Assignment 1 - Trajectory Replanning

Homework 1 Solution Report

Team Members:
YUHUA NI (yn85)
QIANG LI (ql119)

Questions:

Part 0:

Our maze generator was obtained from wikipedia. The wikipedia maze generator utilized a random depth-first search algorithm. The python code gave us a boolean numpy array where each element of True indicates a blocked cell. We transform this boolean array into an integer array which allows us to place more varied indicators for each cell in the grid.

Part 1:

a) The first shortest presumed unblocked path that the agent finds is the path directly to the east. This is because it does not yet know where the blocked cells are. Because the cell in the east has the smaller f-value (:= g-value + h-value) than the cell in the north. In Figure 8, we can see that g(east) equals to g(north), but h(east) equals to 2 and h(north) equals to 4, which is larger than h(east). Therefore, A* search will go first to the cell in the east which has the smallest f-value.

b) A* will keeps expanding unexplored cells and put them into the closed set so that it will never touch that again. By following this fact, the target cell will must be explored if there is a path from the initial cell to this target. On the situation of the unreachable target, the frontier set will eventually be empty so that there is no cells can be explored any more and the program will terminate. The detailed proofs following the below two lemmas.
Lemma 1: A path that is taken out of the open set is not placed into the open set again at a later step.
Proof: (using logical deduction)
If P = {v0, v1, ..., vk} is a path that is taken out of the open set, then P = {v0, v1, ..., vk} is not placed into the open set at a later step. Assume that P = {v0, v1, ..., vk} is taken out of the open set. Then, P must be placed into the queue at an earlier step. Then, v0 must be in the closed set at this step. Then, P cannot be placed into the open set again at a later step, since v0 is in the closed set.
Lemma 2: If a vertex v is reachable from S, then v is placed into the closed set after a finite number of steps.
Proof: (by contradiction)
Assume v is reachable from start state S, but it is never placed on the closed set. Since v is reachable from S, there exists a path that is of the form {v0, v1, ..., vk}, where v0 = v and vk = S. Let vi be the first state (starting from vk) in the chain that is never added to the closed set. [statement #1] Then, (vi+1, vi) is in the path. Since vi+1 was in the closed set, the open set included a path {vi+1, ..., vk}, where vk = S. This path must have been popped from the open set, since there are only finitely many different partial paths and no path is added twice (by Lemma 1) and vi was not in the visited list. Since it is popped from the open set, then {vi+1, vi, ..., vk} must be placed into the open set and vi placed into the closed set, which contradict the [statement #1]
We know that there exists path from state s to state t, then by following the above two lemmas we can say that t will be placed in the closed set in finite steps. This is exactly when the A* algorithm terminates.

The A* search algorithm puts each cell into the open set exactly once when the cells are discovered but have not been visited. Then, the algorithm puts these cells into the closed set when they have been visited. Based on the proof following lemma 1 and 2, we know that each cell will be visited at most once. In the worst case, we need to visit every unblocked cell and then put them into the close set for each run of A*. Therefore, the number of cell expansions or moves of one A* search is bounded above by the number of unblocked cells (assume this number is N) in the grid.

Therefore, it follows that the number of moves for Repeated A* is the N * (number of repetitions). The number of repetitions is at most N (if A* only advances the agent one space each time). Thus, the number of moves is bounded by N*N or N^2.

Part 2:

The implementation part:

The implementation for this question had the starting cell be (1,1) in the grid and the target cell be (101,101) in the grid. The algorithm was tested on multiple mazes generated using the random depth-first search maze generator.

Observation:

Favoring larger g-values provided a much faster runtime. The number of cells expanded for A* favoring small g-values was generally larger than A* favoring small g.

The explanation part:

While they may both obtain similar paths to the target, Repeated Forward A* using small g-values will expand more broadly for each run of A*. By favoring small g-values, the algorithm will make sure to give equal priority to all possible paths to the target. This causes a lot of resources to be expended on paths that do not end up being used unless the information is saved somehow. Using large g-values favors quicker and more focused expansion by valuing cells which are further from the initial start cell more so than cells that are closer to the starting cell. This reduces the number of cells expanded because the algorithm will only look into one shortest presumed unblocked path for each run of A*.

In figure 9, every cells have the same f-value, therefore we will have tie in every step. When we use the larger g-value to break the tie, we can make sure we are one step away from the initial point and therefore one step closer to the end point. However, we have to touch all the cells if we choose to use the smaller g-value because each time we need to expand all the cells in the open set (frontier set) whose g-value are the smallest.

Part 3:

Implementation:

The implementation for this question had the starting cell be (1,1) in the grid and the target cell be (101,101) in the grid. The algorithm was tested on multiple mazes generated using the random depth-first search maze generator.

Observation:

From the implementation, it can be seen that the number of expanded cells for Forward Repeated A* is generally less than the number of expanded cells for Backward Repeated A*. In cases where the solution was very simple, straightforward, and symmetric from both the start and the target, the number of expanded cells were similar if not the same.

Explanation:

Repeated Backwards A* is slower than Repeated Forwards A* because it does not take advantage of the heuristics. In Backwards A*, the heuristics for the cells near the goal are misleading because it does not take into consideration possible blocked cells near the start. This can cause Backwards A* place more emphasis on a path that is actually less favorable than another one. Thus, Repeated Forwards A* is generally faster than Repeated Backwards A* due to having more informed heuristics when calculating the paths.

Part 4:

a) Because we can only move vertically or horizontally, therefore manhattan distance never overestimate the cost to reach the goal state, and therefore it is admissible. Therefore, we have h(s) - h(s') <= g(s') - g(s) (the heuristic segment is always no larger than the actual segment cost), if s' is a successor of s. We also have that g(s') - g(s) = c(s, a, s'), where a is any action that the agent can take to move from s to s'. Therefore, h(s) - h(s') <= g(s') - g(s) = c(s, a, s'). So, we have h(s) <= c(s, a, s') + h(s'). Consistent!

b) Proof by induction: The initial heuristics are provided by the user and thus consistent. It thus holds that h(goal_state) = 0. This continues to hold since the goal state is not expanded and its heuristic thus not updated. It also holds that h(s) <= h(succ(s, a)) + c(s, a) for all non-goal states s and actions a that can be executed in them. Assume some action costs increase. Let c denote the action costs before all increases and c' denote the action costs after all increases. Then, h(s) <= h(succ(s, a)) + c(s, a) <= h(succ(s, a)) + c'(s, a). Thus, the heuristics remain consistent. Now assume that the heuristics are updated. Let h denote the heuristics before all updates and h' denote the heuristics after all updates. Let gd(s) denote the goal distance of state s and $f* = gd(start\_state)$ denote the cost of the cost-minimal path found after an A* search. We distinguish three cases:

- First, both s and succ(s, a) were expanded, which implies that,
  h'(s) = f* - g(s)
  h'(succ(s, a)) = f* - g(succ(s, a))
  Also, g(succ(s, a)) <= g(s) + c(s, a),
  which is the same as g(succ(s, a)) - c(s, a) <= g(s)
  Thus, h'(s) = f* - g(s)
  <= f* - (g(succ(s, a)) - c(s, a))
  = f* - g(succ(s, a)) + c(s, a)
  = h'(succ(s, a)) + c(s, a)
- Second, s was expanded, succ(s, a) was not. Similarly, we have:
  h'(s) = f* - g(s)
  h'(succ(s, a)) = h(succ(s, a))
  Also, g(succ(s, a)) < g(s) + c(s, a)
  f* <= f(succ(s, a)), since succ(s, a) was generated but not expended.

Thus, h'(s) = f* - g(s)
<= f(succ(s, a)) - g(s)
= g(succ(s, a)) + h(succ(s, a)) - g(s)
= g(succ(s, a)) + h'(succ(s, a)) - g(s)
<= g(succ(s, a)) + h'(succ(s, a)) - g(succ(s, a)) + c(s, a)
= h'(succ(s, a)) + c(s, a)

- Third, s was not expanded. We have:
  h'(s) = h(s)
  Also, h(succ(s, a)) <= h'(succ(s, a)) since the heuristics of the same state are monotonically nondecreasing over time.
  Thus, h'(s) = h(s) <= h(succ(s, a)) + c(s, a)
  <= h'(succ(s, a)) + c(s, a)

In sum, h'(s) <= h'(succ(s, a)) + c(s, a) in all three cases and the heuristics thus remain consistent.

c) Consistent heuristics are also admissible. Prove that by induction. Assume Ss, S1, S2, …, Sg is the actual path that the agent took from start state (Ss) to goal state (Sg). Denote c(Sn, Sm) the cost to move from state Sn to Sm. Because the h-value of each state is consistent, therefore we have:
h(Ss) <= c(Ss, S1) + h(S1)
h(S1) <= c(S1, S2) + h(S2)
h(S2) <= c(S2, S3) + h(S3)
……
h(Sg-1) <= c(Sg-1, Sg) + h(Sg)
Sum of the above inequalities, we get:
h(Ss) <= c(Ss, S1) + c(S1, S2) + … + c(Sg-1, Sg) + h(Sg) = g(Sg) + h(Sg)
Because h(Sg) = 0, therefore we have:
h(Ss) <= g(Sg), which means the heuristic value never overestimate the actual cost, therefore is admissible.

Part 5:

Implementation:

The implementation for this question had the starting cell be (1,1) in the grid and the target cell be (101,101) in the grid. The algorithm was tested on multiple mazes generated using the random depth-first search maze generator.

Observation:

From our observations, Adaptive A* generally had a faster runtime than Repeated Forward A*. Adaptive A* had fewer expanded cell counts on average for each run of a randomly generated maze. In cases where the solution was very simple and straightforward, the number of expanded cells were similar if not the same. In some cases, Repeated Forward A* was faster.

Explanation:

Repeated Forward A* has a vulnerability in that it can fall into a local minimum while searching for the optimal path. This is shown in the examples of Fig. 7 and Fig. 5. The update heuristics of Adaptive A* makes the h-value heuristic more informed than it originally was by updating and

keeping it stored for past expanded cells. In terms of the example given, Adaptive * updates the local minimum near the starting point which has lower heuristics, due to its close proximity to the goal, and provides newer heuristics that inform the algorithm that area actually is further from the goal.

Part 6:
Our current implementation carries information of all cells in 2D array of cells. Each cell contains the following attributes: g-value (4 bytes), h-value (4 bytes), f-value (4 bytes), h_{new}-value (4 bytes), x and y-value for position (8 bytes), and a pointer to the parent cell (4 bytes). This is a total of 28 bytes per cell. Our implementation also contains separate arrays holding with expanded cells represented by an int (4 bytes) and blocked cells represented by an int (4 bytes). This adds an additional 8 bytes. Thus for each cell there is about a total of 36 bytes used and each node in the grid is filled by a cell, initially initialized to its default values.

To reduce memory usage, we can first apply the optimization noted in the question itself. By using only two bits (00, 01, 10, 11), we can use the bits to point to which direction the parent node is (up, down, left, right). In addition, we can add on to the previous idea by having three more bits holding information on expanded, blocked cells, and nodes that the agent has passed over for a total of 5 bits per node in the grid or round up to 1 byte.

For cells nodes, we can dynamically allocate additional memory for nodes that get added to the open list and become “cells.” We can calculate h-values and h_{new}-values on the fly instead of storing them as they do not serve a purpose other than to calculate f-values. We can stop storing each cell and only have them be stored in the heap when added as neighbors of an expanded cell. This reduces memory usage per cell node down to 17 bytes per cell node, 8 to keep position, 8 for f-values and g-values, and 1 to maintain attributes in bits.

Now, given that at any given time, only at most 2001 nodes can be expanded by any single run of A* and added to the open list if we make optimize the backtracking process. This can be accomplished by optimizing memory usage during backtracking so that previously expanded nodes need not create a new cell node and should only be required to change their bits to point to the new parent node when backtracking. Each node has 3 new neighbors, with one being the eventual child, thus two will stay in the open list. Thus, at most only about 4004 nodes will be cell nodes. The rest will be empty nodes holding only the single byte of information.

- Final memory counts does not take into consideration overhead for python objects.

Calculations:
Cell nodes memory usage = 4004 * 17 = 68068 bytes
Empty node memory usage = (1002001 - 4004) * 1 = 997997 bytes
4 MB = 4,194,304 Bytes

Total = 68068 + 997997 = 1066065 bytes < 4194304 bytes