Assignment 1 - Search

[redacted]

October 1 2024

Part 1

Part A The first move of the agent in Figure 7 is to the east rather than the north. This is because it is calculating the shortest path based on what it currently knows. At this starting point, it does not know what coordinates on the grid are blocked and unblocked. Based on its starting point at E2, it evaluates the shortest path to the target to be from E2 $\rightarrow$ E3 $\rightarrow$ E4 $\rightarrow$ E5. It does not yet know that E3 and E4 are blocked. Based on the estimated heuristic cost from the node we are looking at to the target and the movement cost from E2 to the corresponding square we are looking at,

$$f(E3) = 1 + 2 = 3, \quad f(D2) = 1 + 4 = 5, \quad f(E1) = 1 + 4 = 5.$$

Based on these computations performed using the A* algorithm for the agent, it will choose to go east, towards E3, because it assumes that this will lead to the shortest path to the target, not knowing that it will encounter blocked squares in the process of reaching it.

As stated in the project description, the agent initially does not know which cells are blocked.

Hence, the unknowing of whether a cell is blocked and the results of these starting computations for the f-score is why the agent moves to the east rather than the north.

Part 1B

The agent in a finite gridworld either reaches the target or discovers that it is impossible in finite time. The A* algorithm utilizes a cost function to determine the shortest path, $f(n)$, given the estimated heuristic costs from the current node to the target and the cost from the node we are examining to the starting node:

$$f(n) = g(n) + h(n)$$

The algorithm then uses priority to determine whether this is the shortest current path to attempt to travel.

The agent can remember the blockage status of its north, south, east, and west cells for future shortest-path determinations. This enhances the effectiveness and optimality of the A* algorithm. The algorithm, and consequently the agent, employs the freespace assumption, which ensures that the path it is on leads from the current cell to the target, assuming that the cells along the path are unblocked and that this is the shortest such path.

However, if the agent encounters blocked cells along its path, it remembers this information for future path-searching. If the agent hits a blocked cell, it uses the information it already knows about blocked cells to determine another shortest possible path using the cost function and its updated knowledge. This process essentially forms a cycle with two possible outcomes: either the shortest path to the target is determined, or the agent concludes that no such path to the target is possible. Essentially, the agent repeatedly performs the A* algorithm to keep track of paths, if they exist.

Moreover, since this is a finite grid, there are only a finite number of paths that the agent can explore, which means the agent will either determine that it is impossible to find such a path to the target or it will determine the shortest possible path.

The agent follows a systematic exploration strategy by utilizing an open and closed list to keep track of neighboring unblocked nodes that have been visited or have not been visited yet. In its worst-case form, the agent will explore all unblocked nodes before determining whether a path exists or does not exist.

Therefore, due to the reasons described, the agent in a finite gridworld will either reach the target or discover that it is impossible in finite time.

The number of moves the agent makes until it reaches the target or discovers that this is impossible is bounded from above by the square of the number of unblocked cells. This upper bound represents the worst-case situation for the A* traversal. In the case of A*, based on its grid traversal to find the shortest path to the target, it would remember what it observed in terms of blocked cells in the nodes and neighbor nodes that it traverses. Hence, it is highly unlikely for it to ever hit the upper bound of number unblocked cells squared. However, this is the bound because, assuming there is x unblocked cells, in worst case, to find the shortest path, it would traverse the node's neighboring unblocked cells that have already been visited. Hence, this proves that the upper bound until the agent either reaches the target or finds it impossible is number of unblocked

cells squared.

Part 2

In this part, the effect of different tie breaking methods is examined in the Repeated Forward A* algorithm. The two strategies when coming across a situation that requires a tie breaker are: expand in favor of cells with smaller g-values or in favor of cells with larger g-values. Through the algorithm in the code provided with the assignment, it is apparent that nodes with larger g-values are faster in terms of runtime. One observation that can be made is that the use of larger g-values to break ties leads to less nodes being expanded than using smaller g-values. When the algorithm expands the larger g-value nodes in tie breaking situations, it is essentially looking at paths that have higher cost, which could lead the search away from dead ends and focus on areas that are more likely to lead to the goal, leading to a more directed search. On the other hand, smaller g-values leads to the algorithm exploring less promising paths and increasing unnecessary expansions.

In addition, the idea that less cells are being expanded also complements the fact that it significantly decreases runtime. The decrease in runtime is due to the less expanded nodes through crossing off the unnecessary expansions. This will help the algorithm run to be more optimally which is important to the performance in large grid environments such as in the experiments conducted. The algorithm focusing on paths that are more likely to lead to the target will improve speed overall.

Part 3

This section focuses on the difference of performances between Repeated Forward A* (RFA*) and Repeated Backward A* (RBA*). Specifically, the number of expanded cells were observed to examine the runtime of both algorithms. Between the two algorithms, the key difference lies in the direction the search algorithm is conducted. In RFA*, the starting point is the initial start node and continues to proceed toward the goal. In RBA*, the starting point is the goal node and the search moves towards the initial start node. In both search algorithms, the exploration considers the same function f(n) = g(n) + h(n) where g(n) is the path cost from what has already been traveled and h(n) is the heuristic estimate of the future.

In Figures 1 and 2, the number of expanded cells and run time are illustrated for the two algorithms. RFA* search has an estimated mean of 34,634 expanded cells as opposed to RBA* which expands 37,186 cells. It should also be noted that RBA* search runs slightly faster than the RBA* in this specific generated environment. In addition, the generated maze paths in Figure 3 and 4 illustrates the differences of how each algorithm chooses to expand. RBF* seems to access a more directed path than RFA* which appears to be more sparse in its

exploration.

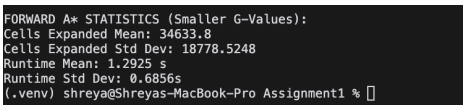

Figure 1: Data and Observations from Repeated Forward A* (Smaller G* Values) Search. The first line under the line shows the mean of the number of cells expanded with the standard deviation underneath. The following lines after illustrate the runtime average and its standard deviation.

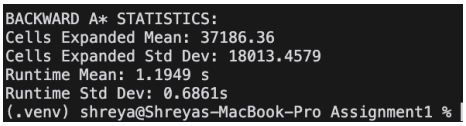

Figure 2: Data and Observations from Repeated Backward A* Search. The first line under the line shows the mean of the number of cells expanded with the standard deviation underneath. The following lines after illustrate the runtime average and its standard deviation.

The distinctions of runtime and number of expanded cells could be due to the layout of the maze and bias of one algorithm over the other by the heuristic function in the search problem in this specific generated state space. For example, if the area around the goal is in a more narrow and constrained space, it might working in favor of the RBA* since it has less nodes to expand that stray from the path. Since there is less of a sparse exploration, it may also mean that RBA* is conducting faster node processing and effectively running faster. However, the number of expanded cells might be more due to the node selection in the algorithm being less efficient and how the maze is structured.

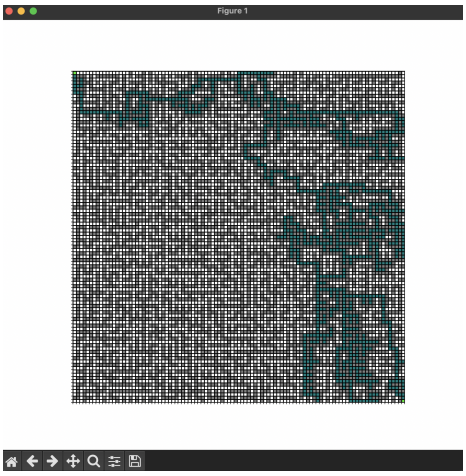

Figure 3: Path in Generated Maze with Repeated Forward A* search

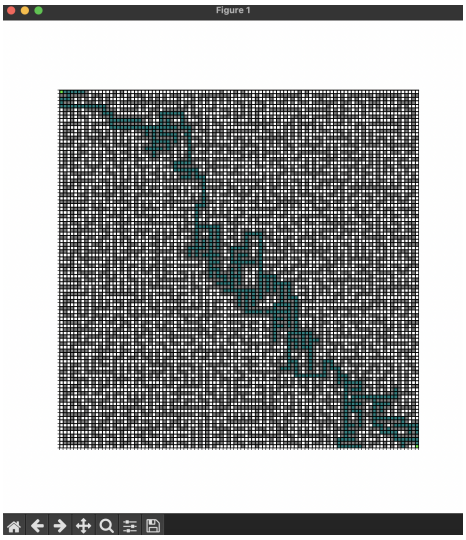

Figure 4: Path in Generated Maze with Repeated Backward A* search

Part 4

Part A

The Manhattan Distance is consistent in gridworlds in which the agent can move only in the four main compass directions. To prove this is the case, we will utilize the triangle inequality, the definition of a consistent heuristic

function, and what we know about the movements of the agent in a coordinate plane.
There are four possible movements that the agent can make: up, down, left, right. Assuming the starting point is (x0, y0), each of the directions is denoted as follows in terms of coordinates:
up: (x0, y0 + 1)
down: (x0, y0 - 1)
left: (x0 - 1, y0)
right: (x0 + 1, y0)
Each of these coordinate computations represent the changes to cost it will take to move from the starting coordinate through one of the four directions to the target. As can be seen, the cost of moving down or left has a -1 to the total cost heuristic function, whereas the cost of moving up or right has a +1 to the total cost heuristic function based off a coordinate plane.
The triangle inequality is as follows: h(n) $\leq$ c(n,a,n') + h(n')
The cost function is depicting the cost it takes to get from the original node n to the node n' by performing an action denoted by a. In our case, the actions are either the +1 or -1 to the cost based on the direction it moves.
To prove that the Manhattan distance is consistent, we will do a proof as follows:

Proof that shows moving up or right still keeps Manhattan Distance consistent:
Let us assume that the target node is located at a position below the location of our starting node.
Using the coordinate depictions of the movements from above:

$$h(n) \leq c(n, a, n') + h(n')$$
$$h(n) \leq 1 + h(n') + 1$$
$$h(n) \leq 2 + h(n')$$

This above statement is true because moving up or right increases the cost by 1 but the distance from the new node to the target also increases, meaning we add a total of 2 to h(n').
Despite the up/right movement, h(n) $\leq$ 2 + h(n'), meaning this shows it is consistent in gridworld for these specific movements.

Now, here is the proof that shows moving down or left still keeps Manhattan Distance consistent:
Using the coordinate depictions of the movements and the assumption made from above, we have that:

$$h(n) \leq c(n, a, n') + h(n')$$
$$h(n) \leq -1 + h(n') + 1$$

$$h(n) \leq h(n')$$

The cost of moving down/left has a cost of 0 because moving down or left leads to a -1 in cost. We are assuming that moving left or down will bring us closer to the target node. Hence, putting together the inequality, we have that moving down/left also leads to a consistent Manhattan distance in gridworld. If we were to assume that moving left/down were to take us further away from the target, we would increment the cost by 1. However, even if we did that, the inequality would still prove true.
Hence, based on the above proofs, the Manhattan distance is consistent in gridworld when moving up, down, left, or right.

Part B
We know that given the definition of a consistent heuristic function:

$$h(n) \leq c(n, a, n') + h(n')$$

Assuming that at the initial point the h values are consistent as given in the problem, we denote this as $h_{\text{new}}(n)$.

$$h_{\text{new}}(n) \leq h_{\text{new}}(n') + c'(n, n')$$

$$c'(n, n') \geq c(n, n')$$

This c' cost function above represents the new function with increased action cost, which would be greater than the previous cost function denoted by the c.

Hence:

$$h_{\text{new}}(n) \leq h_{\text{new}}(n') + c(n, n') \leq h_{\text{new}}(n') + c'(n, n')$$

The Adaptive A* algorithm accounts for the increases in action costs and updates the heuristic function accordingly. When the cost increases, it remains consistent.
This satisfies the criteria for a consistent heuristic, as identified by $h_{\text{new}}(n)$ being less than or equal to the remaining heuristic and cost expressions for the updated action costs. As shown in the proof, with the assumption that initially the h-values are consistent, we proved that even if action costs increase, as represented by the updated action cost functions, the consistency still holds.

Part 5

Adaptive A* takes into account the drawbacks of Repeated Forward A* and attempts to improve upon it. Like repeated Forward A*, the goal is still to repeatedly find the shortest path in the state space. However the difference lies in the manner in which the Adaptive A* utilizes information from the previous searches and changes the heuristic values in the nodes for future searches.

Essentially, the algorithm is learning from what it previously explored and recomputing the h-values for states expanded during the search. From the image generation of the two searches, Adaptive A* is shown to be more focused in the later stages (Figure 6), where it is getting closer to the goal. This is in contrast to Repeated Forward A*, where the algorithm does not learn from previous searches and therefore the search is less focused and running into more dead ends (Figure 3).

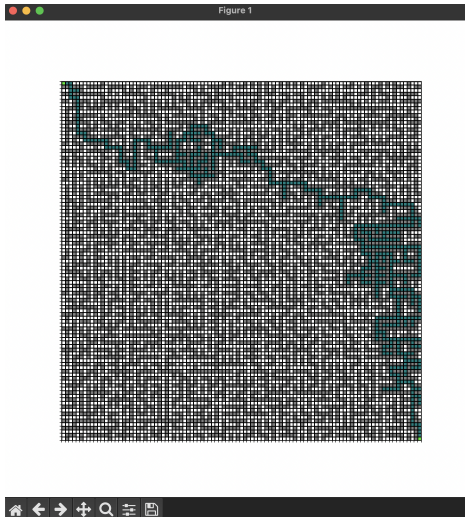

Figure 5: Data and Observations from Adaptive A* Search. The first line under the line shows the mean of the number of cells expanded with the standard deviation underneath. The following lines after illustrate the runtime average and its standard deviation.

From the Figures 1 and 5, the number of expanded cells for RFA* was 34,633 while it was 29,969 cells for Adaptive A*. The runtime for the Adaptive A* was 0.5841 seconds compared to 1.2925 seconds for RFA*. From this data, it shows that Adaptive A* performs much more efficiently and faster than RFA* in terms of number of cells expanded and runtime.

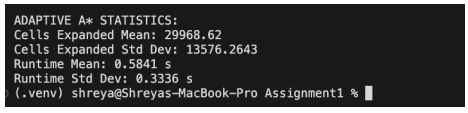

Figure 6: Path in Generated Maze with Adaptive A* Search

The data illustrates the efficiency of the Adaptive A* algorithm, because the search process is more focused. Since the heuristic gets updated with the previous forward A* searches, the h-value can better guide the search and expand

less nodes that are not needed. This is as opposed to the RFA* where the same heuristic value is used so it is not using information from previous searches.