean of our two values. Lack of knowledge in mathematical optimization, we utilize the survival rate and end node information separately. The selection priority is: 1. unvisited nodes(better than visited nodes); 2. nodes with better survival rate; 3. nodes with smaller manhattan distance which means you are reaching the goal. As we expected, this function leads to smaller success rate than a* method since it ignores the validity of its path. (We did some work to see if something unexpected will happen after swapping 2 and 3 but not mentioned in the report.)

4.1.4 Ant Colony Optimization

We are also curious about whether optimization algorithms can be used to solve this problem. So we implement ant colony optimization(ACO) algorithm and check its performance compared to the searching algorithms. The flowchart of ACO is as follows.

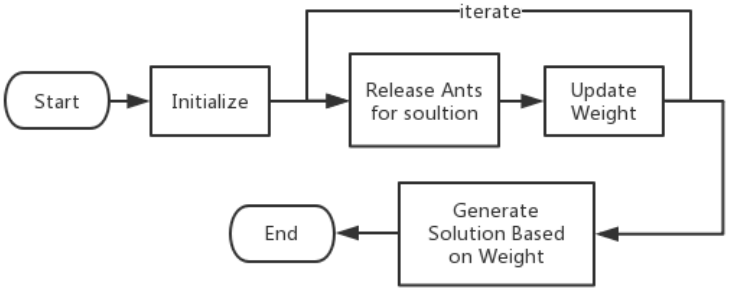

Fig 3.5 Ant Colony Optimization

When we release an ant into the maze, it starts walking randomly through the maze, assume that an ant is at point i, it takes the next step by examining the probability of take steps to each neighbors. For example, the probability of it stepping to point j for the next step is calculated by the formula(3.1).

$$P(i,j) = \frac{[\iota(i,j)]^{\alpha}[\mu(i,j)]^{\beta}}{\sum_{s}\{[\iota(i,s)]^{\alpha}[\mu(i,s)]^{\beta}\}} \tag{3.1}$$

$\iota(i,j)$ stands for the weight we get from the previous iteration. $\mu(i,j)$ is the heuristic value we set to help the ant for route finding. In this case, we set it as the manhattan distance between point j and the goal point.

If an ant manage to reach the goal point before the fire kills it, the weight of all the node in its path is updated using the formula(3.2).

$$\iota(i,j) = \iota(i,j) + \frac{1}{PathLen} \tag{3.2}$$

That is to say the weight is added by the reciprocal of the length of the path.

After an iteration, the weight of all points are updated by multiplied to a constant minus or equals to one.

$$\iota(i,j) = \rho \cdot \iota(i,j) \tag{3.3}$$

By doing this we enable the algorithm to converge faster by “forgetting” the past information gradually.

In conclusion, Ant Colony Optimization algorithm is similar to random walk algorithm, but we manage to include the information of a flaming maze by updating the weight with the information bought by “surviving ants”.

4.2 Experiments

4.2.1 Baselines, Weighted A* Algorithm and Timeless Methods

We ran the algorithms above respectively and recorded the success rate as well as time assumption to assess their performance. In the experiment, we generate mazes of dimension 30, 50, 100 adapting to our operating environment(I know the dimension is a little bit small but we currently have no appropriate devices), and obstruction occupation rate 0.2 in order to reduce bias (The occupation rate higher, the success rate more likely to be influenced by obstructions rather than fire since we regenerate the maze under unsolvable conditions.). Then, we set the flammability rate as 0.5 and 0.6 for dimension 30 and 50 to ensure its spread speed. As dimension grows, the speed needs to be cut down to follow our robot’s pace, and we choose 0.4 and 0.5 for dimension 100.

The weights of survival rate and manhattan distance(We only used manhattan distance based on the knowledge from part 2 that manhattan distance performed better known and there’s no huge difference between the two metrics) are hard to choose, thus we pre-tested under dimension 50, flammability rate 0.5 to determine the parameters (w1, w2) and the result is shown in diagram 4.1:

Fig 4.1: success rate given (w1, w2) under specific parameters

According to this result we set (w1, w2) in all experiments to (-8, 1), a roughly good value. In addition it confirmed that the fire information indeed improved the performance of a* method, comparing with (0, 1) condition which is also a baseline.

After determination of experiment settings we ran 1000 times under each condition and recorded success rate every 100 times to avoid bias. The success rates of 30-dimension maze over 1000 times are recorded below:

success rate	dfs	bfs	a*	weighted a*	weight	2step
30, 0.5	0.346	0.198	0.263	0.486	0.219	0.221
30, 0.6	0.048	0.042	0.200	0.243	0.061	0.075

Form 4.1 success rates of 30-dimension mazes

Having a first glance from form 4.1 we found that in small scale of maze, baselines performed quite well, especially dfs(This may came from the re-initial method we used but I'm not sure), so we only use dfs in following experiments. Unfortunately timeless but unfounded simple methods in 4.1.3 could not beat all the baselines(fortunately part of them were beaten), and 4.1.3 methods performed better when the flammability rate rises, which means the survival path would be more influenced by fire rather than obstructions. Besides their own limits of determining directions simply, one explanation for that is our limited work on weight computation, and there is some space for further optimization. Encouragingly our weighted a* method gave a good result. More results of 50-dimension and 100-dimension mazes are shown in form 4.2:

success rate	dfs(baseline)	weighted a*	weight	2step
50, 0.5	0.186	0.471	0.160	0.168
50, 0.6	0.008	0.145	0.007	0.008
100, 0.5	0.079	0.303	0.065	0.077
100, 0.4	0.489	0.868	0.337	0.344

Form 4.2 success rates of 50-dimension and 100-dimension mazes

Thus we conclude following facts:

1) As dimension becomes larger, our methods will perform better than the baseline under high flammability rate.
2) 4.1.3 methods didn't perform well out of both arithmetic and limits and lack of mathematical optimization, 2-step idea performed a little bit better since we didn't give an optimal mathematical model in original weight idea. They may show time superiority on time under very large dimensions but now not very obvious since we didn't strictly test. All in all they are just attempts.
3) weighted a* method performed well due to the usage of survival rate when a robot came across a cell. When dimension and flammability rate become larger, the extent that success rate given by a* method better than baseline rises rapidly, given a possibility for this method to apply to larger circumstances.

4.2.2 Ant Colony Optimization

Because of the fact that ACO algorithms and different performance from the other algorithms discussed above, in this part we analyze the experiment results and make conclusions.In the experiments of this section, the parameters is set as follows.

ants number	α	β	ρ
1000	2	5	0.9

Form 4.3 parameters of aco

Firstly, we run ACO algorithm in relatively low-dimension mazes and see how the algorithm converges.The index we observe is the survival rate of ants in each iteration. If the survival rate grows higher in each iteration, we can say that the possibility of the solution surviving the burning maze is becoming higher.

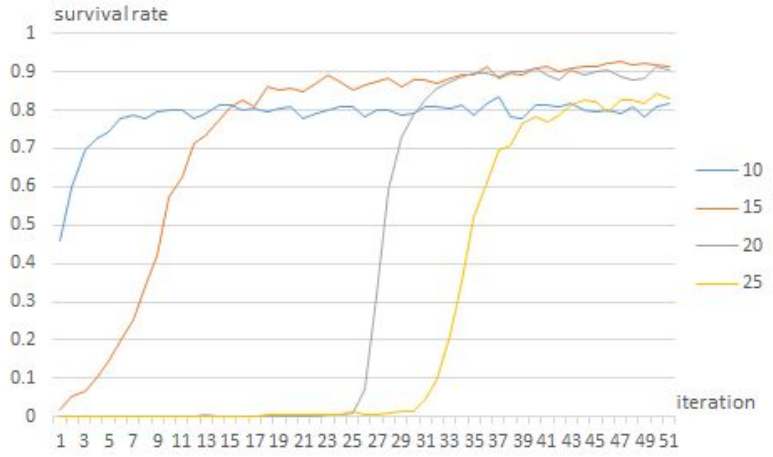

Fig 4.2 convergence of survival rate in different maze size

As shown above, we set the dimension of maze from 10 to 25 and run the algorithm four times. We can list the following conclusions from the chart.

1. The convergence of algorithm becomes slower as the dimension of maze increases.
2. once the convergence starts, it would reach the best solutions in at a relatively fast speed.
3. In this case, the survival rate can reach [0.8, 0.9].

We also try to run the algorithm in a larger maze. As a result, the running time becomes increasingly slower and the 1000 ants in the algorithm fail to find a solution in the first few iterations. So we make the conclusion that although ACO algorithm converges fast and can always find a good solution, its large time complexity makes it almost impossible to deal with high dimensional maze.

5 Contributions

yz956: DFS, BFS in part 1, simulated annealing algorithm in part 3, weighted a* algorithm and some timeless methods in part 4, code skeletons, experiments and analysis in part 4

xg142: a* search in part 1, beam search in part 3, visualization, experiments and analysis in part 2 and 3

jw1308: BDBFS in part 1, genetic algorithm in part 3, baseline and ant colony optimization and in part 4, experiments and analysis in part 3 and 4