Assignment 1: Fast Trajectory Re-planning

Chaoqin Chen, Mingzhe Li, Guantao Zhao

Part 0 – Set Up Your Environments:

For the set-up grid world environments, we use pygame. In the cell.py we use similar method that is describe in the assignment. The cell.py has Boolean value that can indicate whether a cell is blocked or not. If the value is blocked then mark the cell as blocked cell and return the blocked cell. Then, we import cell in the grid to set-up the blocked and unblocked cells. In the grid.py under the generate method, we have sup-up the probability of 70% being unblocked and adding the neighbors to stack, and also some members related to heuristic search algorithms: f,g,h (f(x) = g(x) + h(x)), bool visited, bool expanded.

For more information, please see our implementation.

Part 1 – Understand the methods:

a. The main reason why the problem search from the east rather than the north because of the heuristic search method. The purpose of heuristic function is to guide the search process in the most profitable path among all that are available. From the problem we can see that the A is in the east of T. so the most profitable path is east rather than north.

b. According from our book, in the agent search terminates $\text{key}(S_{goal}, 0) \leq \text{key}(\text{u}, 0)$, for all u in the open set of agent search, then:
$g_0(S_{goal} \leq w_1 * g^*(g_{goal}))$ when $w_2 \geq 1.0$.
if a search other than the agent terminates in the i^{th} search, then:
$g_i(S_{goal}) \leq w_2 * OPEN_0.Minkey() \leq w_2 * w_1 * g^*(g_{goal}))$.
Therefore the cost of moves of the agent is always bounded within the factor of $w_1 * w_2$. Now, If the numbers of moves of the agent is impossible is bounded from above by the number of unblocked cells squared, then we will have the following:
We know that $OPEN_0.Minkey() \leq w_1 * g^*(g_{goal})$
Since $OPEN_0.Minkey(\quad) \geq \infty = g^*(g_{goal}) \geq \infty$.
When the pseudo code run into an infinite time, the agent will stack in the loop and unable to modified or bounded back.

Part 2 – The Effects of Ties:

In the first, the higher g-score value tends outperforming the lower g-score value, and it was more observable when we run the algorithm and display the speed and efficiency. To know the performant in speed and efficiency of the algorithm. Let that the f-score value of the sum of the g-score and the heuristic cost for each cell. Cells with the same f-score, but having a lower g-score will therefore have a higher heuristic cost, meaning a higher estimate of the moves needed to reach the goal. Cells that have higher g-score will have a lower heuristic time cost and of course will have a better path to the goal cell. So, if we are choosing the higher g-score as the tiebreaker, the agent will move towards the goal cell with reducing the time and distance that will needed, in other word a better performant.

<u>Part 3 – Forward vs. Backward:</u>

The compile time for Forward A* is very similar for every maps(environments). The main difference for our Repeated Forward A* and Repeated Backward A* is that the Repeated Forward A* works fine and have a similar result, but the Repeated Backward A* seems to take very long time on 101x101 map to compile and we did not get a result from it, sometimes it will have a memory error. So, in this situation we can said that the Repeated Forward A* is performing better regarding to the same f-value with larger g-values.

<u>Part 4 – Proving Heuristics in the Adaptive A*:</u>

First, we need to figure out how the heuristic functions affect the performance. As from the book, $\mathrm{h} = |\mathrm{goal.x} - \mathrm{current.x}| + |\mathrm{goal.y} - \mathrm{current.y}|$. We use this idea of Manhattan distances,

we find that when the heuristic function is admissible, we can find the lowest cost path from start to goal. However, the number of expanded cell and the time cost will become larger. On the contrary, when the heuristic function is not admissible, they will have less number of expanded cells and time but the path they lead to will have more cost than the admissible heuristics ones. The reason for this phenomenon is that when the heuristic function is not admissible, it is an over-estimate and thus able to dominate the evaluation function. When the heuristic is dominating the function, the cost will have less influence to the function. In this case, the path will more likely to follow the heuristic's guidance rather than the actual cost. Thus, the cost will become higher and the time and number of expanded cell will become lower. Second, we need to figure out how the different algorithms affect the final path. From the table, we find out that when the A* algorithm uses an admissible heuristic, the cost of the final path is very close to what we get from uniform cost algorithm. However, the performance of inadmissible heuristic A* is worse than uniform cost search. This shows that when the A* has admissible heuristic, it can get the shortest path from any cell to the goal cell while exploring way less cells.

<u>Part 5 – Comparing Heuristics in the Adaptive A* and Repeated Forward A*:</u>

When we are comparing the Adaptive A* and Repeated Forward A*, we can see the main differences in compile time and number of expanded cells. The Adaptive A* tends to have better performance than Repeated Forward A*. The reason is that the Adaptive A* once it compiled the future A* searches do not have to spend the time on heuristic costs. The previous heuristic costs of the cells would have their heuristics be passed into current search to reduce the compile time on current search. As the agent moves toward the goal, the Adaptive A* cells and time will deceasing compare to Repeated Forward A*. So, the Repeated Forward A* is better preform then Adaptive A* because adaptive A* decease the performance from each cell.

<u>Part 6 - memory Issues:</u>

When we talk about memory issues, we already have a problem on Repeated Backward A* Search on 101x101 map. The compiler will take very long take, and finally crash in the end. If we have a size of 1001 x 1001 large gridworld, I don't think that 4 Mbytes memories is enough to run the gridworld. However, our algorithm might be wrong, if this is the case, the amount of memory should be close to 4 Mbytes.

<u>Tables:</u>

Repeated Adaptive A* Test cases	Time taken	Expanded Cells
1	1.562000	16335
2	2.077000	18962
3	3.397000	21539
4	2.616000	17670
5	5.647000	32049
6*	6.220000	30495
7	2.522000	16699
8*	7.178000	28178
9*	5.719000	28681
10	4.146000	27378
11*	7.350000	29533
12	2.421000	20237
13	2.462000	19475
14	2.475000	16920
15	1.442000	15993
16*	6.015000	30526
17	4.897000	33662
18	1.853000	15402
19	2.839000	22645
20*	7.813000	29323
21	2.282000	15739
22	1.636000	18510
23*	5.476000	26016
24	3.508000	29311
25*	0.015000	153
26*	5.632000	28856
27	2.090000	19052
28	1.437000	21681
29*	6.258000	30789

30*	10.09900	42917
31*	6.125000	27309
32*	6.843000	29411
33	2.432000	16104
34	2.123000	15062
35*	6.977000	30623
36	1.450000	16392
37	4.024000	21707
38	2.208000	18094
39	3.413000	20966
40	1.877000	16078
41*	19.19800	88944
42	1.459000	17025
43*	6.594000	29506
44	2.704000	17841
45*	0.015000	153
46	1.683000	19868
47	3.991000	25104
48	2.119000	15466
49*	18.36100	43091
50	2.910000	21258

(no path = *)

Repeated Forward A*	Time taken	Expanded Cells
1	0.919000	15723
2	1.190000	18962
3	1.318000	21181
4	1.051000	18305
5	1.994000	42972
6	No path-6.808	No path-32093
7	0.992000	16563
8	No path-7.4	No path-28562
9	No path-6.801	No path-31060
10	1.488000	28909
11	No path-6.448	No path-30138
12	1.207000	21385
13	1.143000	19296
14	1.053000	17106
15	0.944000	16388
16	No path-7.609	No path-40641
17	1.970000	38787
18	1.202000	18997
19	3.216000	62521
20	No path-8.242	No path-29541
21	1.002000	15822

22	1083000	20210
23	No path-6.785	No path-26781
24	1.768000	34325
25	No path-0.022	No path-153
26	No path-6.538	No path-25998
27	1.162000	20333
28	0.892000	14554
29	No path-7.213	No path-34281
30	No path-8.432	No path-40071
31	No path-6.91	No path-27629
32	No path-6.677	No path-28598
33	0.999000	16275
34	0.972000	15094
35	No path-7.072	No path-28885
36	1.003000	16913
37	1.369000	20786
38	1.111000	18281
39	1.454000	22541
40	0.982000	16272
41	No path-13.146	No path-67116
42	1.035000	17238
43	No path-7.039	No path-29744
44	1.174000	17784
45	No path-0.022	No path-153
46	1.111000	18830
47	1.712000	32650
48	1.098000	18039
49	No path-21.324	No path-46715
50	1.407000	21640