Report for HW-1

Group members: Yuwei Huang(yh480)
Tongpeng Zhang(tz136)

The Solution of Each Question

Part 0 - Setup your Environments:

As the question asking, we need to make a program that can generate a random map, which in size 101 * 101. And after random map generating, a shortest routine will be shown at the map by our main function. To get our map, we use Visual Studio and set “1” as blocked path, “0” as unblocked path, and “*” as routine.

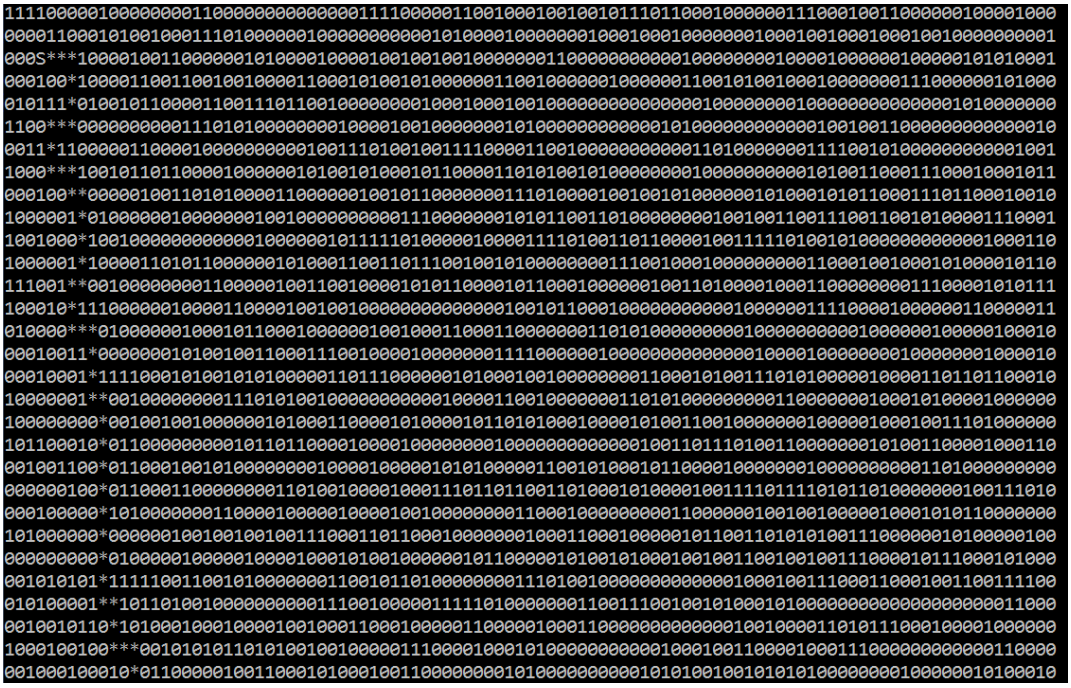

Figure-1 The Initial Environment

Part 1 - Understanding the methods:

(a) After reading the project description, we know that the Repeated Forward A* is “g (Sgoal) > min s’∈OPEN (g(s’) + h(s’)). Even if the agent doesn’t know initially which cells are blocked, using function “f=g+h” to choose the smallest direction. As the project showing, to east is 4, and to north is 6, the agent will choose east.

(b) The first condition is that agent couldn’t reach the target, which means the target is separated by the block cells from agent. Through the function “f=g+h”, the program will pick the smallest value f to get a path, if there is no appropriate value f, that program can’t find path. As the project description showing code:

ComputePath();

if OPEN=Ø

print “ I cannot reach the target.”;

The second condition is that the agent could reach the target. The totally steps of agent to reach target equals the number of unblocked cells. As we first idea was that there is only one path from agent to target, but the result shows that there could be more than one path. Because of the algorithms, program would choose the shortest path, which means there must be some other unblocked cells that agent have never met. So, the number of unblocked cells squared bound from above the number of moves of the agent until it reaches the target or discovers that this is impossible.

Part 2 - The Effect of Ties

When there are several same smallest f-values, the agent needs to choose one of them. To decide pick which one by comparing with these f-values, we use two methods to break the tie, one is to choose the cell with smaller g-values, and another one is to pick the larger g-values. We run ten times of our program to record their runtime and compare it to get the result. And finally, we find the larger g-values cost less time. When agent has several same smallest f-value, that means smaller g-value has a larger h-value, on opposite, larger g-value has a smaller h-value. As a result, the cells with smaller g-value have a larger Manhattan Distance than larger g-value. So the smaller g-value could cause more expanded cells. While the cells with larger g-value usually near the current cell which the agent stayed, so when it choose to expand the cells with larger g-value, it won’t go back to the former area and it just starts searching near the current cell, so obviously, the larger g-value have a better performance.

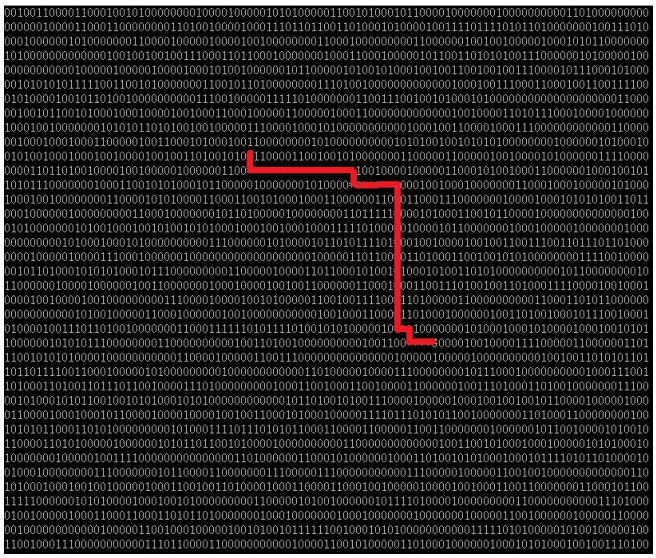

Figure-2 Smaller g-values

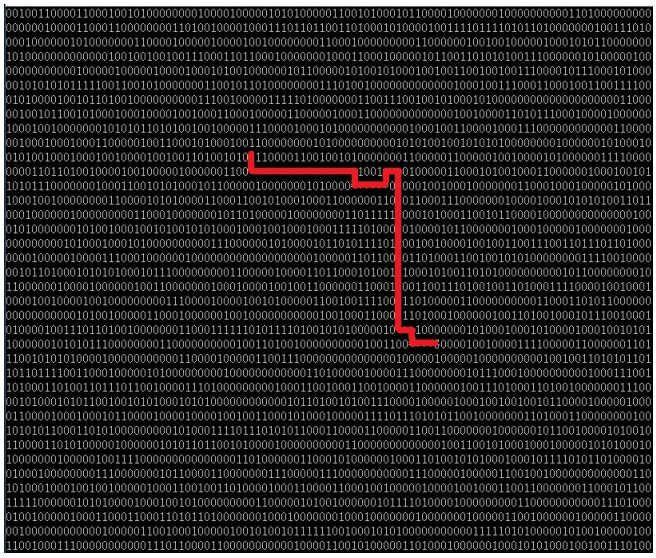

Figure-3 The Larger g-value

Larger gvalue	Smaller gvalue
0.012	0.017
0.036	0.051
0.028	0.056
0.04	0.052
0.009	0.001
0.028	0.048
0.018	0.051
0.077	0.089
0.028	0.1
0.028	0.045

Figure-4 Runtime of Larger-g and Smaller-g

Part 3 – Forward vs. Backward

To get the result of these two different methods, we use the same agent and same target. Because in the detective map, with forward, we can avoid the wall at first, but with backward, maybe it build the path to the goal and near the goal, it meets the wall according to the detective map, then it will find other Nodes with smaller f in the openlist and expand more cells. Actually, at that time, there too many Nodes in the openlist. So expand more cells. Robot moves from the start, so for the forward, if it meets the wall near the goal, the start is also near the wall, it won't expand too many Nodes.

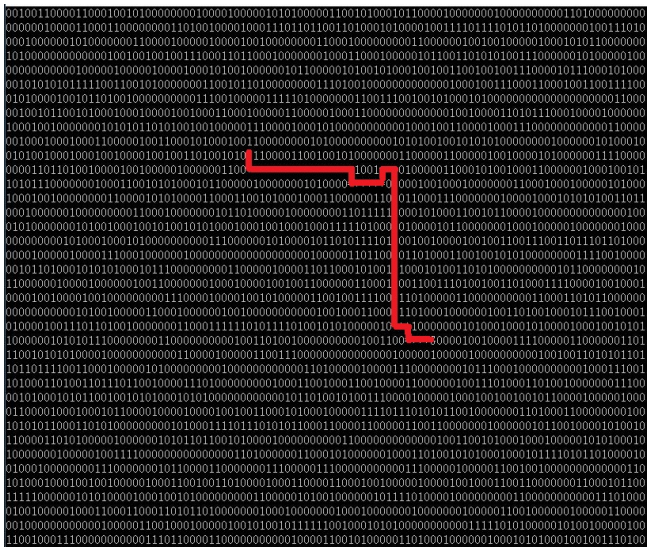

Figure-5 Forward

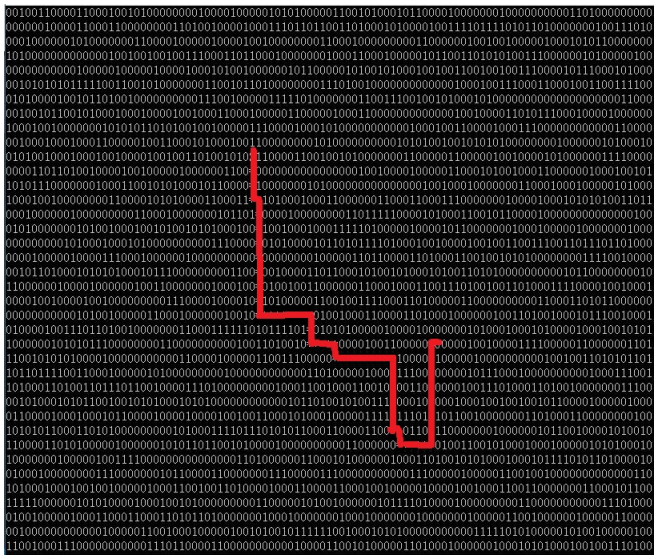

Figure-6 Backward

forward	backward
0.012	0.016
0.036	0.043
0.028	0.016
0.04	0.037
0.009	0.002
0.028	0.051
0.018	0.016
0.077	0.04
0.028	0.018
0.028	0.037

Figure-7 Runtime of forward and backward

Part 4 – Heuristics in the Adaptive A*

A consistent heuristic function is a function that estimates the distance of a given state to a goal state, and that is always at most equal to the estimated distance from any neighboring vertex plus the step cost of reaching that neighbor. Formally, for every node N and every successor P of N generated by any action a, the estimated cost of reaching the goal from N is no greater than the step cost of getting to P plus the estimated cost of reaching the goal from P. In other words:

$$h(N) \leqslant c(N,P)+h(P)$$

$$h(G)=0$$

Where

- *h* is the consistent heuristic function
- *N* is any node in the graph
- *P* is any descendant of *N*
- *G* is any goal node
- c(N,P) is the cost of reaching node P from N

Consistent heuristics are called monotone because the estimated final cost of a partial solution, f (Nj)=g (Nj)+h (Nj) is monotonically non-decreasing along the best path to the goal, where g (Nj)= $\sum$c(Ni-1,Ni) is the cost of the best path node N1 to Nj. It's necessary and sufficient for a heuristic to obey the triangle inequality in order to be consistent.

In the A* search algorithm, using a consistent heuristic means that once a node is expanded, the cost by which it was reached is the lowest possible, under the same conditions that Dijkstra's algotithm requires in solving the shortest path problem (no negative cost cycles). In fact, if the search graph is given cost c' (N,P)= c (N,P) + h (P) –h (N) for a consistent h, then A* is equivalent to best-first search on that graph using Dijkstra's algorithm In the unusual event that an admissible heuristic is not consistent, a node will need repeated expansion every time a new best (so-far) cost is achieved for it.

Part 5 – Heuristics in the Adaptive A*

As the requirement, we use forward A* algorithm and adaptive A* algorithm to compare which one have better performance. The figure-8 shows the two algorithms have a same path, but though the figure-9, we can see adaptive A* algorithm cost less time than forward A*. Because every time as the A* star algorithm find a route, the Adaptive A* use g(goal), the g value from the start state to the goal state and the g(s), the cost from the start state to the current state to update the h value calculated by Manhattan distance. By updating the h value with

the difference of g(goal) and g(s), we can get the new h value. H new is equal to g(goal) – g(s). By this way, the Adaptive A* search can make the following search more focus, so it can expand less nodes and be more efficient than the repeated forward A* search. We record the run time cost of adaptive A* search and the repeated A* search with same start point and goal point. From the data in figure 9, we can find that in most case, the adaptive A* search has less run time than the repeated A*, it means that adaptive A* is indeed more efficient. However, in some case, the repeated A* has less run time cost, we think it may because that the distance in two nodes is not very far and the total nodes is not so many. So the adaptive A* search does not have apparently advantage and may cost much than the repeated A*.

Figure-8 Forward A* & Adaptive A*

forward	adaptive
0.012	0.007
0.036	0.038
0.028	0.024
0.04	0.039
0.009	0.005
0.028	0.029
0.018	0.015
0.077	0.069
0.028	0.027
0.028	0.024

Figure-9 Runtime of Forward A* & Adaptive A*

Part 6 – Memory Issue

In our program, we use Visual Studio 2015 to get the memory and runtime of size 101*101 and 1001*1001. And as the figure-10 & figure-11 shown, we can see the largest gridworld like 1001*1001; the memory usage is larger than smaller one. Because there are more information that is stored per cell, it will cost more memory usage during searching a path.

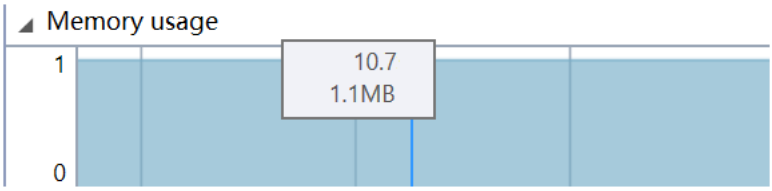

Figure-10 Size of 101*101

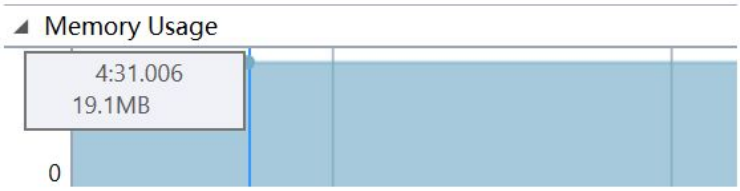

Figure-11 Size of 1001*1001

Reference:

[1]: Consistent Heuristic, Wikipedia,
https://en.wikipedia.org/wiki/Consistent_heuristic