Intro to AI Homework 1

Roshan Patel 200000858 Abhay Dhiman 192009888

Devin Shin 199003539

February 25th 2024

1 Introduction

In this homework assignment we explored heuristic path finding. From lecture we learned how programs can use heuristics to make educated guesses to inform their decision making. From there we went in-depth on implementing heuristics in path-finding, namely through the A* search algorithm. This algorithm is used extensively in video game development for path-finding. From there we were tasked with studying and implementing the Repeated A* Search on our own.

First, we begin with developing a randomized grid-world. Then we explored why heuristics and the algorithm work the way they do. From there we studied an important assumption/decision in implementing the Repeated A* Search algorithm. Next, we actually implemented the algorithm, its forward and backward variations. Then we explored the Adaptive A* Search which improves it efficiency and implemented that. Finally we end with a discussion of the statistical significance of the difference in cells visited(a metric of efficiency) between the Repeated Forward A* and the Repeated Backward A* algorithms.

2 Grid-world Generation

To generate random grid-worlds, we initialized an array of 0s, picked a random coordinate to start at and essentially performed a depth first search to mark cells/coordinates as blocked or unblocked. Blocked was signified as a 1 in the array and had a 30% of being chosen. Accordingly, unblocked was signified as a 0 in the array and had a 70% of being chosen. Once all the neighbors of the current coordinate were visited, the stack would be popped and the second latest cell would continue to have DFS run on it. This is an efficient way to generate random grid-worlds each time. It's only input is the size of the array, which generates a (size x size) square grid-world. Figure 1 is an example 101 x 101 grid-world.

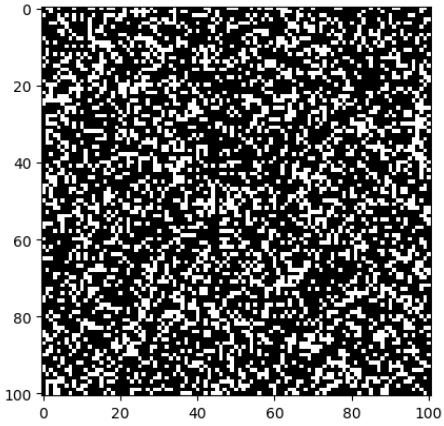

Figure 1: 101 x 101 Grid-world

3 Understanding the Methods

3.1

To understand how heuristics works we take a look at Figure 2. A is the current state of the Agent and T is its target cell. The Agent starts by not knowing which cells are blocked so it assumes at first that none of the cells are blocked. With this knowledge to find the shortest path, it sees which cell to go to next. The cell above and the cell to the left both have f-costs of 5, where the cell to it's right has a f-cost of 3. All three cells have the same g-costs, but the right cell has a lower h-cost because it is closer to the Target cell then the other neighboring cells. Thus because the right cell has the lowest f-value the Agent moves there. It repeats the process again and finds that the cell to the right is closer than its neighbors. It does it one more time to find that the cell to the right is the Target. Heuristics, as displayed by the h-cost, essentially finds the shortest path length from each cell to the target ignoring which cells are blocked or not.

3.2

Due to the fact that it does not know which cells are blocked and that the Agent initially assumes that none of the cells are blocked, the algorithm will find a new path at every state in which it gains new knowledge about blocked cells. In the case that the goal state is blocked from being reached, the algorithm will essentially try many or all different paths and traverse every state possible in its effort to find a path to the goal state. In the worst case scenario, it will try every path possible from every unblocked state, which would be N^2, with N being the number of unlocked cells.

Once it has gone through this has realized that no path exists to the goal it will know that the goal is impossible to reach. This proves that if the goal is reachable no matter how complex the maze is, the algorithm will find it. If the goal is not reachable, the algorithm will realize this after traversing every state, with at worst N^2 traversals. Knowing this, it can be proved that the number of moves to reach the goal or to realize that the goal is impossible to reach is upper bounded by the number of unblocked squares squared because in the worst possible case of the goal being impossible to reach the Agent would essentially try every path from every unblocked cell.

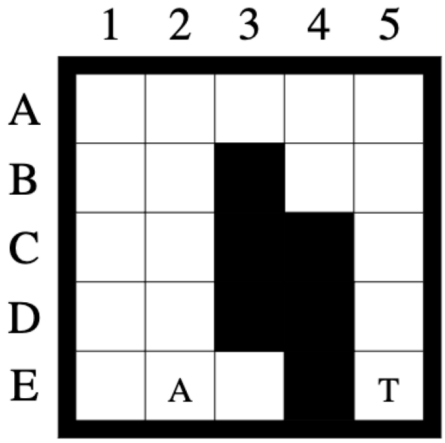

Figure 2: Example Search Problem

4 The Effects of Ties

A very important part of the A* Search algorithm is the influence g-cost has on the performance of the search. Essentially, when the algorithm picks cells in the open list with the minimum f-costs, there needs to be procedure to break the tie of which cell should be expanded next. To break the tie, we compare the g-costs of the cells with the same minimum f-costs. However, we have two ways to go about this: choose the cell(s) with the greatest g-cost or the cell(s) with the least g-cost. (If there are multiple cells in each case of the extremes, then we can randomly choose a cell).

To understand which case performs better, we run an experiment on the grid-world presented in Figure 3. We run an A* Search where the greatest g-cost is chosen for tie-breaking and we find that the number of expanded cells is 8. This is really good considering 8 is the length of the shortest path that the Agent can take. On the other hand, running an A* Search where the least g-cost is chosen for tie-breaking we find the number of expanded cells is 23. This is un-

derstandable because when you chose the least g-cost when tie breaking you are essentially expanding the cells that are closest to the Start position and farthest from the Target. Thus, you end up expanding cells that are not the most optimal to expand to get to the Target and therefore you expand more cells than needed.

To further investigate, we generate 50 101x101 grid worlds and run each implementation on them. In doing so we found that the average total number of cells expanded is a difference of 10-fold. The greatest g-cost implementation averaged 2045.7 cells expanded per grid-world, while the least g-cost implementation averaged 19058.9 cells expanded per grid-world. This confirms our understanding and proves that choosing the greatest g-cost is roughly 10x more efficient than choosing the least g-cost for tie-breaking.

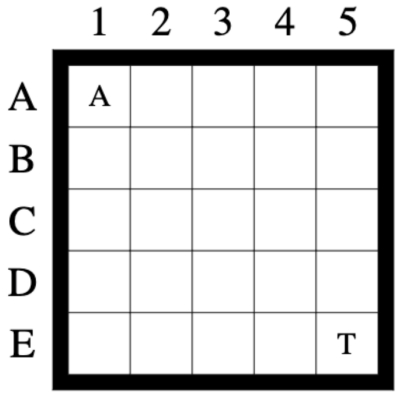

Figure 3: Tie-breaking Search Problem

5 Forward vs Backward

Now that we understood the heuristics and the importance of the g-cost in the A* search algorithm, we implement it. We used Repeated A* search as our approach which meant that the Agent would find the shortest path through the A* search and then begin traveling through that path. If it encounters a blocked cell in it's path, it would record the coordinates of the blocked cell and run A* search again now knowing about the blocked cell. It would continue performing A* search and learning the position of blocked cells until it finds a path to the Target that doesn't have a blocked cell in it. From implementing this method we achieve the Repeated Forward A* search. However, there is another approach to this called Repeated Backward A*. In Repeated Backward A*, we find the shortest path from the Target to the Agent (essentially switching Start and Target). When a path is found, the Agent traverses until it runs into a block, where another path from the Target to the current state of the Agent is

found.

Implementing both we can compare their performance through the number of cells that are expanded. To test this we experiment on a simple 5 x 5 grid-world (Figure 4) . We pick the Start cell of the Agent as (0,4) and the Target cell as (0,1). We perform both forward and backward implementations on this scenario and we get the steps shown in Figure 4. As we can see they both have

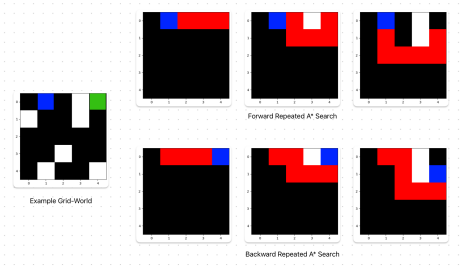

Figure 4: Forward vs Backward Simple visualization

the same number of steps and they find the same paths at each step. However, a closer look reveals that the Forward implementation expanded only 15 cells in total, whereas the Backward implementation expanded 30 cells total. This shows that Repeated Forward A* is much better in performance than Repeated Backward A*. Applying this to a 101 x 101 size grid-world (Figure 5), we set the

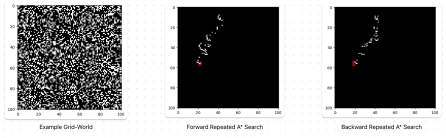

Figure 5: Forward vs Backward Complex visualization

start cell of the Agent to (11,41) and the target cell to (58,2). We perform the Forward implementation and get the total cells expanded is 2114. We perform the Backward implementation and get the total cells expanded is 135873. This shows a huge gap in the performance of these two implementations. We ran the same experiment on 10 different mazes each with a randomized starting and end point and got the average number of total cells expanded by Repeated Forward A* to be 2644 and the average number of total cells expanded by Repeated Backward A* to be 179757.9. We believe that the reason Repeated Backwards A* performs much worse is because you are unnecessarily expanding the same cells that are near the Target only for the Target to not move. It essentially creates an inefficient redundancy of which cells are expanded.

6 Heuristics in the Adaptive A* - Part 1

The Manhattan distance heuristic is consistent and able to remain consistent because of the nature of the environment.
The heuristics are based on a tangible aspect of the environment, this being the grid. The heuristic is calculated using vertical and horizontal movements, and coincidentally our Agent is actually only capable of such movements. This means the heuristic is not just an estimate. In this case it is literally the distance to the goal, regardless of the (presumed blockage-free path) the agent takes. This is why the heuristic is consistent.

Now, a question that naturally comes to mind is that when action costs increase (in other words when our agent runs into a wall), how can we say that the heuristics remain consistent? We've already established their admissibility and consistency due to the nature of the grid environment and our agent's movement ability, but how can we know this holds? The reason we can say that h values remain consistent even if our Agent runs into a wall is because the Agent, or rather Adaptive A*, recalculates the heuristics every time it runs into a wall on its presumed-blockage-free path. So, by updating them based on observed costs every time the agent runs into a wall, the algorithm prevents any overestimation or inaccuracy in the h values.

7 Heuristics in the Adaptive A* - Part 2

After seeing how much better Repeated Forward A* search is than Repeated Backward A* search, it might seem that the Repeated Forward A* is the best performing path finding algorithm. However, that is not the case when you utilize heuristics, namely the h-costs, to their full potential. In Adaptive A* search, you are essentially using the length of the path and the g-costs of the expanded cells of the previous A* search to calculate new h-costs for the expanded cells. Having the knowledge of which cells are blocked, these new h-costs give a more accurate estimation of the distance to the Target cell. Using these new h-costs to calculate the new f-costs when the A* search is run again, we find a new path that expands fewer cells as opposed to simply running Repeated Forward A*. This happens because the more accurate h-costs that the Agent learns from previous searches, let the Agent know which cells are more optimal to expand rather than letting the Manhattan distance make the Agent believe the Target is a lot closer than it seems by ignoring the blocked cells that it has learned.

To see how these two implementations compare we generate a 101x101 grid-world and choose the Start cell as (44,1) and the Target cell as (10,96), then compare the total number of cells expanded by each implementation. After performing each implementation on the 101x101 grid-world shown in Test 1 in Figure 6 with the aforementioned Start and Target cells, we get the total number cells expanded by the regular Repeated Forward A* to be 5450 and

the total number of cells expanded by Adaptive A* Search to be 4270. From this we realize that the Adaptive A* approach performs better than the regular Repeated Forward A* search approach. This makes sense because as mentioned before, the Adaptive A* approach learns from previous searches to get more accurate heuristics which lets the Agent know the optimal cells that are worth expanding. This optimization leads to less unnecessary cells being expanded thus improving the efficiency and performance of the algorithm. To test our observation further, we performed this experiment on 50 101x101 grid-worlds each with a randomized Start and Target cell. The average total number of cells expanded by the regular Repeated Forward A* search was 2413.8, while the average total number of cells expanded by the Adaptive A* search was 2293.66. These averages confirm our observations and our understanding of why Adaptive A* performs better. Further experiments that also observe Adaptive A* performing better are shown in Figure 6.

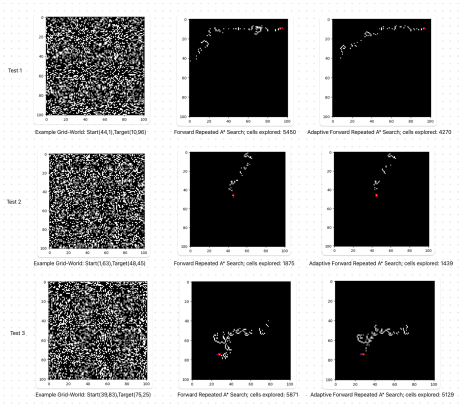

Figure 6: Repeated Forward A* vs Adaptive A*

8 Statistical Significance

For the description of how a statistical hypothesis test could be performed, let us look again at the comparison between the Repeated Forward A* and the Repeated Backward A* algorithms. The goal of doing such a test is to determine whether or not the two algorithms indeed have a significant systematic difference in performance.

First, we formulate our null and alternative hypotheses. The null hypothesis will be that there is no significant performance difference between Repeated Forward A* and Repeated Backward A*. The alternative hypothesis will be the opposite, that there is in fact a significant performance difference. Performance will be measured by how many cells each algorithm moves into the closed list and thereby “expands”, or “visits”.

Next, we would collect data. We would run both algorithms at least 30 times and record the number of cells that are visited by each algorithm in each instance. The reason for 30 times is that this allows the central limit theorem to kick in, which will be useful later. We would also choose a significance, or alpha level. This will be 0.05, which signifies a 95% confidence interval. This alpha level is a good starting point and is used in most statistical hypothesis tests, according to one of our group members' statistics professors.

Then, we choose a statistical test, the T test. The T test is chosen because the T test can be used to compare the means of 2 different data sets. The 2 different data sets in this scenario are the numbers of cells visited in the same size grid by the 2 different algorithms, Repeated Forward A* and Repeated Backward A*.

We calculate the t value and degrees of freedom. We use the degrees of freedom and significance level(α) to determine the critical value using a t-distribution table. We also calculate the p value.

Now, the t value can be compared to the critical value or the p value can be compared to the α value. Since this is a hypothetical, we can go the extra mile and do both. If t is greater than the critical value or $p < \alpha$, then we reject the null hypothesis. This means there is in fact a significant difference between the 2 means, so one of the algorithms visits a larger amount of cells than the other, in a statistically significant way.