Fast Trajectory Replannng

Assignment 1

Hanxiong Wang, Xiaobing Zhang

Contents

February 2017

1 Part 0: Setup your environment

Before building the 50 grid world, we setup the node class. In the node class, information like it's g value, h value, whether it is searched during n time step and connections between surrounding nodes are stored.
In the buildWorld.py code, two separate maps, one blank and one filled, are created. Both maps contained 50*50 nodes. To create the maze, nodes in the filled map are visited one by one, with 70% probability to mark it as unblocked and 30% chance to mark it as blocked. Pickle package (https://docs.python.org/3/library/pickle.html) is used to help save and load the built grid world. And matplotlib (http://matplotlib.org/) is used to visualize the maps.
The built world are shown in figures below.

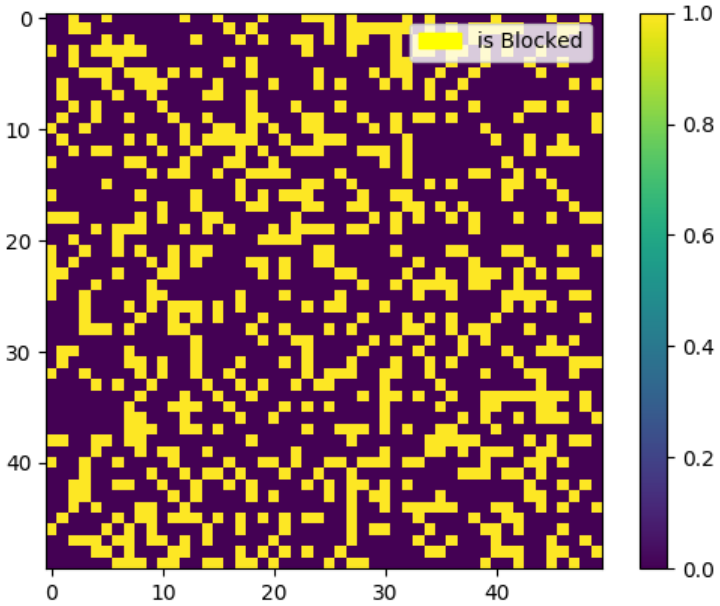

Figure 1: The maze, yellow is blocked.

2 Part 1: Understanding the methods

(a) In Figure 8, the agent doesn't know E4 is blocked initially. And the h value calculated is by Manhattan distance. For E3, h value is 2 and g value is 1, thus f value is 3. While for D2, h value is 4, g value is 1, thus f valuekl is 5. In A* method, the agent always prefer lower f value node. So the first move of agent is to the east rather than the north.
(b) In our project, we estimated the distances between goal and each node by Manhattan distance. So we didn't overestimate the h values. If a solution exists, there is a node M lies on the way to the goal. Each node expansion increases the length of one possible path, the method will eventually expand the node M, thus finding the solution. Unless it is impossible to find the path in finite time.

No matter where the start or goal is, the number of unblocked cells squared gives a square map that holds all the possible moves of the agent. So the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of blocked cells squared.

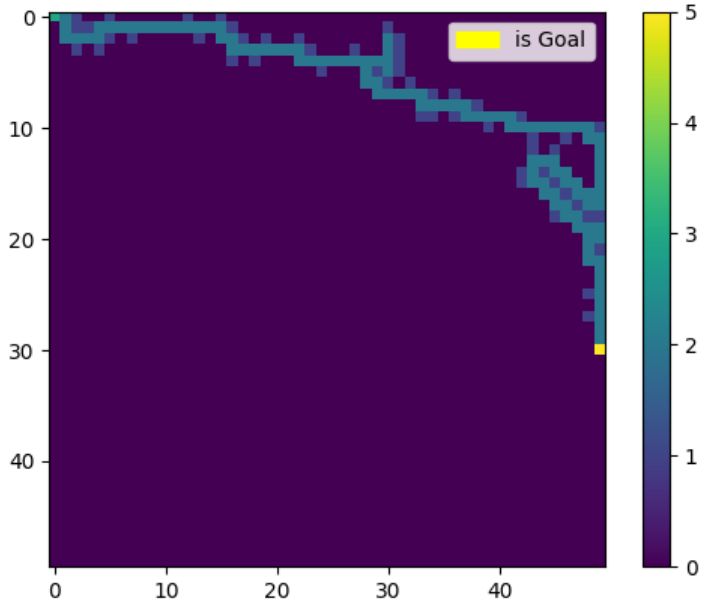

Figure 2: The searched path through A*.

3 Part 2: The effect of Ties

Similar to the algorithm in the hint, 100*f(s) - g(s) is used as priority criterion in favor of larger g-values. 100*f(s) + g(s) is used as priority criterion in favor of smaller g-values.

Table. 1 Compares between different priorities

Priority	Running Time	Total Expanded Cells
Larger g-values	0.07605648040771484	1164
smaller g-values	0.7868521213531494	14465

We kept other conditions the same but only switched the priority standard. By comparing the running time, we could see that larger g-values makes the program much faster, almost by a magnitude. Also, it expands much less cells.
f-values = g-values + h-values = constant. So larger g-values means smaller h-values. The meaning of h-values is the goal distance of state s. In other words, smaller h-values gives states that are 'closer' to the goal. So it takes much less running time to reach the target.

4 Part 3: Forward vs. Backward

To switch from forward A* to backward A*, we need to set g-start as ∞ and g-goal as 0. Search goal instead of start in the main while loop and so on. For ComputePath function, we need to set the target as start instead of goal.
In my code, to switch from forward A* and backward A* is easy and straightforward. Simply set flag as 0 if you want forward A* and set it as 1 if you want to run backward A*.

Table. 2 Compares between forward and backward A*

Name	Running Time	Expanded Cells at last step
Forward A*	0.06891846656799316	1164
Backward A*	0.8037607669830322	5266

The implementation shows that, repeated backward costs almost 10 times running time compared

to repeated forward A*. And in most cases, repeated forward A* expands less number of cells compared to repeated backward A*.
Koenig and Likhachev explained the reason in their paper (Koenig and Likhachev 2005). For A* searches, the agent assumes all cells are unblocked at the beginning. This assumption is 'true' around the goal in initial searches before the agent gets close to the target. So the repeated forward A* expands only the vertices on a shortest path to the goal state. However, for repeated backward A*, the search is not as informed as forward search. So it expands more nodes during the initial searches.

5 Part 4: Heuristics in the Adaptive A*

Assume for a general location of state $s(x_s, y_s)$ and the target $t(x_t, y_t)$, the heuristics of $h(s)$ is

$$h(s) = |x_t - x_s| + |y_t - y_s|. \tag{1}$$

After an action, say the agent move to new state $s_r(x_s + 1, y_s)$ (actually we have four states, right, left, up and down and we chose one of them $s_r$as an example). Then the heuristics of $h(s_r)$ is

$$h(s_r) = |x_t - (x_s + 1)| + |y_t - y_s|. \tag{2}$$

The cost action here is 1 and it is easy to see that

$$h(s) = |x_t - (x_s + 1) + 1| + |y_t - y_s| = 1 + |x_t - (x_s + 1)| + |y_t - y_s| <= 1 + h(s_r). \tag{3}$$

Thus, the Manhattan distance are consistent in gridworlds here.

For adaptive A*, the heuristics of $h(s)$ is

$$h(s) = g(s_{goal}) - g(s) \tag{4}$$

and the state after the action $h(s + a)$ is

$$h(s + a) = g(s_{goal}) - g(s) \tag{5}$$

we have the following relation

$$h(s) = g(s_{goal}) - g(s + a) + g(s + a) - g(a) <= g(s_{goal}) - g(s + a) + cost(a) = h(s + a) + cost(a) \tag{6}$$

which shows the consistency even if the action costs can increase.

6 Part 5: Heuristics in the Adaptive A*

Different start and goal locations are selected and tested. The running time fluctuates but in general the Adaptive A* runs faster than repeated forward. Adaptive A* expanded fewer cells than A*. For adaptive A*, it doesn't need to expand unnecessary cells, thus reducing the memory costs as well. Overall, Adaptive A* ran faster and rated better than A*.

Table. 2 Compares between forward and backward A*

Name	Running Time	Expanded Cells
A*	0.08509135246276855	1164
Adaptive A*	0.06904959678649902	1164

Adaptive A* updates the h-values after each search and could deliver more informed searches in the future steps. Adaptive A* updates heuristics between search steps(Adaptive A*, Sven Koenig and Maxim Likhachev). So it is faster but the difference is not very obvious.
In our example, the Manhattan distance is a pretty good estimation to the actual h-values, so the difference is not that huge. But in some other cases, the difference could be large. In such cases, Adaptive A* will show a much bigger improvement.

7 Part 6: Memory Issues

In general, more information stored gives more informed searches, thus having higher chance to find the shortest path. However, when the gridworld grows huge, the large amounts of information stored may slower or even kill the program. So it is very important for us to pay attention to memory issues. We propose several ways to reduce the memory:
(1) Set an bound for memory. When the memory reaches bounds, delete the worst-rated nodes, i.e. the nodes with largest f-values. In this way, the OPEN or CLOSED lists will not grow.
(2) Removes OPEN or CLOSED lists. Instead, integrates the heuristic estimates into a depth-first search strategy that requires only linear space in the depth of the search.(Memory-Bounded A* Graph Search, Rong Zhou and Eric A. Hansen)

8 Part 7: References

(1) Koenig, S., and Likhachev, M. 2005. Fast replanning for navi- gation in unknown terrain. IEEE Transactions on Robotics and Automation 21(3):354–363.
(2) Zhou, Rong, and Eric A. Hansen. "Memory-Bounded A* Graph Search." FLAIRS conference. 2002.
(3) Koenig, Sven, and Maxim Likhachev. "Adaptive a." Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems. ACM, 2005.