CS 520: Assignment 1 - Path Planning and Search Algorithms

Haoyang Zhang, Han Wu, Shengjie Li, Zhichao Xu

October 7, 2018

1 Introduction, group members and division of workload

In this group project, apart from implementing DFS, BFS, A^* with Euclidean Distance and A^* with Manhattan Distance, we also did some modification to these algorithms for different performance out of our personal interests.

Name RUID	Workload
Haoyang Zhang 188008687	Implemented DFS, Iterative Deepening DFS, BFS, Bidirectional BFS, Bidirectional A^*, Simulated-Annealing-Based Beam Search and the visualization of maze. Modified DFS to make it able to return optimal path. Added Last-in First-out feature to A^*. Managed to combine Simulated-Annealing-Based Beam Search with Genetic Algorithm. Ran tests for DFS and BFS in question 10. Finished half of the writing of report for part 2.
Han Wu 189008460	Wrote python scripts for testing the performance of algorithms. Combine the data and generated figures for question 1, 2, 4 and 5. Finished the writing of report for question 1, 2, 4 and 5.
Shengjie Li 188008047	Implemented A^* with Euclidean Distance, A^* with Manhattan Distance and Genetic Algorithm. Ran tests for A^* with Euclidean Distance and Manhattan Distance inquestion 10. Finished the format design of whole report using LaTeX.
Zhichao Xu 188008912	Wrote python scripts for testing the performance of algorithms. Combine the data and generated figures for question 3, 6 and 7. Finished the writing of report for question 3, 6 and 7. Suggested an improvement of A^* using Chebyshev Distance.

2 Part 1: Path Planning

1 For each of the implemented algorithms, how large the maps can be (in terms of dim) for the algorithm to return an answer in a reasonable amount of time (less than a minute) for a range of possible p values? Select a size so that running the algorithms multiple times will not be a huge time commitment, but that the maps are large enough to be interesting.

In order to find a reasonable size of the maze, we tested the running time of the algorithm. For each size, we generated 10 mazes and run different algorithms on these 10 mazes. We recorded the average time each algorithm needs to return an answer. The 10 mazes in each test ccould be either solvable or unsolvable. We set p equals to 0.3 and executed the

experiment.
The results are shown below as Table 1:

		Size					
		100	200	400	800	1600	3200
Time(s)	DFS	0.01942	0.07087	0.28435	0.93919	4.5064	10.83366
	BFS^2	0.07342	0.33946	1.73646	6.40967	25.69253	91.73234
	A^* Euclidean	0.07811	0.41706	1.62604	5.97804	25.27724	100.37974
	A^* Manhattan	0.06093	0.29222	0.88752	3.63068	11.147	63.8882
	BFS^1	254.36093 (size=30)					

Table 1

DFS was the default setting. BFS1 was the default setting. However, it took too much time. When the size was 30, it took more than 250 seconds to return an answer. So, we changed the settings a little to make BFS acceptable. Here came BFS2. In BFS2, checkFringe=True. The others were also False. A* Euclidean means that A* algorithm used Euclidean Distance as the heuristic function. A* Euclidean means that A Star algorithm used Manhattan Distance as the heuristic function.

In the table, we could see that when size becomes large, the average time of returning an answer (whether solvable or unsolvable) increases. BFS2 and A* Euclidean are often the most time-consuming algorithms. We need to run the algorithm repeatedly for the purpose of validations. In order to make it faster in the following test, we chose 200 as the default size of our maze.

2 Find a random map with $p \approx 0.2$ that has a path from corner to corner. Show the paths returned for each algorithm. (Showing maps as ASCII printouts with paths indicated is sufficient; however 20 bonus points are available for coding good visualizations.)

In this question, we drew the path of using different algorithms to solve the maze and we studied the effect of different parameters on the problem solving.
We took 20×20 maze as an example. The maze is shown below:

TODO

3 For a fixed value of dim as determined in Question (1), for each $p = 0.1, 0.2, 0.3, ..., 0.9$, generate a number of maps and try to find a path from start to goal - estimate the probability that a random map has a complete path from start to goal, for each value of p. Plot your data. Note that for p close to 0, the map is nearly empty and the path is clear; for p close to 1, the map is mostly filled and there is no clear path. There is some threshhold value p_0 so that for $p < p_0$, there is usually a clear path, and $p > p_0$ there is no path. Estimate p_0 . Which path finding algorithm is most useful here, and why?

In this question, we tried to use a grid-search method first to narrow down the scope of parameters. We used 200 as the mazeSize, and we generated 800 different mazes for the validation of the parameter value.

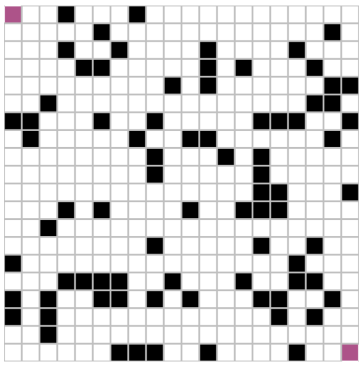
Figure 1: A really Awesome Image

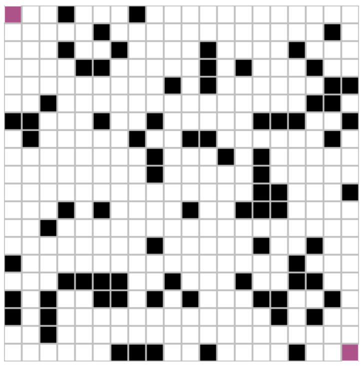
Figure 2: A really Awesome Image

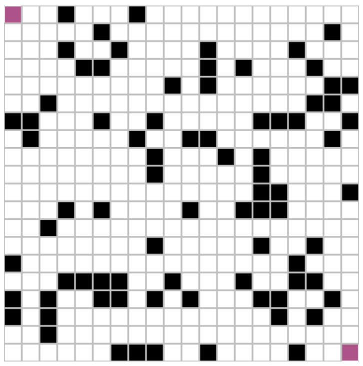
Figure 3: A really Awesome Image

After getting the result, we plotted the rate of successfully finding a path vs the p-value. See Figure 4.

Then we tried to decrease the step size, and selected the p-value from 0.16 to 0.40 as the x-label to generate a new plot for better analyzing the threshold value here. See Figure 5.

After analyzing the result, we notice that the when p is at 0.40, the success rate stays at 0. Thus we came to conclusion that the threshold is 0.40 for p here. And to gain a 50% of success rate, the p-value should be set to 0.30.

In this problem, we used the Bidirectional A^* method, and the optimal distance function is Manhattan Distance. After the calculation, we found that BDA^* consumes less time than other algorithms. In this specific problem, because we simplified the settings, we could only move to 4 directions. Manhattan Distance better simulates such scenarios thus it has the best performance here.

4 For a range of p values (up to p_0), generate a number of maps and estimate the average or expected length of the shortest path from start to goal. You may discard all maps where no path exists. Plot your data. What path finding algorithm is most useful here?

We set p from 0.05 to 0.35 and the step size was 0.01. For every p, we generated 100 mazes and let the algorithm solve the maze. At this time, all mazes were solvable. We used A^* with Manhattan Distance to solve the problem because A^* algorithm could return the shortest path. For every p, we took the average of 100 shortest path, and we plotted its relation with the probability p. See Figure 6:

From Figure 6, we can see that when p is less than 0.2, the average shortest is the same. Actually, it equals to the length of the shortest path from the upper left to the lower down. When p becomes greater, the average length rises fast. When p is beyond p_0, the average rises fast.

5 For a range of p values (up to p_0), estimate the average length of the path generated by A^* from start to goal (for either heuristic). Similarly, for the same p values, estimate the

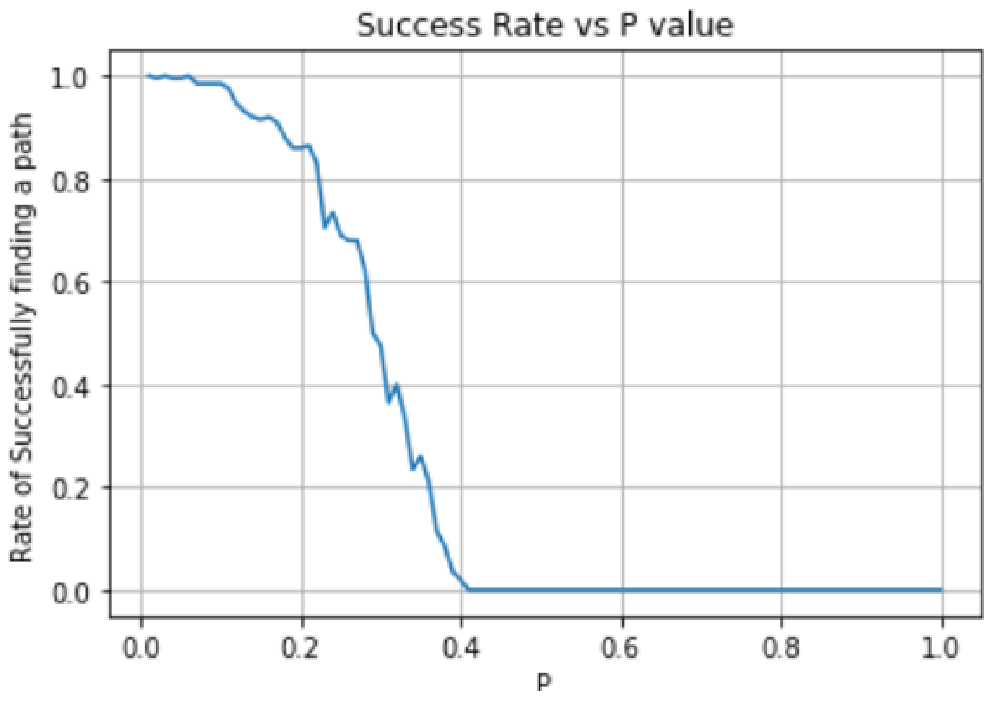

Figure 4: rate of successfully finding a path vs the p-value.

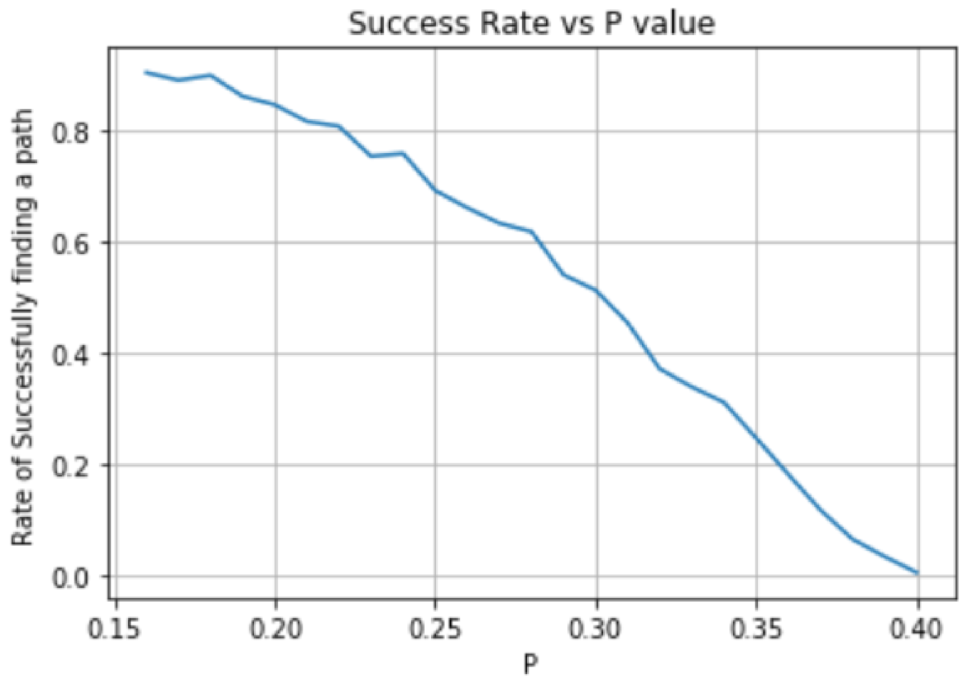

Figure 5: rate of successfully finding a path vs the p-value.

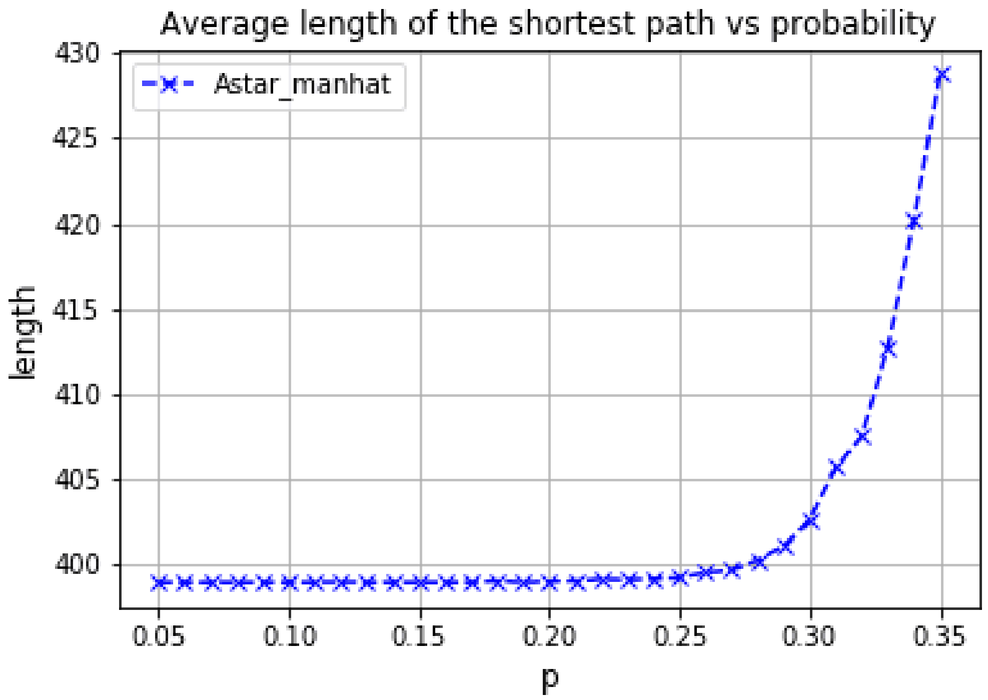

Figure 6: rate of successfully finding a path vs the p-value.

average length of the path generated by DFS from start to goal. How do they compare? Plot your data.

We generated 100 mazes under different p and used 3 algorithms – A^* with Euclidean Distance, A^* with Manhattan Distance, and DFS(quickGoal = True, randomWalk = True) to solve the mazes. All mazes are solvable. We calculated the average path of each algorithm and plot them together. The result is shown in Figure 7:

From Figure 7, we can see that A^* with Euclidean Distance and A^* with Manhattan Distance have the same average shortest path. It is easy to understand this result because A^* returns the shortest path no matter what heuristic function is used. The average length of DFS is larger than A^*. DFS is fast, but it can't return the shortest path when it is not optimized. When p is greater than 0.2, the difference increases quickly. When p is large, there exists oscillation in the average length of DFS. However, we could come to conclusion that the overall trend is rising.

6 For a range of p values (up to p_0), estimate the average number of nodes expanded in total for a random map, for A^* using the Euclidean Distance as the heuristic, and using the Manhattan Distance as the heuristic. Plot your data. Which heuristic typically expands fewer nodes? Why? What about for p values above p_0?

In this question, we generated 200 different mazes to calculate our result and the mazeSize we used is 200×200 here. We used the Euclidean distance as the distance function. The result is demonstrated below as Figure 8.

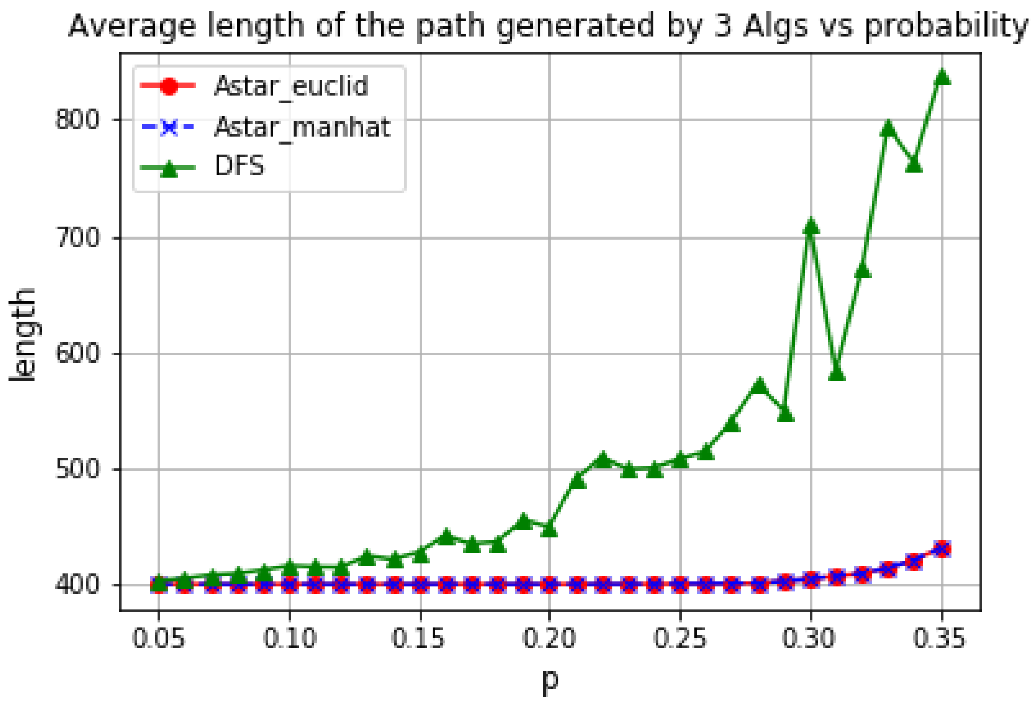

Figure 7: rate of successfully finding a path vs the p-value.

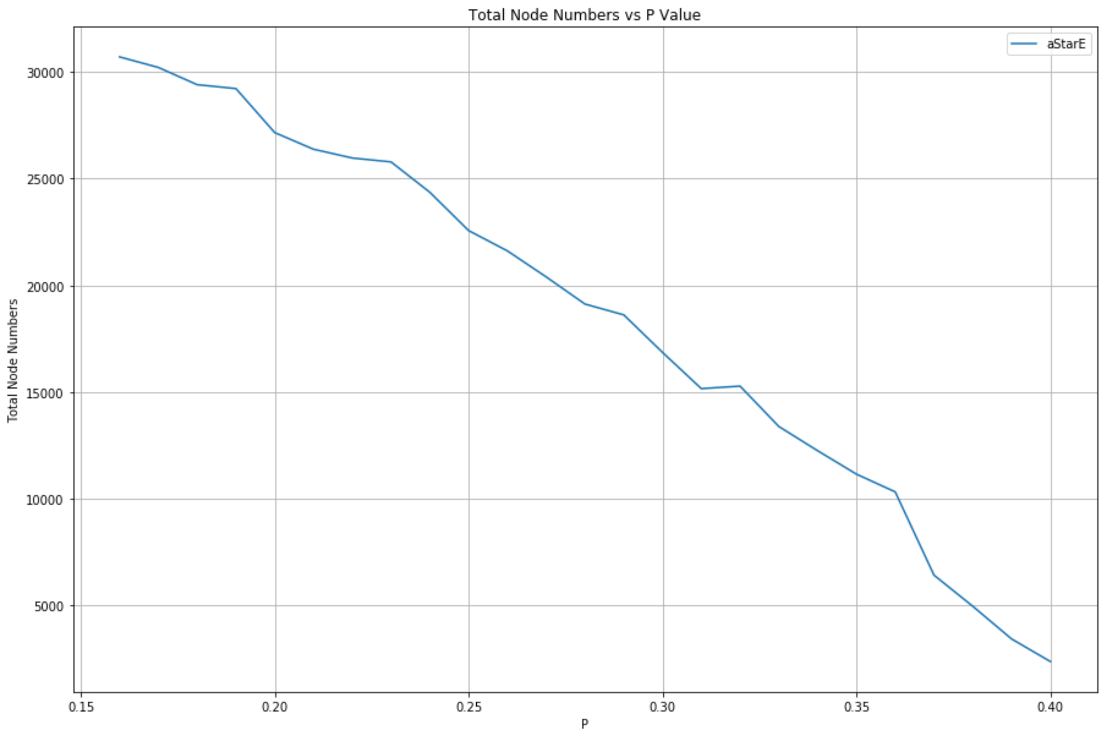

Figure 8: A^* with Euclidean Distance.

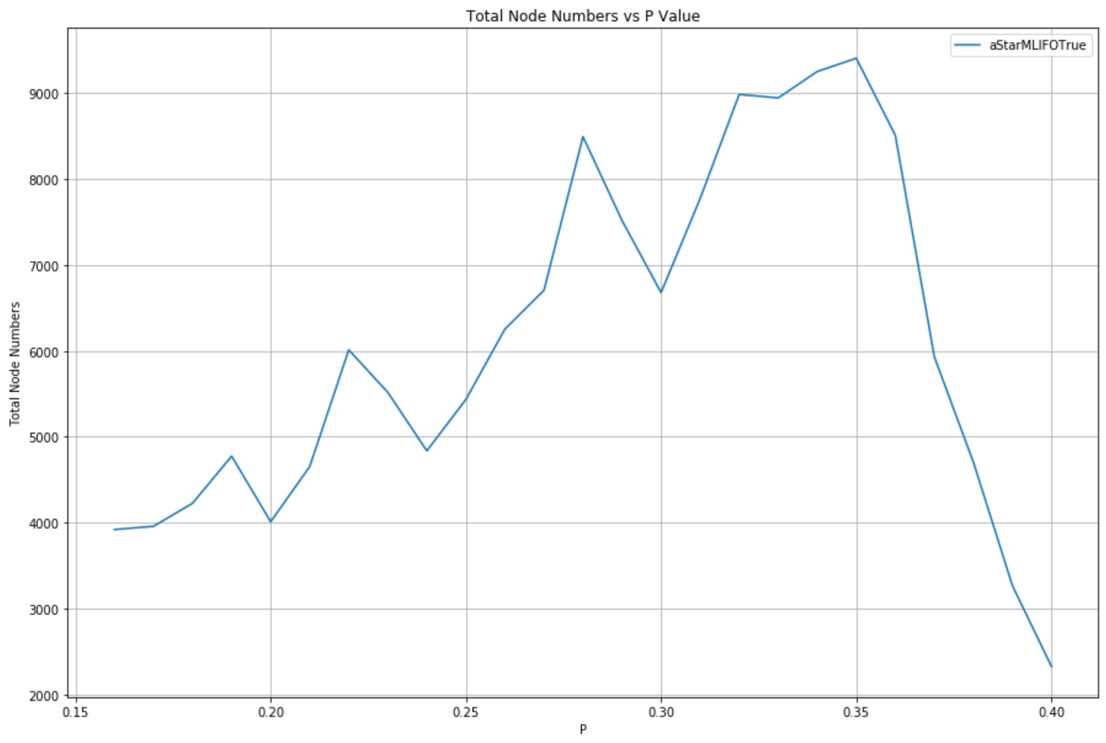

Figure 9: A^* with Manhattan Distance when LIFO=True.

Note that LIFO is our special bool parameter. When LIFO=True, for nodes whose heuristic values are same, the last node entering the priority queue gets popped first.

When we used the Manhattan Distance, we conducted two tests. Figure 9 was the results of LIFO=True and Figure 10 was the results of LIFO=False.

In this scenario, Manhattan distance when LIFO=True clearly expands fewer nodes. This is because that Manhattan is a better simulation in this scenario. We simplified the situation such that the node could only explore four directions, which suits the way M distance works. As for LIFO, when p is small enough, the whole maze almost has no walls. In this case, almost every node has the same cost thus LIFO is similar to DFS, while FIFO could be regarded as BFS.

For the Total Node Numbers after p_0, we have Figure 11, Figure 12 and Figure 13:

From these figures, we could tell when $p >= p_0$, their shape are similar.

7 For a range of p values (up to p_0), estimate the average number of nodes expanded in total for a random map by DFS and by BFS. Plot your data. Which algorithm typically expands fewer nodes? Why? How does either algorithm compare with A^* in Question (6)?

In this question, we generated 200 different mazes to calculate our result and the mazeSize we used is 200×200 here. The result for DFS is shown as Figure 14:

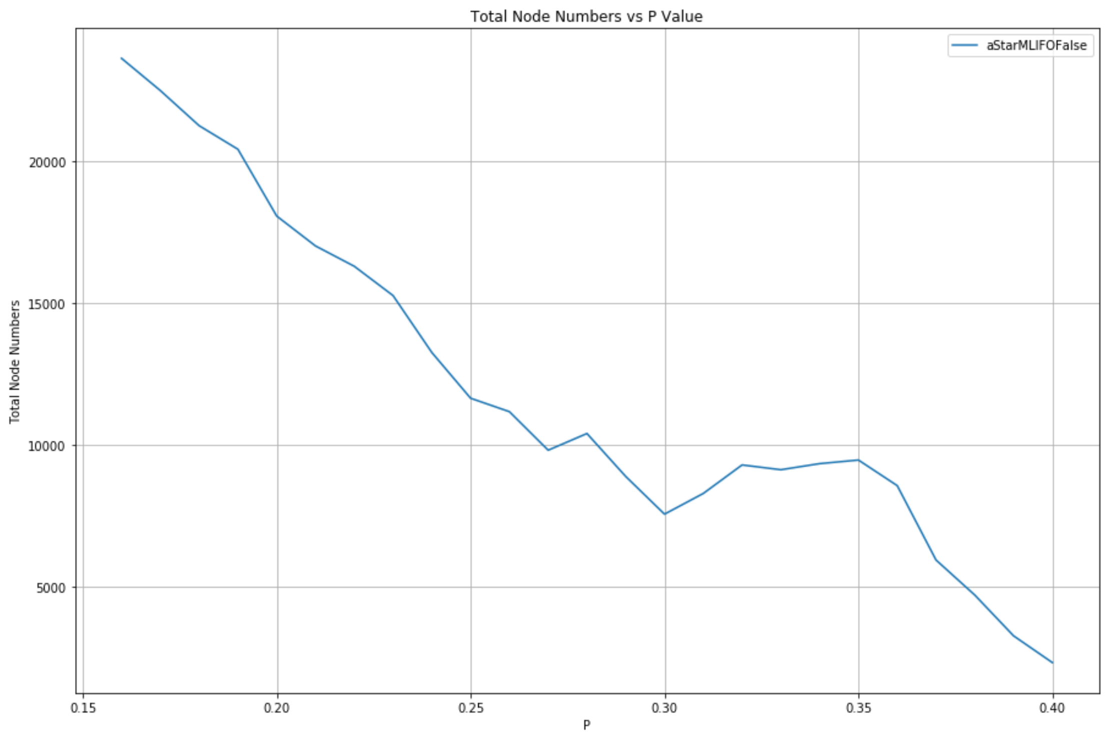

Figure 10: A^* with Manhattan Distance when LIFO=False.

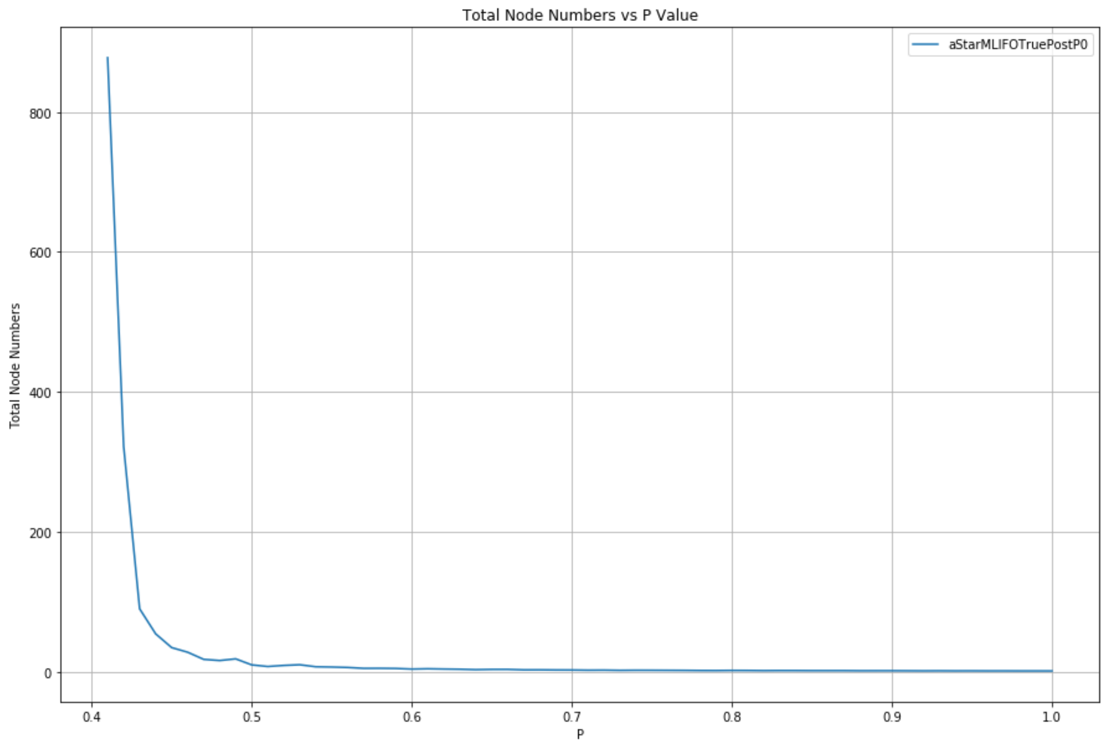

Figure 11: A^* with Manhattan Distance when LIFO=True and $p >= p_0$.

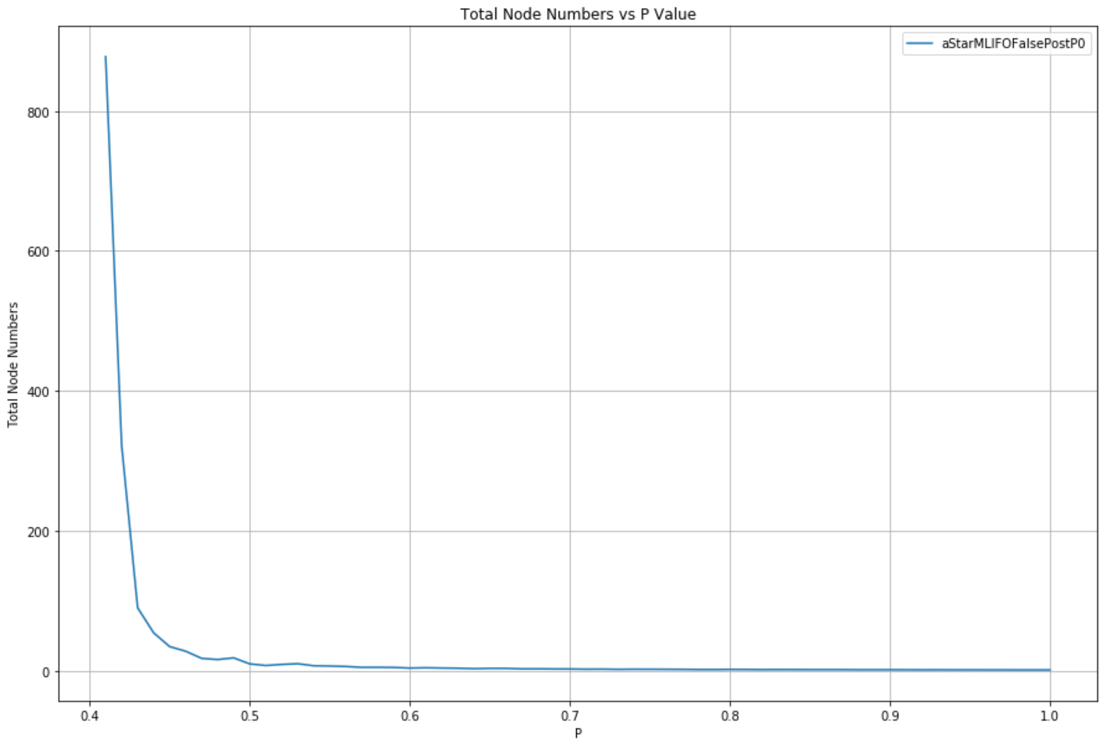

Figure 12: A^* with Manhattan Distance when LIFO=False and $p >= p_0$.

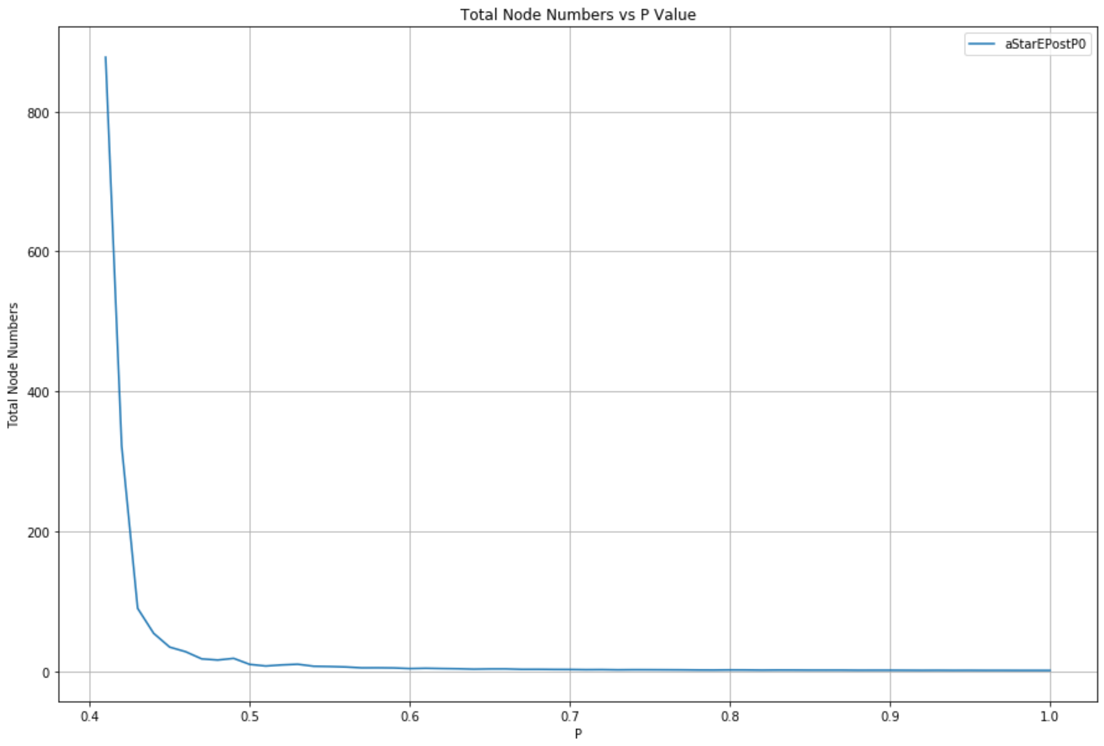

Figure 13: A^* with Euclidean Distance when $p >= p_0$.

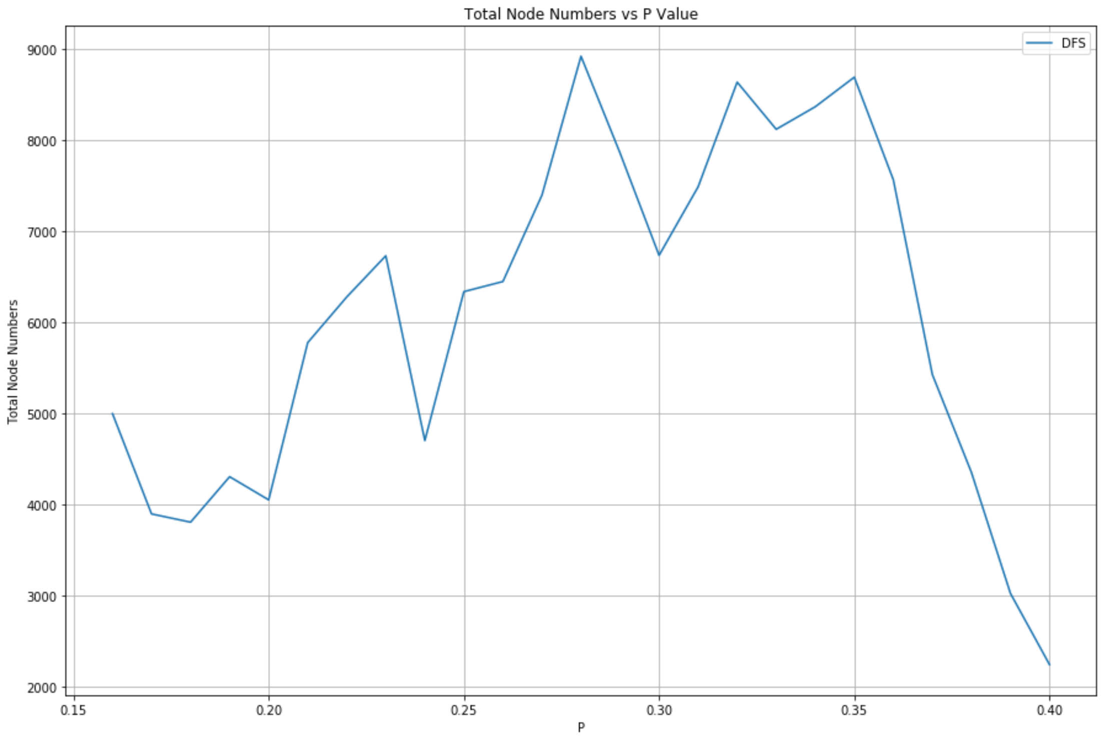

Figure 14: Average number of nodes expanded for different p using DFS.

When we used the BFS, the result is shown as Figure 15:

When p is low, DFS does not need to trackback for a lot of times. As p increases, DFS needs to traceback more. Such that the nodes DFS explores increases as p-value grows. Also, DFS performs better than BFS. When DFS explores a path, it will go down the path until it come to a dead end. At the same time, BFS will explore all the nodes it is adjacent to. In the maze-exploration problem, the correct path often exists in the middle of the nodes, thus typically DFS overperforms BFS.

To draw the comparison, we have Figure 16:

Bonus 1 Why were you not asked to implement UFCS?
UFCS is equal to A^* algorithm with a heuristic function that always returns 0. In this mazerunner case, the cost of all steps are 1, so UFCS is exactly equal to BFS.

3 Part 2: Building Hard Mazes

1. What local search algorithm did you pick, and why? How are you representing the maze/environment, to be able to utilize your chosen search algorithm? What design choices did you have to make to apply this search algorithm to this problem?

   (a) What local search algorithm did you pick, and why?

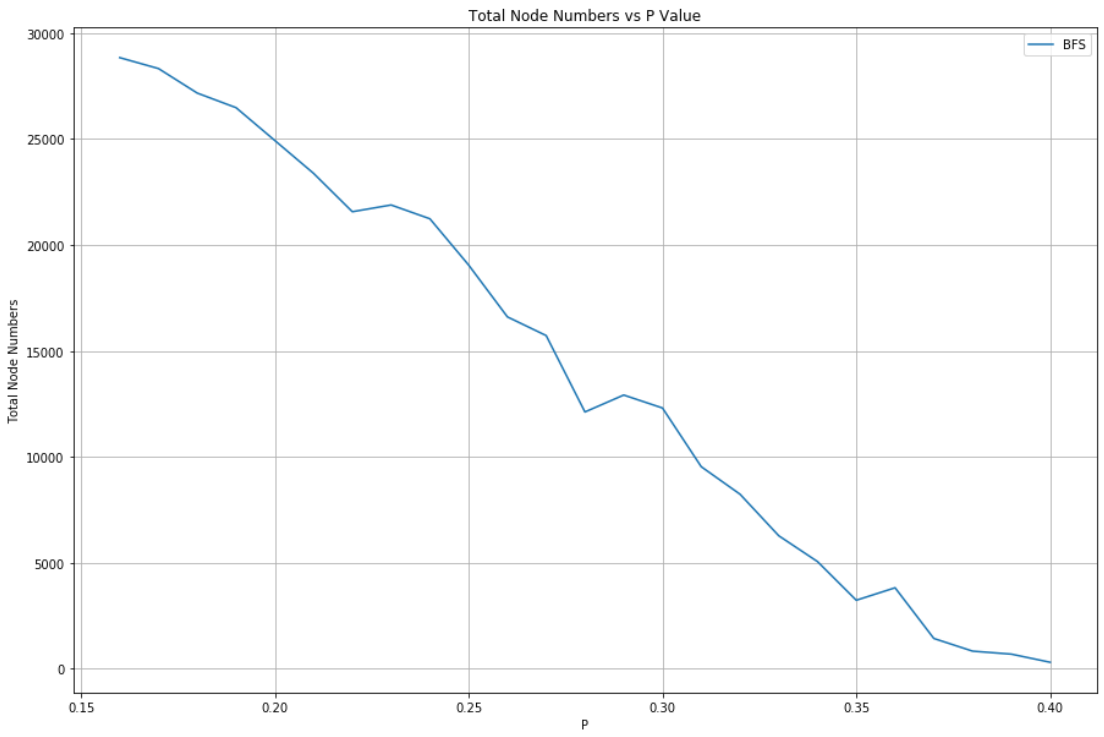

Figure 15: Average number of nodes expanded for different p using BFS.

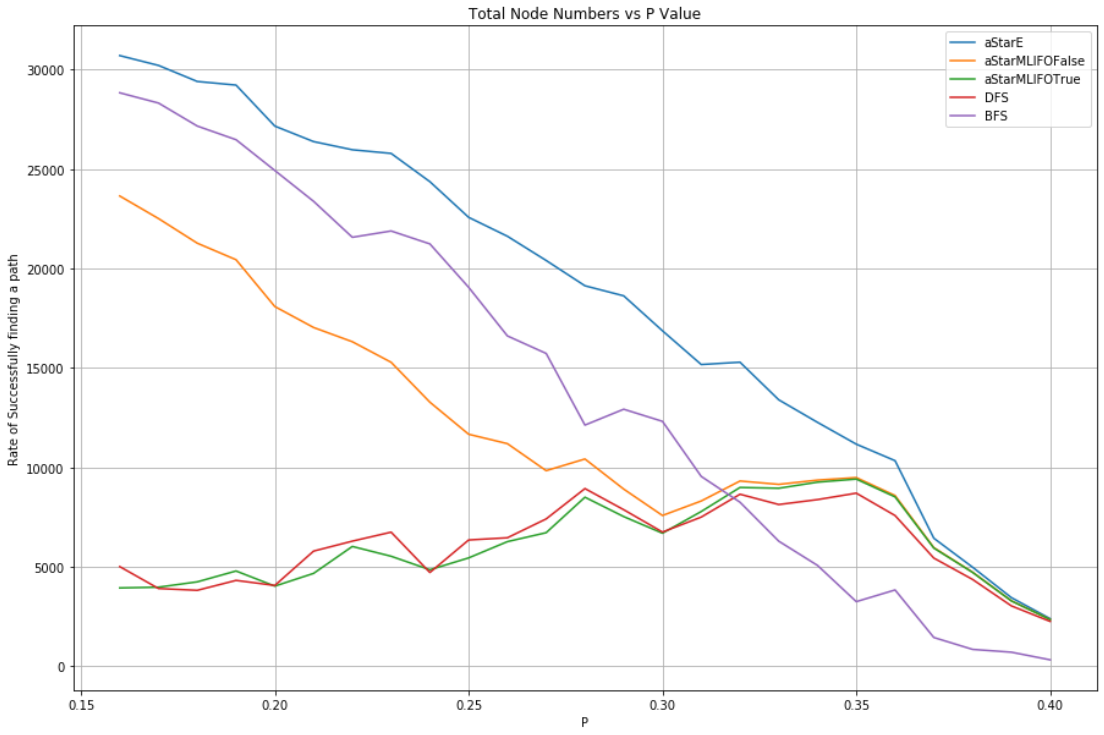

Figure 16: Comparison of average number of nodes expanded.

We combined Genetic Algorithm with Simulated-Annealing-Based Beam Search. We have several reasons:

i. Genetic Algorithm works much better than Simulated-Annealing-Based Beam Search when the mutation rate is large.

ii. Genetic Algorithm usually returns mazes with the "pattern"(or say "schema") that makes a maze hard. However, it is hard to finetune the "pattern" when its size is too large because the pattern is likely to be cut into 2 halves and unlikely to be pieced together again. For instance, (one of) the ideal pattern of the longest path maze should seem like a Hilbert curve, which is as large as the maze. A slight dislocation will make the maze unsolvable. Hence we need another algorithm to finetune.

iii. Beam Search requires a brunch of initial states(mazes), which perfectly matches the population of final iteration from Genetic Algorithm. Also, Simulated Annealing can be helpful when it runs into a local maximum. Combine Beam Search and Simulated Annealing to both utilize most mazes that Genetic Algorithm returned and avoid converging at a local maximum.

(b) How are you representing the maze/environment, to be able to utilize your chosen search algorithm? What design choices did you have to make to apply this search algorithm to this problem?

Overall:

i. A maze can be represented by a $rows \times cols$ boolean matrix, where True means this block is a wall, and False means the agent can go through this block.

ii. The objective function value(fitness) of a maze is calculated by this formula:

$$ObjectiveFunction(maze) = \begin{bmatrix} BlockCount & PathLength & FringeSize \end{bmatrix} \times \begin{bmatrix} w_{block} \\ w_{path} \\ w_{fringe} \end{bmatrix}$$

If we want to find the hardest maze in terms of a single aspect, set one weight to 1, and set others to 0.

iii. Genetic Algorithm will return much more mazes than what Beam Annealing requires. Therefore, using the idea similar to Simulated Annealing to pick some mazes as seeds of Beam Annealing.

A. Pick a few best mazes as "perfect seeds".

B. Discard mazes whose fitness is lower than some percent, say 90%, of maximum fitness, if the number of remained mazes is larger than the size of Beam Search.

C. Randomly pick some "non-prefect seeds" with the probability:

$$P = e^{\frac{Weight}{PerfectRatio}+Bias}, PrefectRatio = \frac{ObjectiveFunction(otherMaze)}{ObjectiveFunction(bestMaze)}$$

D. Keep picking until the number of ”seeds” equals the size of Beam Search.

iv. After Beam Annealing, compare the fine-tuned result with previous Genetic Algorithm’s result. Return best several mazes of both results. (In this case, Beam Annealing failed because Genetic Algorithm presented a nearly perfect result. However, Annealing allows mazes to get a little easier, which is difficult to become harder again since it is hard to randomly get a perfect maze.

Genetic Algorithm:
TODO

Beam Annealing:

i. A maze’s neighbors are all the mazes that can become this maze by changing several blocks. Therefore, we choose a very small probability to flip each block’s value to generate some neighbors of this maze for Stochastic Beam Search.

ii. Each solution algorithm returns 3 values: the number of blocks it has explored, the length of the path found, and the maximum size of the fringe it has used.

iii. The probability to move down is calculated by this formula:

$$P = e^{\frac{Weight*\Delta E+Bias}{Temperature}}, \Delta E = ObjectiveFunction(newMaze) - ObjectiveFunction(maze)$$

A larger $Weight$ can decrease P without any other influence. A larger $Bias$ mainly decrease the probability to keep step forward and back in a ”plateaux”, which works together with ”impatient halt”(explained in Question 9).

iv. Decrease the temperature by multiply a const smaller than 1.0, called ”cool rate”, which makes the temperature go down fast at first and then getting slower and slower to let agent have more chance to climb up(explained in Question 9).

v. Speaking of Beam Annealing, the key point is to make an agent can ”regress” to where it was, especially when an agent ”teleports” to another agent’s region. Hence, once an agent teleport, one of its neighbors is set to a maze in the previous region.

vi. In order to increase the performance, each neighbor maze will be check if it is solvable by using BDA*. If it is unsolvable, regenerate it.

2. Unlike the problem of solving a maze, for which the ‘goal’ is well-defined, it is difficult to know when we have constructed the ‘hardest’ maze. That being so, what kind of termination conditions can you apply to your search algorithm to generate ‘hard’ if not the ‘hardest’ mazes? What kind of shortcomings or advantages do you anticipate your approach having?

Genetic Algorithm:
TODO

Beam Annealing:
There are 3 different ways to halt the iteration:

(a) Maximum Iteration Limit: Count the time it iterates. Once it exceeds the limit, halt and return the result.

  - It is the easiest way to terminate iteration, but actually, there is no reason to terminate it EXACTLY there, except that it has run too many times.

(b) System Cooled Down: As time goes(it iterates), the temperature of Annealing decreases. If it is below a minimum temperature, halt and return the result.

  - If the temperature is too ”cold”, there is no need to keep Annealing. No easier mazes could be picked up. It is reasonable.
  - It related to the cool rate. If the cool rate is too small, the temperature will decrease too fast, and there is not enough time for a maze getting harder after becoming easy. However, if the cool rate is too large, the temperature will keep a high level for a long time, which may totally destroy the work Genetic Algorithm has done.

(c) No Patience Left: If agents do not move for a really long time, it must converge at a local optimal. It is time to halt and return the result.

  - Since we have found a local optimal, we should return it.
  - In the beginning, it is less likely to converge. Hence we should iterate it for a little more times. But if it has been iterated for lots of times, it is unnecessary to keep iterations going.
  - When an agent climbs on a ”plateaux”, it will get lost. Hence, we should limit its ability to move to an equal-difficult maze by using *Bias* to calculate the probability to move.
  - The issue is, sometimes, Genetic Algorithm returns a ”nearly perfect” maze. In this case, it takes at least 30 iterations to halt, since we assume what Genetic Algorithm did is just pre-training.

3. For each of the following algorithms, do the following: Using your local search algorithm, for each of the following properties indicated, generate and present three mazes that attempt to maximize the indicated property. Do you see any patterns or trends? How can you account for them? What can you hypothesize about the ‘hardest’ maze, and how close do you think you got to it?