CS 520 Assignment 1 – Path Planning and Search Algorithms

Member Name	NETID	RUID	Class Section
Linchen Xie	lx100	188004552	Section 1
Ningyuan Zhang	nz146	186002871	Section 1
Weikang Li	wl494	189006134	Section 2
Shenao yan	sy558	187003007	Section 1

Part 1 Path Planning

Questions:

1. We let p = 0.2. And for each of the implemented algorithms, we find that the dimension of the map should be 1000*1000 which can return an answer in a reasonable amount of time.

2. We consider p = 0.2 and dim = 20.

Visualization results of each algorithm are as follows, black squares represent occupied cells, white squares represent empty cells, pink squares represent visited nodes, and red squares represent path.

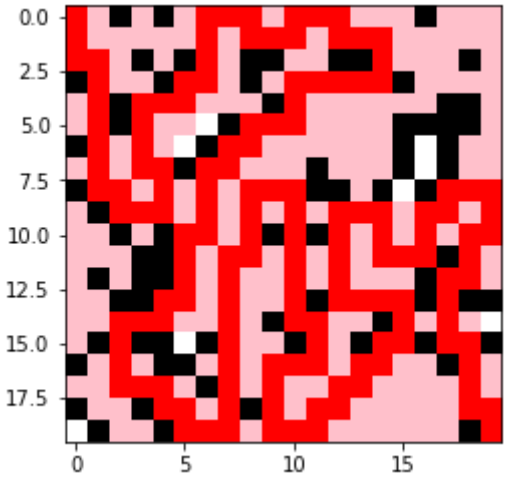

Figure 1: Result using DFS algorithm with p=0.2,dim=20

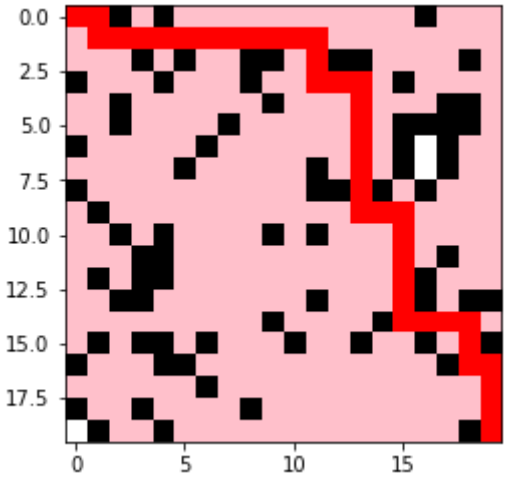

Figure 2: Result using BFS algorithm with p=0.2,dim=20

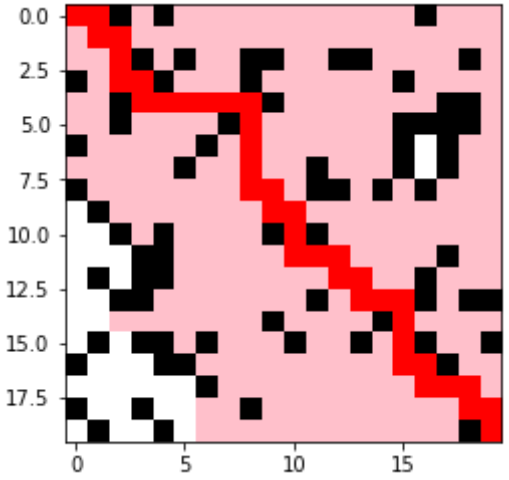

Figure 3: Result using A* Euclidean algorithm with p=0.2,dim=20

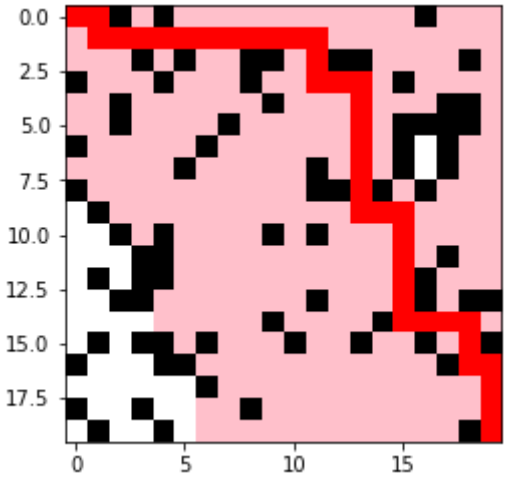

Figure 4: Result using A* Manhattan

algorithm with p=0.2,dim=20

Other parameters of solution using different algorithms are as follows:

Table 1: Parameters of solution with p=0.2,dim=20

	Find path or not	Number of visited nodes	Path length
DFS	YES	316	147
BFS	YES	321	39
A* Eclidean	YES	291	39
A* Manhattan	YES	289	39

We need to notice that priority of different directions while visiting nodes play a significant role in DFS because DFS explores as far as possible along a branch until find a solution. Ih this case, we set left and up direction higher priority to magnify the shortness of DFS.

3. The Depth-First Search method is the most useful here . It is fast and we just want to find whether there is a complete path. So it is not important for us to know the route and length and DFS is more useful than other three methods.

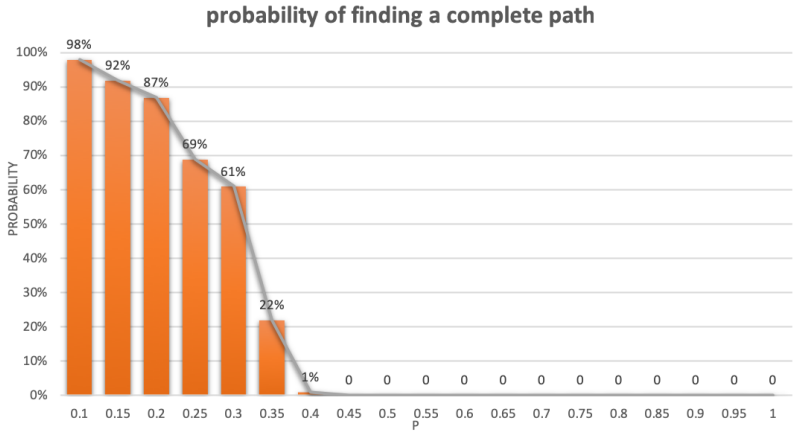

Figure 5: Probabilities of finding a path with different p

We can conclude from figures above that 0.4 is a watershed, which means 0.4 is the p0.

4. We use A* Manhattan algorithm because it has the shortest path and it visits the least numbers of cells.

We set the map size 100*100, and the results are as follows:

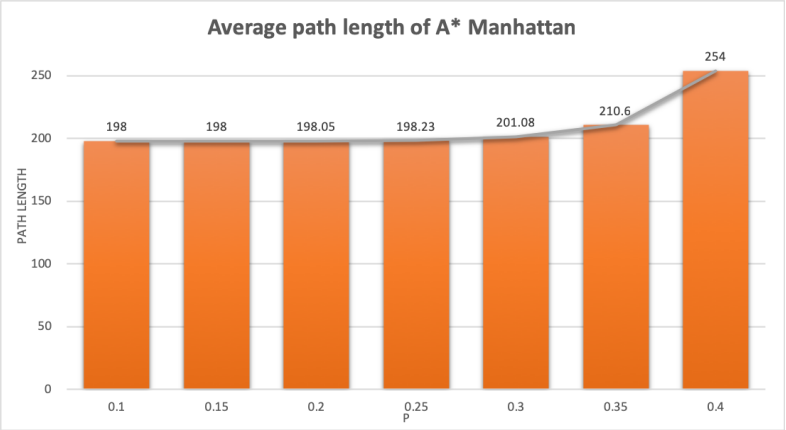

Figure 6: Average path length using A* Manhattan with different p

5. When we use A* Manhattan algorithm, we can get the same results as shown in Question 4 so we won't repeat the result.
When we use A* Euclidean algorithm, the results are as follows.

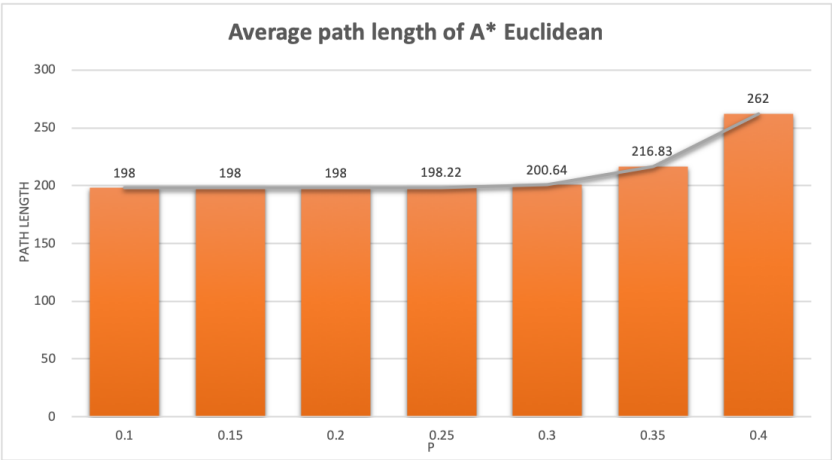

Figure 7: Average path length using A* Euclidean with different p

Similarly, for the same p values, we can get the average length of the path generated by DFS from start to goal.

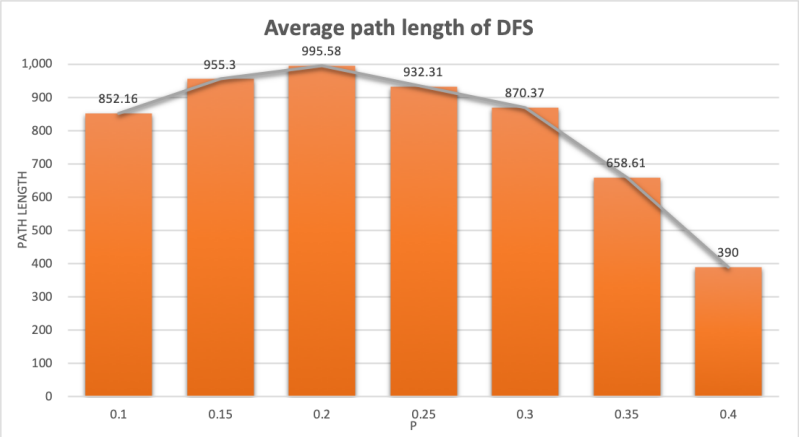

Figure 8: Average path length using DFS with different p

From the results shown above, we can reach a conclusion that average length of the path generated by DFS is longer than the average length of the path generated by A*, no matter what P is.

6. By using A* Euclidean algorithm, we can estimate the average number of nodes expanded in total for a random map as follows (For a range of p values (up to p0)):

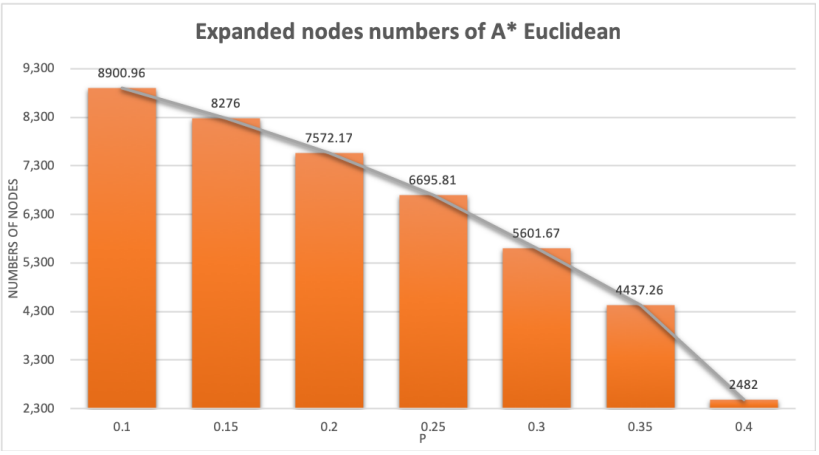

Figure 9: Average nodes numbers using A* Euclidean with different p

Samely, by using A* Manhattan algorithm, we can estimate the average number of nodes expanded in total for a random map as follows:

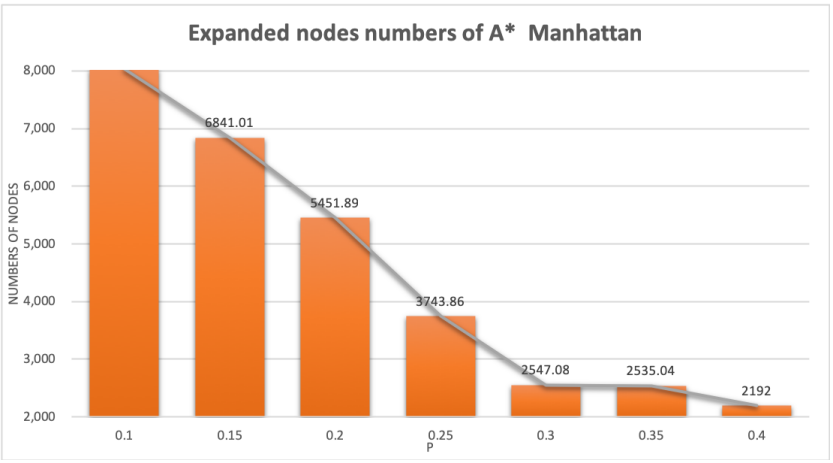

Figure 10: Average nodes numbers using A* Manhattan with different p

From the results, we can see that A* using the Manhattan Distance expands fewer nodes. Because agent in this algorithm can only go up, down, right and left. Comparing Euclidean Distance and Manhattan Distance, we can find that Manhattan Distance is greater than Euclidean Distance with the same (x1,x2), which means Manhattan Distance can increase the difference between nodes and make the search scope more precise.

When p is bigger than p0, we can not get a path from the start of the map to the end. These two heuristics' algorithms both have to expand all nodes. So they give the same value of the expanded nodes.

7. By using DFS algorithm, we can estimate the average number of nodes expanded in total for a random map as follows (For a range of p values (up to p0)):

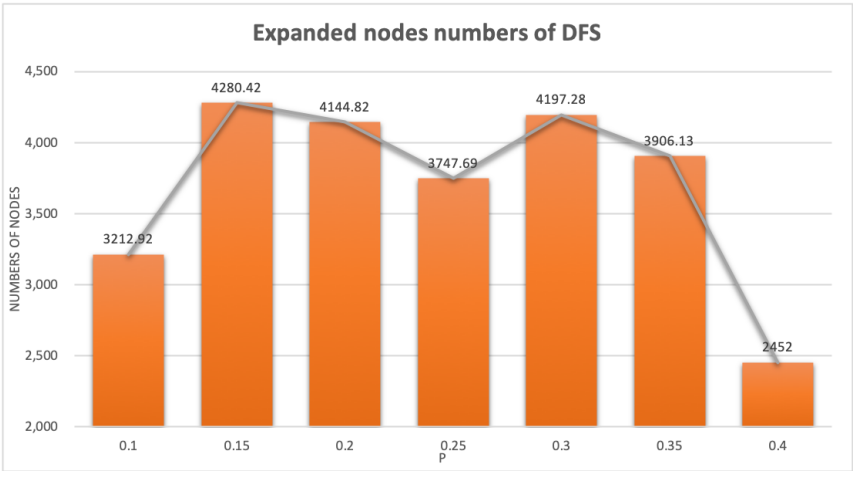

Figure 11: Average nodes numbers using DFS with different p

Samely, by using BFS algorithm, we can estimate the average number of nodes expanded in total for a random map as follows:

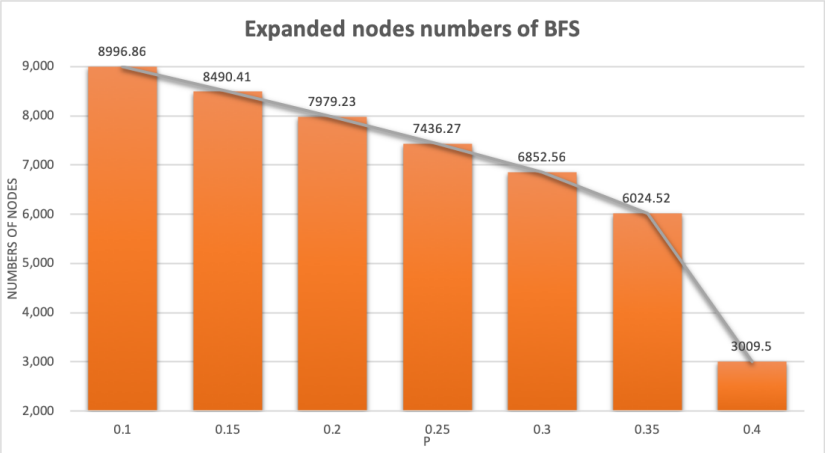

Figure 12: Average nodes numbers using BFS with different p

We can see that DFS algorithm has fewer nodes than BFS algorithm because BFS algorithm will visit all possible nodes to find a shortest path.
Unlike A* algorithm and BFS algorithm, whose expanded nodes numbers are increasing with the increase of p, expanded nodes of numbers DFS algorithm don't change that much with the increase of p. It's because BFS just expands all the way along a branch and return the first solution it reached.But in general, the order of these algorithms regarding expanded nodes numbers is DFS < A* Manhattan < A* Euclidean < BFS.

Bonus
UCS calculates the path cost of all the nodes in the fringe and expands the one that has the minimum cost. But in part one, the path cost of all the nodes in the fringe is same and every cell in the maze has fixed length. So Uniform Cost Search is not a suitable method to solve this problem.

Part 2 Building Hard Mazes

Questions:
8.We consider two kinds of local search algorithms to implement hardest maze. The first one is genetic algorithm. We still generate size*size matrix to simulate a maze, in which 0 means empty cells and 1 means occupied cells. We aim to find a path from node(0,0) to node(size-1,size-1), and during the process we also store some important parameters such as the number of nodes visited and max fringe size to compare the performance of each algorithm. In order to find the hardest maze, firstly we generate a number of random maze with fixed dimension and probability, and sort them based on certain property(path length, number of nodes visited, and maximum fringe size). Then we choose the best 20 mazes to reproduce new mazes. Each new maze is a crossover of its parents: we pick up a random integer N between 0 and size-1, and the first N rows of the new maze come from parent1 maze while the rest of the rows come from parent2 maze, then we can also

reorganize columns with similar method . The whole cross breeding process is like shuffling cards. After the cross breeding process, there's a certain probability that the mutation happens in the new maze. If there exists a mutation, a random cell in the maze will turn 1. Then the reproducing is over and we need to calculate the parameter of this new maze, if it does work better than its parents, which means it's harder, we will store it and let it reproduce in the next iteration to pass down better genes to next generation, otherwise we will discard it to eliminate bad genes. We repeat the reproducing process for a number of times, and then we can get the hardest maze.
The second local search algorithm we used for this problem is Simulated Annealing. The reason why we pick this algorithm is obvious. In some cases, a small change on maze can make it harder to solve. For example, if you put a block in current path that from start to end, then algorithms must find another path, in which case, the maze may be harder than former one in terms of path length, number of nodes expanded or maximum size of fringe. When implementing Simulated Annealing to this problem, we first pick a cell in the maze randomly, if this cell and neighbors are empty, we fill them, otherwise, we remove blocks and make it empty with a certain probability. Then, we find path in this new maze, if new maze is harder, we accept it. If it's easier, we abandon it or accept it with a certain probability. Eventually, we may get the 'hardest' maze.

9. When we implement Simulated Annealing, we set a temperature to this algorithm, and this temperature decreases at every iteration with a cooling schedule. If the temperature falls below a threshold, the algorithm is complete and we assume that this is the 'hardest' maze we ever see.
A shortcoming of Simulated Annealing is obvious but hard to solve: we don't know the where to stop. Just like find global minimum problem, we don't know the minimum value we get now is a local minimum or the global minimum.
However, this algorithm has its advantage. First, it's space complexity is O(1), which means it can be done in place. It is important when we have extremely large maze. Second, this algorithms is probabilistically complete if temperature decreases slowly enough.
10.
Genetic Algorithm:
Due to limited time, we set the number of reproducing process as 2000. The result may not be the definitely hardest one, but we can still see the change and trend.
All results are as follows:
a) DFS
The path length ends with 281, starting with 63. The increase is huge and the map also show vividly that the path is much more complicated than any other maze we've seen before. It makes sense to assume that the hardest maze will have many narrow corners which force the path make as many detours as possible. On our result map,

there’s still some blank area which haven’t been expanded, meaning the distance between our result and the hardest maze. I believe that if we have enough time to experiment, we can get a much harder maze.
The number of nodes expanded ends with 711, starting with 250. The increase is also huge. We can also see that almost all the empty cells turn pink. Because DFS is so stupid since it just expand all the way along a branch before backtracking or finding a solution. And in this case, we set right and down a higher priority. In the result maze, if we turn right and down all the way, we will hit a dead end and the algorithm has to keep backtracking and expanding other branches, which lead to a large number of visited nodes. I think that our result maze is very close to the hardest one because it has expanded all nodes.
The maximum fringe size ends with 241, starting with 74. The reason is similar with the condition using the number of nodes visited as a parameter. Before finding a solution or backtracking, DFS need to store all nodes of before levels.

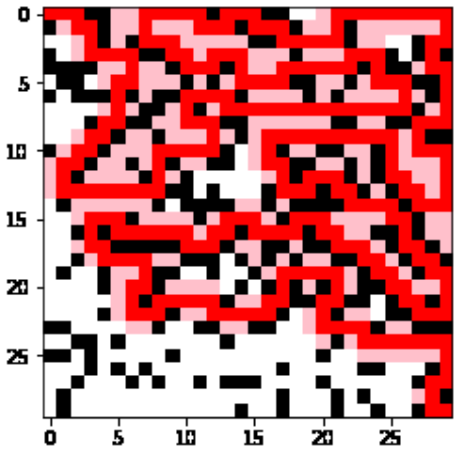

Figure 13: Result using DFS with p=0.2, dim=30,property=path length

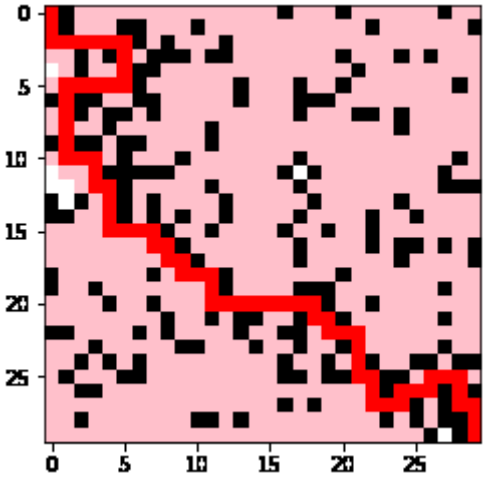

Figure 14: Result using DFS with p=0.2, dim=30, property=number of nodes expanded

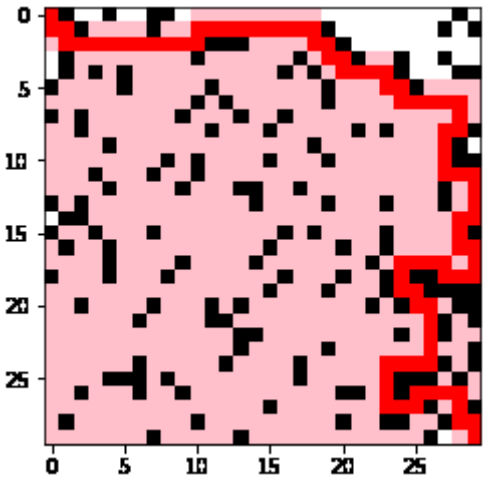

Figure 15: Result using DFS with p=0.2, dim=30, property=maximum size of fringe

b) BFS

The path length ends with 93, starting with 59. The increase is much smaller than DFS. It's because BFS always returns a shortest path and it's harder for us to produce a harder maze to it. We need to repeat producing progress much more times to create a hard enough maze for BFS.

The number of nodes expanded ends with 795, starting with 743. The increase is very small because BFS need to explore more nodes than any other algorithm to find a shortest path. And the whole maze is pink so I consider it a hardest maze.

The maximum fringe size keeps increasing until it reached 59. I also think that it's a hardest maze since the property won't change any more.

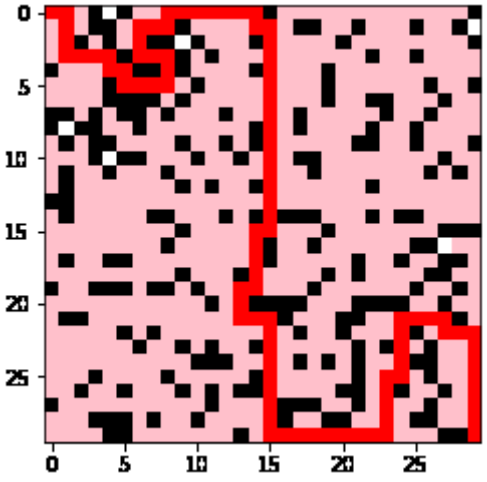

Figure 16 Result using BFS with p=0.2, dim=30, property=path length

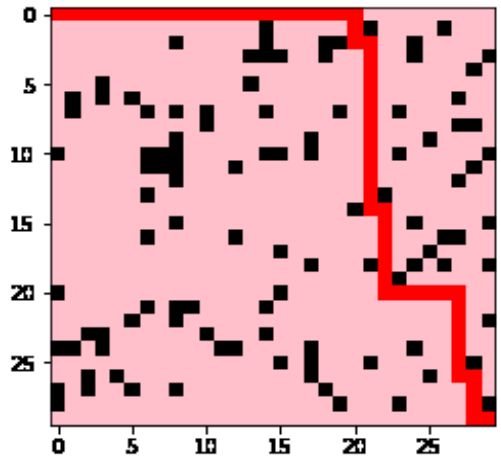

Figure 17 Result using BFS with p=0.2, dim=30, property=number of nodes expanded

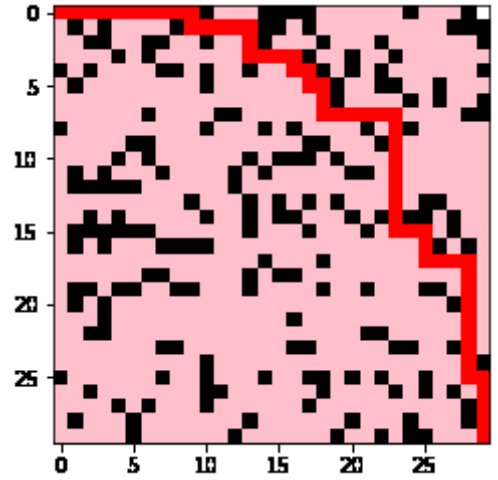

Figure 18 Result using BFS with p=0.2, dim=30, property=maximum fringe size

c) A* with Euclidean Distance Heuristic

The path length ends with 157 and starts with 59. The increase is smaller than DFS but larger than BFS.

The number of nodes expanded ends with 796, starting with 729. We can see that almost all the cells in the maze turn pink so I consider it a hardest maze.

The maximum fringe size increases from 63 to 117. But as you can see, the graph with the property of path length shows that there are some areas that we do not explore. It takes lots of time to run A* with Euclidean Distance Heuristic algorithm and I think we can get better result to find the hardest maze if we spend more time running the program.

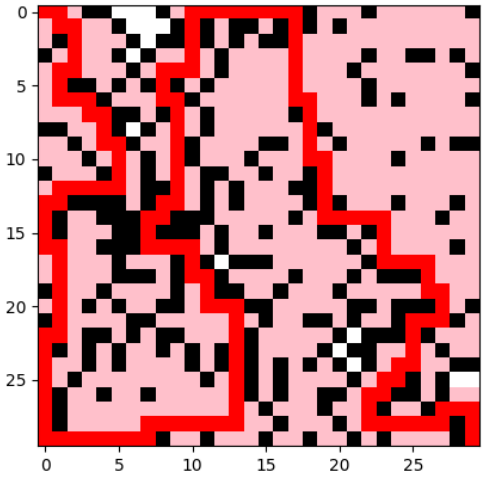

Figure 19 Result using A*Euclidean with p=0.2, dim=30,property=path length

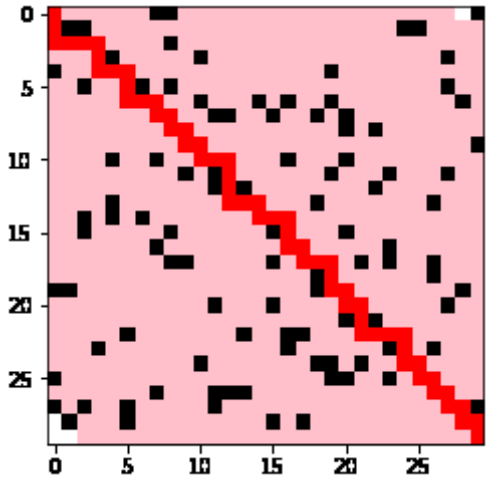

Figure 20 Result using A*Euclidean with p=0.2, dim=30,property=number of nodes expended

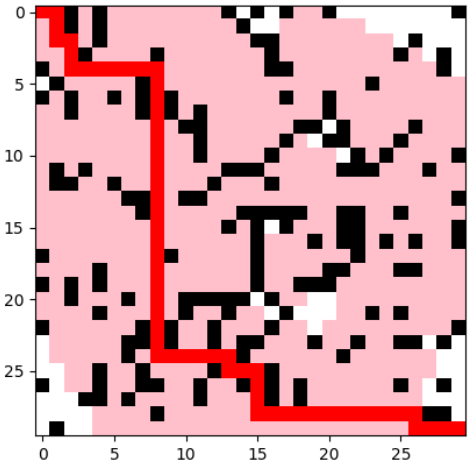

Figure 21 Result using A*Euclidean with p=0.2, dim=30,property=maximum size of fringe

d) A* with Manhattan Distance Heuristic
The path length ends with 91 and starts with 59.
The number of nodes expanded ends with 733, starting with 670. Here we notice that A* with Manhattan still visit more nodes than A* with Euclidean, which accord with what we argue before: manhattan distance is more precise. And the whole maze is almost pink so I consider it a hardest maze.
The maximum fringe size increases from 82 to 162.

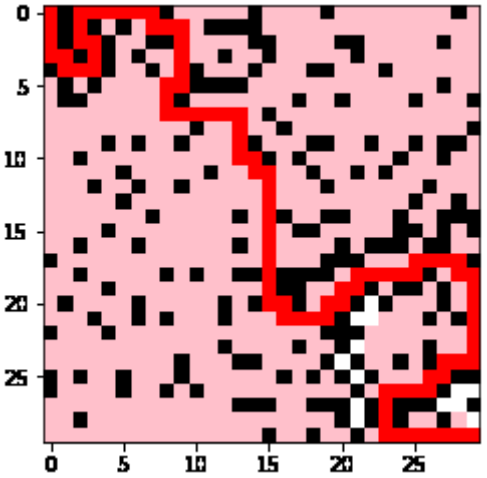

Figure 23 Result using A*Manhattan with p=0.2, dim=30, property=path length

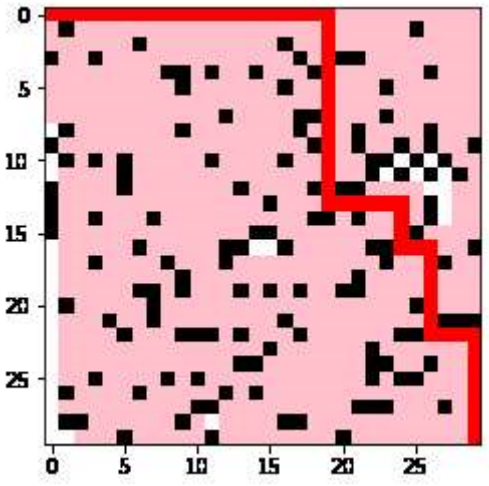

Figure 24 Result using A*Manhattan with p=0.2, dim=30, property=number of nodes expanded

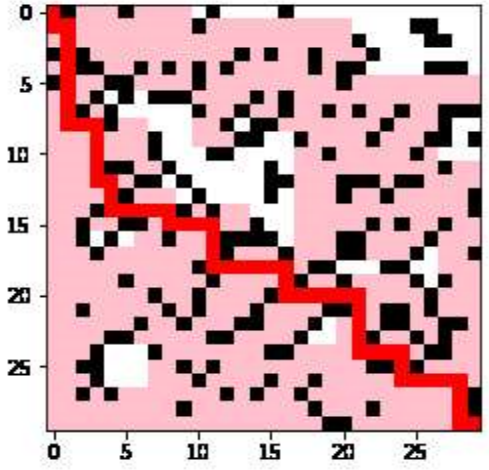

Figure 25 Result using A*Manhattan with p=0.2, dim=30, property=maximum fringe size

Simulated Annealing:

a) DFS

The path length starts with 59 and ends with 335.

With property of Total number of nodes expanded, the algorithm starts with 122 and ends with 680.

The maximum size of fringe starts from 67, ends with 297

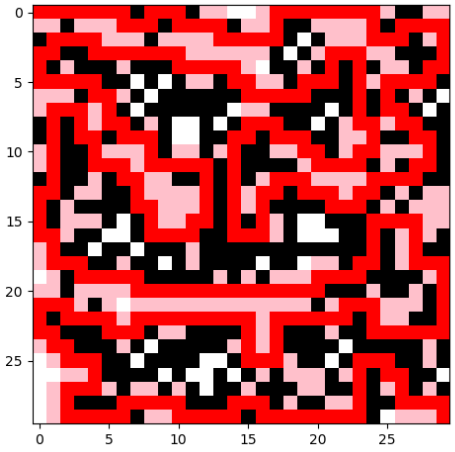

Figure 26 Result using DFS with p=0.2, dim=30, property=path length

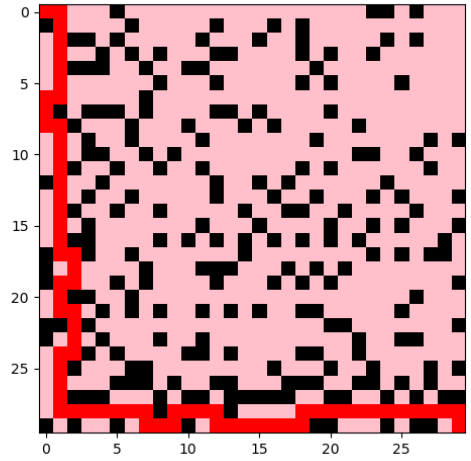

Figure 27 Result using DFS with p=0.2, dim=30, property=number of nodes expanded

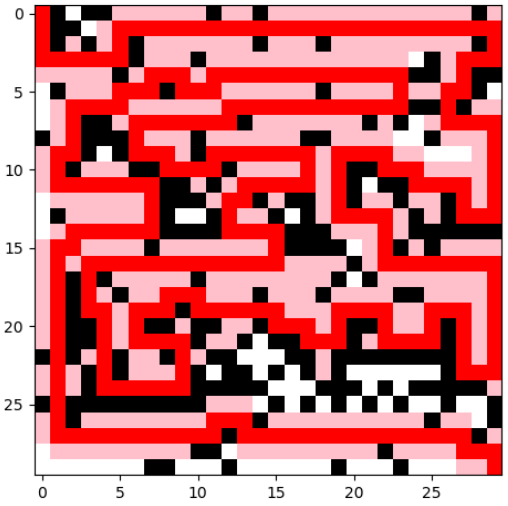

Figure 28 Result using DFS with p=0.2, dim=30,property=maximum size of finge

b) BFS

The path length starts with 59 and ends with 255.

With property of Total number of nodes expanded, the algorithm starts with 715 and ends with 747.

The maximum size of fringe starts from 34, ends with 47.

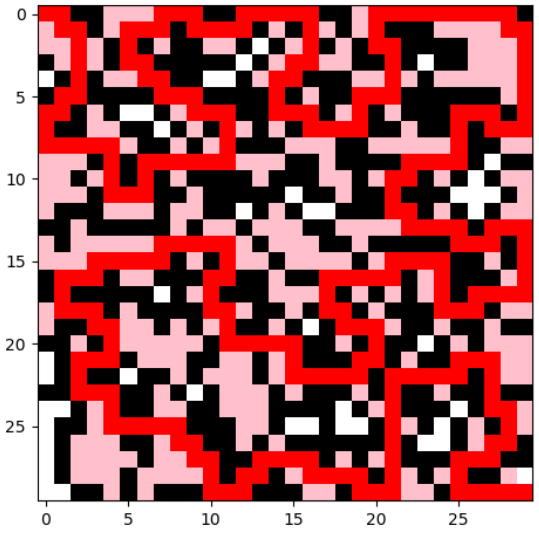

Figure 29 Result using BFS with p=0.2, dim=30, property=path length

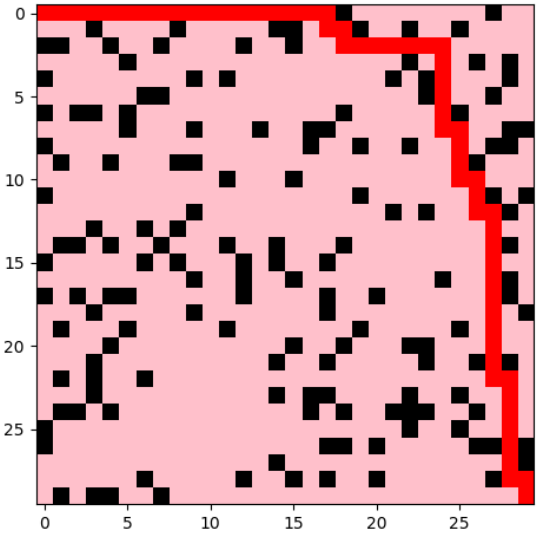

Figure 30 Result using BFS with p=0.2, dim=30, property=number of nodes expanded

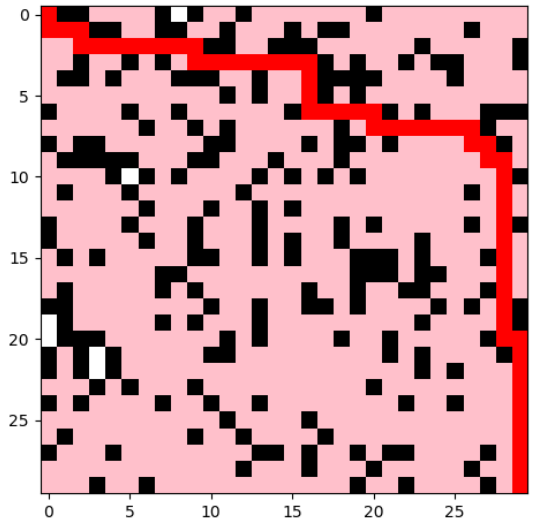

Figure 31 Result using BFS with p=0.2, dim=30, property=maximum size of finge

c) A* with Euclidean

The path length starts with 59 and ends with 157.

With property of Total number of nodes expanded, the algorithm starts with 483 and ends with 653.

The maximum size of fringe starts from 44, ends with 98.

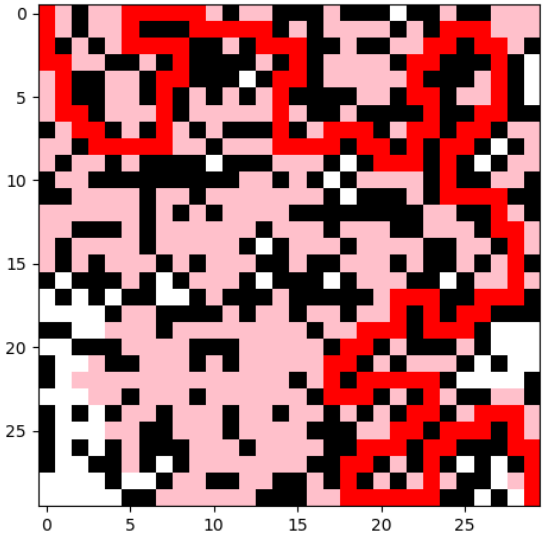

Figure 32 Result using A* with Euclidean with p=0.2, dim=30, property=path length

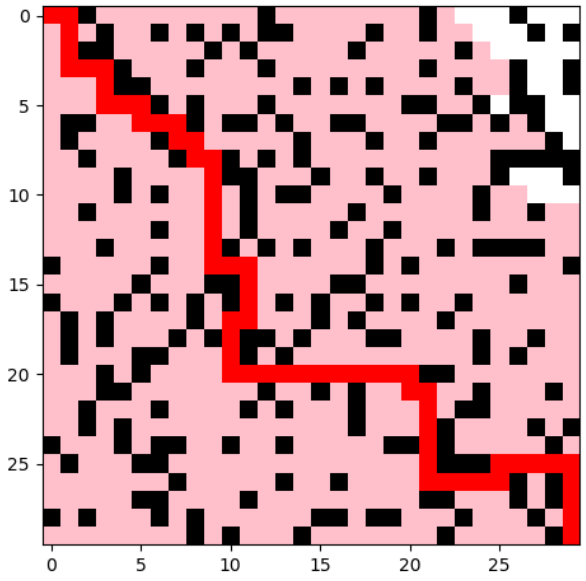

Figure 33 Result using A* with Euclidean with p=0.2, dim=30, property=number of nodes expanded

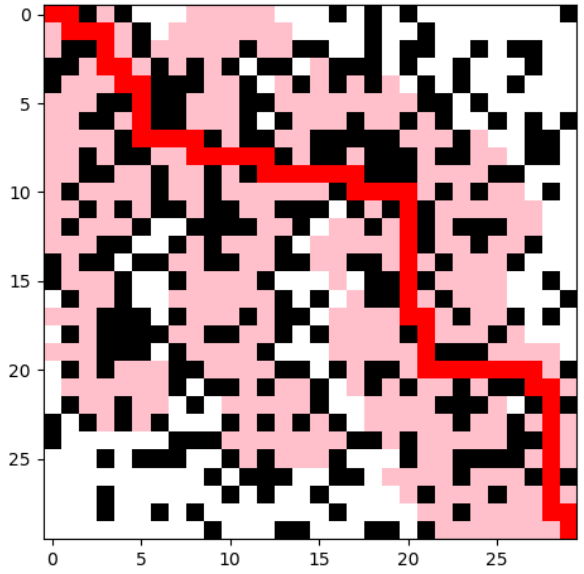

Figure 34 Result using A* with Euclidean with p=0.2, dim=30, property=maximum size of finge

d) A* with Manhattan

The path length starts with 59 and ends with 189.

With property of Total number of nodes expanded, the algorithm starts with 239 and ends with 655.

The maximum size of fringe starts from 40, ends with 108.

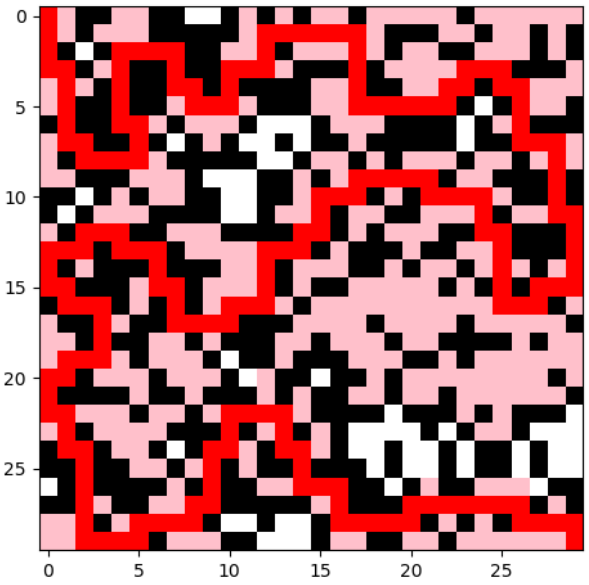

Figure 35 Result using A* with Manhattan with p=0.2, dim=30, property=path length

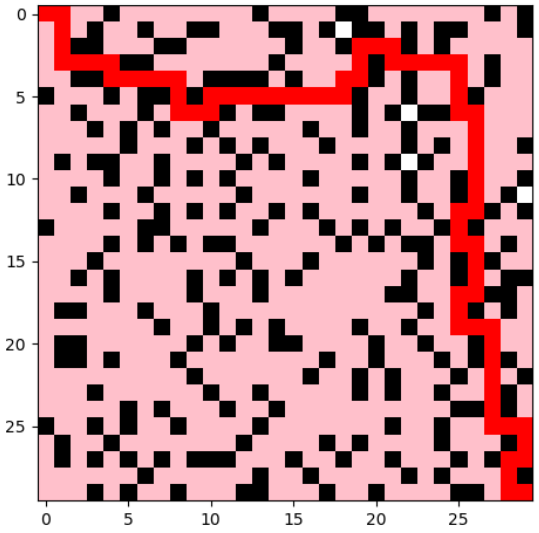

Figure 36 Result using A* with Manhattan with p=0.2, dim=30, property=number of nodes expanded

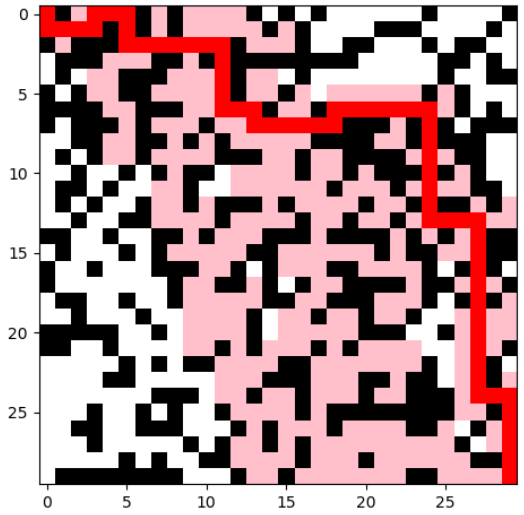

Figure 37 Result using A* with Manhattan with p=0.2, dim=30, property=maximum size of finge

Division of labor:
member1 Linchen Xie: BFS Algorithm,testing code,part1 questions, partmeter tunning for SimulatedAnnealing
member2 Ningyuan Zhang: DFS Algorithm,visualization of part1, Genetic Algorithm,part2 questions
member3 Weikang Li: A* Euclidean Algorithm,visualization of part1,SimulatedAnnealing algorithm, part2 questions
member4 Shenao Yan: A*Manhattan Algorithm, testing code, part1 questions, partmeter tunning for SimulatedAnnealing