

# Assignment 1

## **Part 0**

We generate 50 gridworlds with a size of  $101 \times 101$  and save them in 1.txt, 2.txt, 3.txt,..., 49.txt, 50.txt files under the ./gridworld folder, so that every time we run the algorithm, we only need to read the saved txt file, which can also ensure that when we compare the two algorithms, they use the same gridworld. In each gridworld, each cell has a 30% probability of being blocked and a 70% probability of being unblocked. Then we use the BFS algorithm to verify whether there is a path from the upper left corner of the grid to the lower right corner. If the path exists, this grid is added to the set of gridworlds. Otherwise, discard it and continue generating.

We choose breadth-first search (BFS) to generate, in fact, we could have used depth-first search (DFS) or A\* algorithm. However, we did not choose the latter two because:

- (1) DFS: our grid size is  $101 \times 101$ , in the worst case, DFS may require a recursion depth of more than 1000 (Python's default maximum recursion depth), we can expand this maximum recursion depth, but our program will use a large amount of memory.
- (2) A\*: A\* can find paths faster in many cases by using heuristic-guided search, but it is not necessarily the fastest for the small task of just checking path existence, because it needs a lot of unnecessary computation like computing and updating h, f, g, etc. since its goal is to find the shortest possible path.

For BFS, it explores all the vertices of a graph (cells, in our case) in breadth-first order. We choose it because of its simplicity and efficiency. BFS runs in  $O(V+E)$  time where V is the number of vertices and E is the number of edges (the path connecting a cell and one of its four adjacent cells, in our case). In our case of grid, it's  $O(n^2)$  where n is the width of the grid. The number of vertices is  $n^2$  and number of edges is  $4 * n^2$ , so  $O(V+E)$  is  $O(n^2)$  in our case.

## **Code explanation of Part 0**

part0.py is the code used to complete part 0, run it and we will get 50 grids generated

according to the requirements and they are saved under the ./gridworld folder, named 1-50.txt. I've already created them.

### **Part 1**

In the following figure 1, in the case that only 4 adjacent cells can be seen, the agent cannot see any blocked cells at cell A, so it can only judge where to go by calculating  $h$ , and we use Manhattan distance as a heuristic function. Without considering the blocked cells, after moving to the north and reaching (2, D), the Manhattan distance to cell T becomes 4, and after moving to the east and reaching (3, E), the Manhattan distance to cell T becomes  $2 < 4$ . It is closer to the target when moving to the east. So, it chooses to go east.

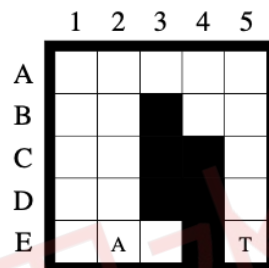


Figure 1 – Figure of question a) of part 1

(b) Based on the graph theory, in a finite gridworld, each cell can be considered as a node in a graph, and each movement from one cell to another adjacent node can be considered as an edge connecting two adjacent nodes. If there are no blocked cells separating the agent from the target, then there exists a path (or multiple paths) from the agent to the target since the start node and the target node are connected.

In A\* search, the algorithm ensures that every reachable node is explored. This is because it uses an open list to keep track of the nodes to be explored, and every time it chooses the node with the smallest  $f$ -value to explore (the  $f$ -value is designed to ensure that the search process always prioritizes nodes that move toward the goal), until the target node is taken out. Therefore, as long as the target is reachable, it must be found within a limited time.

However, if the goal node is not reachable from the start node, then A\* search will also find this in finite time. This is because the search process stops when all reachable nodes have been explored (that means, they have all been removed from the open list and

added to the closed list), but the target node has not yet been explored.

As for the upper bound of the running time, it's the square of the number of unblocked nodes. This is because in the worst case each node may need to be compared against all other nodes in the open list. Therefore, if there are  $n$  unblocked nodes, the upper bound of the running time is  $O(n^2)$ .

## **Part 2**

We implement two versions of the repeated forward A\* using the grid of the figure 2 to get the path and we show the visualization results for comparison in figure 3.

	1	2	3	4	5
A	A				
B					
C					
D					
E					T

Figure 2 – Figure of question of part 2

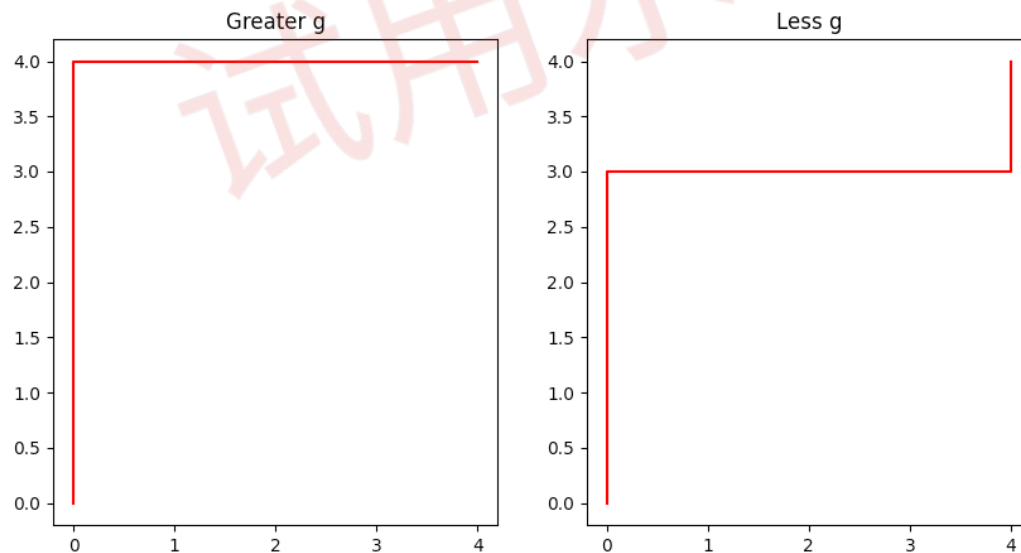


Figure 3 – Visualization results of part 2

Analysis by visualization results and principles:

Select a point with a greater g value (Greater g): Prioritize the point that is farther away from the starting point, that is, it is more inclined to move forward directly.

Select a point with a smaller g value (Less g): Prioritize the point that is not that far,

that is, it is more inclined to explore unknown routes.

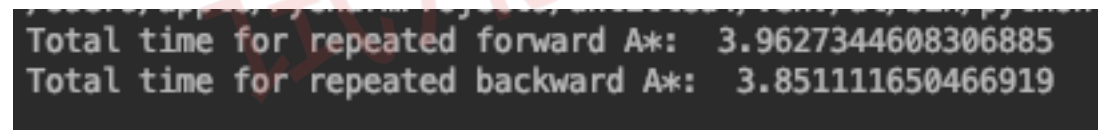
This explains the two different paths of the visualization results. In the Greater g strategy, it first tries to follow the line  $x = 0$ , then turn right when it can't continue. In the Less g strategy, the algorithm is more inclined to explore new routes, it tries to turn right earlier.

### **Code explanation of Part 2**

part2.py is the code used to complete part 2, run it and we will generate the grid of figure 2 and compute using the two different methods separately and compare them visually.

### **Part 3**

We implement and compare the running time of Repeated Forward A\* and Repeated Backward A\*, we only compare the modules that actually perform calculations in our code (that means, we compare forward.solve() and backward.solve(), see our code), that is, we do not consider the time to read grid from txt files (because this part involves reading files and it may take a lot of time to read the file, so we can't compare the real calculation time), the final result is as follows:



```
Total time for repeated forward A*: 3.9627344608306885
Total time for repeated backward A*: 3.851111650466919
```

Figure 4 – Comparison of repeated forward A\* and repeated backward A\*

We can see that repeated forward A\* costs 2.9% more time than repeated backward A\*.

The possible reasons are:

Repeated forward A\*: The agent begins from the start and uses A\* to find a path to the target. It searches from the current cell of the agent toward the target. As the agent moves, when it encounters unknown obstacle, it returns and repeats the search.

Repeated backward A\*: The direction of search is reversed, it searches from the target toward the current cell of the agent. The agent is moving towards the target while expanding cells in the backward direction. This often results in fewer expansions as the target (which is the starting point for the search) is fixed. So, in our case, repeated backward A\* is faster. However, repeated forward A\* may be more efficient in

situations where the start is fixed and the target is changing, such as in multi-objective path planning.

Besides, the following is the comparison of visualization results:

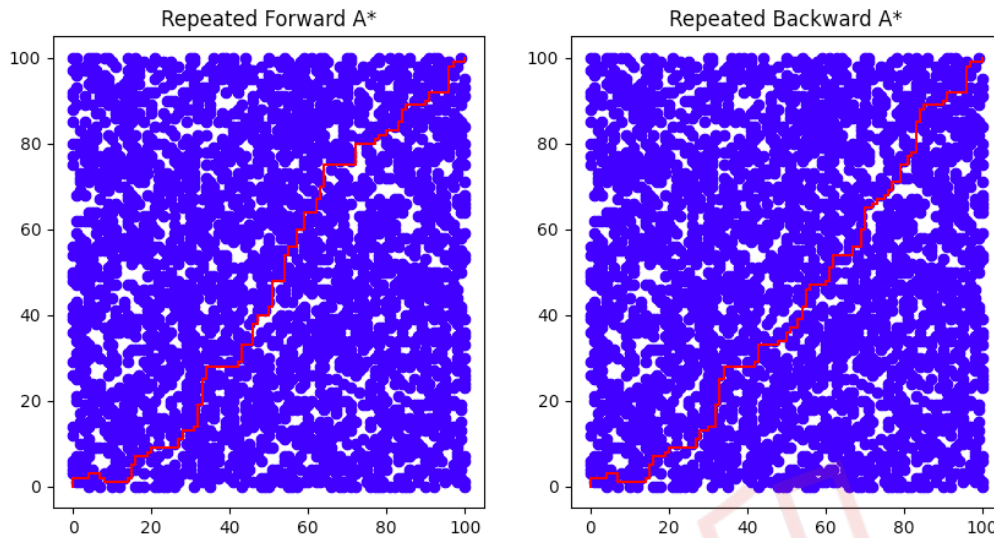


Figure 5 - Comparison of visualization results of repeated forward A\* and repeated backward A\* for grid of file 1.txt (blue: blocked cells, red: path)

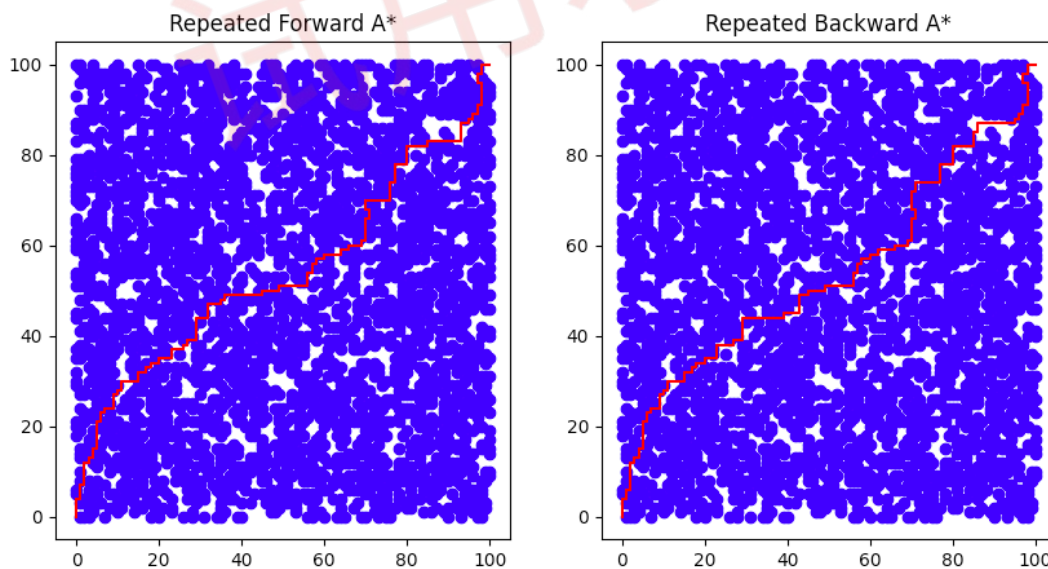


Figure 6 - Comparison of visualization results of repeated forward A\* and repeated backward A\* for grid of file 2.txt (blue: blocked cells, red: path)

### **Code explanation of Part 3**

part3.py is the code used to complete part 3, run it and we will show the visualization

results and the comparison of the runtime. In part3.py, we have the class Cell and the class AStar, we also have a class RepeatedAStar, which inherits the class Astar and has self.forward. When self.forward=True it computes using repeated forward A\* and when self.forward=False it computes using repeated backward A\*.

#### **Part 4**

(1) We prove that “Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions”:

A heuristic  $h(n)$  is consistent or monotonic if for every node  $n$  and every a successor  $n'$  generated by one or several actions from  $n$ , the estimated cost to reach the target from  $n$  is smaller or equal to the cost to get to  $n'$  from  $n$  plus the estimated cost to reach the goal from  $n'$ .

That means,  $h(n) \leq h(n') + c(n, n')$ , where  $c(n, n')$  is just the cost to get to  $n'$  from  $n$ . Note  $n = (x_n, y_n)$  and  $n' = (x_{n'}, y_{n'})$ . In our case, assume all steps of moving up, down, left, or right cost the same (cost 1), then  $c(n, n') = |x_n - x_{n'}| + |y_n - y_{n'}|$  since agent can only move in these four directions.

Note target =  $(x_t, y_t)$ , then  $h(n) \leq h(n') + c(n, n')$  becomes:

$$|x_n - x_t| + |y_n - y_t| \leq |x_{n'} - x_t| + |y_{n'} - y_t| + |x_n - x_{n'}| + |y_n - y_{n'}|.$$

Since  $|x_n - x_t| \leq |x_n - x_{n'}| + |x_{n'} - x_t|$  and  $|y_n - y_t| \leq |y_n - y_{n'}| + |y_{n'} - y_t|$  are always true because of the triangle inequality, so  $h(n) \leq h(n') + c(n, n')$  is always true. Thus,  $h(n)$  is consistent.

(2) We prove that Adaptive A\* leaves initially consistent h-values consistent even if action costs can increase:

Adaptive A\* updates the heuristic values after each iteration based on the actual cost observed during the search. After a path has been found, Adaptive A\* sets the heuristic  $h_{new}(s)$  of a state  $s$  to be the cost of the path found from  $s$  to the target.

To prove the consistency, we need to prove  $h(n) \leq h(n') + c(n, n')$ .

Suppose that  $h(s)$  is updated to a new value  $h_{new}(s)$ , which is the cost of the path found from  $s$  to the target. For any successor  $s'$  of  $s$  along this path,  $h_{new}(s') + c(s, s')$  is exactly  $h_{new}(s)$ . Thus, the heuristic  $h_{new}(s)$  is consistent.

#### **Part 5**

We implement and compare the running time of Repeated Forward A\* and Adaptive A\*, we only compare the modules that actually perform calculations in our code (that means, we compare `forward.solve()` and `adaptive.solve()`, see our code), that is, we do not consider the time to read grid from txt files (because this part involves reading files and it may take a lot of time to read the file, so we can't compare the real calculation time), the final result is as follows:

```
Total time for repeated forward A*: 3.959256887435913
Total time for adaptive A*: 2.0575876235961914
```

Figure 7 – Comparison of repeated forward A\* and adaptive A\*

We can see that repeated forward A\* costs 92.4% more time than adaptive A\*! The adaptive A\* is much faster than the repeated forward A\*! The possible reasons are:

**Adaptive A\*:** During the search process, it will update the heuristic estimate of each node that has been expanded to make it equal to the real cost from the node to the target. In this way, Adaptive A\* will give priority to those nodes that are closer to the target node in the subsequent search process, so the target can be found faster.

**Repeated Forward A\*:** In contrast, it uses a static heuristic function (Manhattan distance, in this case), and does not adjust during the search progresses. Therefore, it may expand more nodes in the process of finding the target, resulting in increased running time.

So, adaptive A\* can more accurately point to the target and reduce unnecessary node expansion by dynamically adjusting the heuristic estimation, so it performs better than repeated forward A\* in terms of running time.

Besides, the following is the comparison of visualization results:

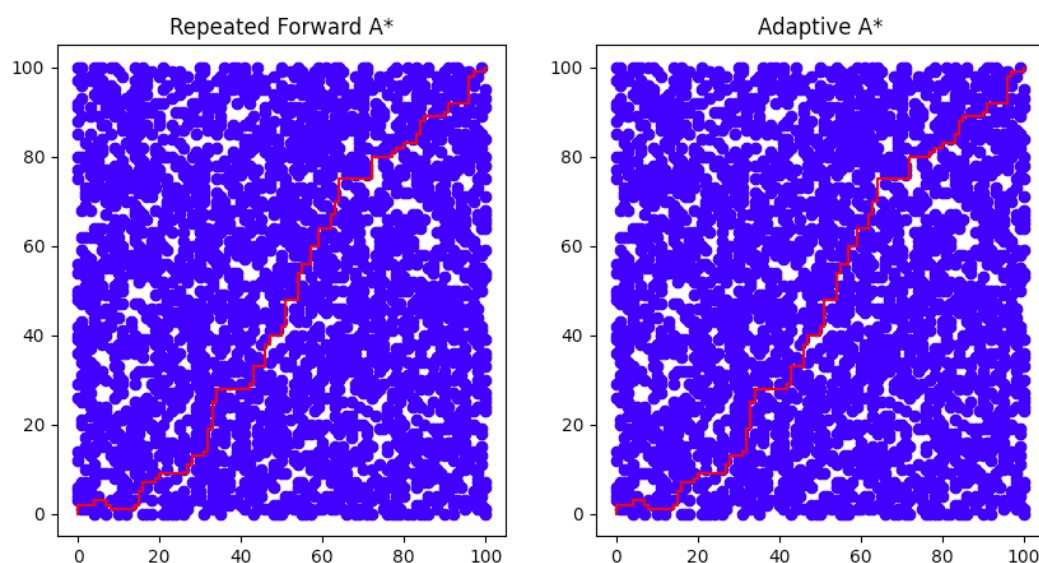




Figure 8 - Comparison of visualization results of repeated forward A\* and adaptive A\* for grid of file 1.txt (blue: blocked cells, red: path)

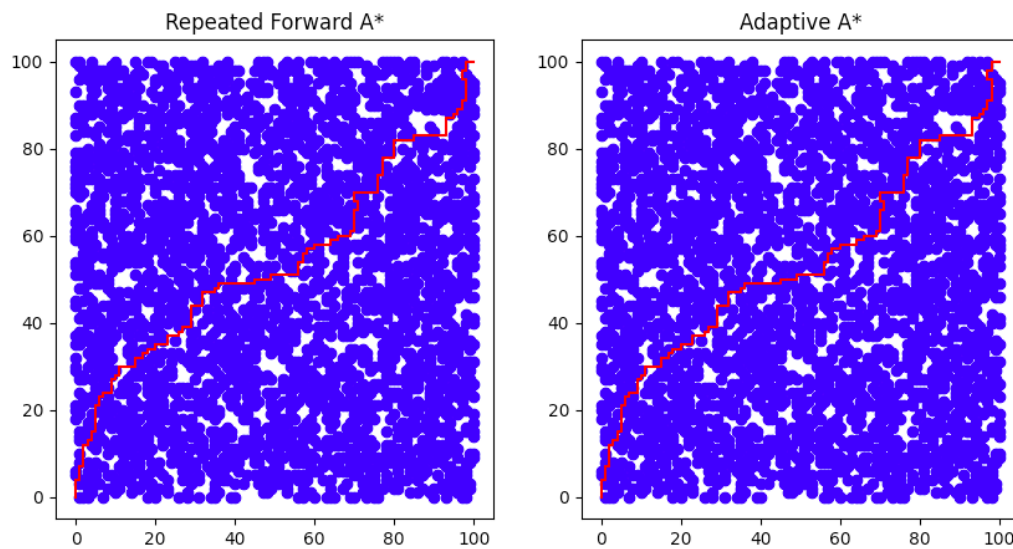


Figure 9 - Comparison of visualization results of repeated forward A\* and adaptive A\* for grid of file 2.txt (blue: blocked cells, red: path)

### **Code explanation of Part 5**

part5.py is the code used to complete part 5, run it and we will show the visualization results and the comparison of the runtime. In part5.py, we have a new class: AdaptiveAStar, which also inherits the class Astar and aims to compute using adaptive A\*. It also has a new function, get\_expanded\_cells, which is used to get expanded cells.

### **Part 6**

In part 5, we observed that the runtime of the Adaptive A\* algorithm seems to be less than the runtime of the Repeated Forward A\* algorithm. However, we need to use statistical hypothesis tests for to decide if this observation is statistically significant.

We can use a two-sample t-test to investigate this question. A two-sample t-test is a statistical method used to compare whether the difference between two independent samples is significant. In this experiment, our two samples are the runtimes of the Adaptive A\* and the Repeated Forward A\* algorithms across multiple runs.

Our steps:

(1) Hypothesis:



H0: There is no significant difference in runtime between the two algorithms, the difference is due to noise.

H1: There is a significant difference in runtime between the two algorithms, the difference is due to the principles.

(2) Data collection:

Perform multiple independent runs of both the Adaptive A\* and Repeated Forward A\* algorithms, recording the runtime for each run.

(3) t statistic:

Calculate the t statistic based on the means, standard deviations, and sample sizes of the two samples.

(4) Significance level:

We choose a significance level (0.05 or 0.01 is OK).

(5) p-value:

Calculate the p-value based on the t statistic and the degrees of freedom.

(6) Conclusion:

If the  $p\text{-value} < \text{significance level}$ , we reject H0 in favor of H1, meaning that we consider that there is a significant difference in runtime between the two algorithms, the difference is due to the principles.

If the  $p\text{-value} > \text{significance level}$ , we cannot reject H0, meaning that we consider that there is no significant difference in runtime between the two algorithms, the difference is due to noise.

---

