Intro to AI Assignment 1 - Fast Trajectory Planning

Rajeev Atla, Jasmin Badyal, Dhvani Patel

April 15, 2025

§0 Part 0 - Setting Up the Environment

§1 Part 1 - Understanding Methods

a. We established the agent can only observe the four squares immediately adjacent to itself in the gridworld, and any squares we have not encountered yet are assumed to be empty until it is observed otherwise. Therefore in Figure 8, the agent can only observe its neighbors 1E, 2D, and 3E, and has no knowledge of any blocked squares. E4 is outside of the observable range and so the agent assumes it is free and the cost to move from 3E to 4E is 1. From here we calculate the g + h for each of the neighbors, where g is the cost to get to the neighbor and h is the approximate distance to the goal (ex: manhattan distance):

1E = g + h = 1 + 4 = 5 2D = g + h = 1 + 4 = 5 3E = g + h = 1 + 2 = 3

Since E3 has the lowest sum of cost and heuristic values, we greedily select the smallest value. It is not until after we select E3 and do another iteration of A* to observe the neighbors of E3 that we observe E4 is blocked, and correspondingly update the cost to move from E3 to E4 to ∞, so that in the next calculation of g + h values E4 can never be selected as the minimum and we have to turn around.

b. The guarantee that A* either finds a path to the goal state or terminates in finite time is due to the fact that A* maintains a closed list of nodes that have already been expanded in order to avoid cycles within the graph. Since A* maintains both a closed list and an open list, a node is added to the open list when it is first discovered as a neighbor and then taken out of the open list and added to the closed list when the node is processed. A* does not reprocess previously checked nodes in the closed list during the search. Since we have a finite number of unblocked positions in our gridworld which we'll call N, in the worst case we visit all 4N unblocked nodes, process them, and add N nodes to the closed list. In the worst case there will also be an upper bound of N nodes in the open list (liberal estimate), since if we encounter a state that's already in the open list but not in the closed list, we keep the smallest associated (g + h) rather than having multiple nodes for the same state in the open list. Once popped from the open list, processed, and added to the closed list, nodes cannot be revisited, and so a max of N nodes in the open list will correspond to a max of N nodes processed. Once the open list is empty, A* assumes there are no more ways to reach the goal state and the algorithm terminates.

§2 Part 2 - The Effects of Ties

This section delves into the implementation and analysis of two tie-breaking strategies within the Repeated Forward A* algorithm. When multiple cells have the same f-value, the algorithm needs a method to determine which cell to expand next. We investigate the effects of prioritizing cells with smaller g-values versus prioritizing those with larger g-values, comparing their performance in terms of the number of expanded cells.

§2.1 Implementation and Code Snippets

The provided code snippet showcases the implementation of these tie-breaking strategies. The core logic resides within the `gothroughbackwardastar` function (which, despite its name, implements a forward search). The function utilizes a binary heap (`BinaryHeap`) to efficiently manage the open list, ensuring that cells with the lowest f-values are prioritized.

The crucial part lies in the priority calculation, which is influenced by the `tie_break` parameter:

```
if tie_break == 'larger':
    c = rows * cols + 1
    priority = c * neighbor.f() - neighbor.g
else:
    priority = neighbor.f() + neighbor.g
```

When `tie_break` is set to `'larger'`, the priority is calculated as `c * neighbor.f() - neighbor.g`, where `c` is a constant larger than the maximum possible g-value. This effectively prioritizes cells with larger g-values. Conversely, when `tie_break` is not `'larger'`, the priority is calculated as `neighbor.f() + neighbor.g`, prioritizing cells with smaller g-values.

§2.2 Observations and Explanations

The choice of tie-breaking strategy significantly impacts the search behavior and the number of cells expanded. Prioritizing cells with smaller g-values tends to favor a breadth-first-like exploration, expanding nodes closer to the start node first. This can be visualized as a wavefront emanating from the start, gradually exploring the search space.

On the other hand, prioritizing cells with larger g-values encourages a more depth-first-like search, pushing the exploration more directly towards the goal. This strategy prioritizes nodes that have already progressed further from the start, potentially leading to a quicker discovery of the goal if it lies in a direction requiring a longer path.

Referring to Figure 9, if we start from cell 'A' and aim for cell 'T', the 'larger' g-value strategy might lead to a more direct path along the bottom row, as it prioritizes cells that have already ventured further from the start. In contrast, the 'smaller' g-value strategy might explore more cells in the upper rows before reaching the goal, as it prioritizes cells closer to the start.

The impact of these strategies on the number of expanded cells depends on the specific maze structure and the location of the goal. In mazes where the goal is located in a

direction requiring a long path, the 'larger' g-value strategy might expand fewer cells due to its focus on exploring deeper into the search space. Conversely, in mazes where the goal is relatively close to the start but requires navigating through many local paths, the 'smaller' g-value strategy might expand fewer cells due to its breadth-first approach.

In conclusion, the choice of tie-breaking strategy in Repeated Forward A* significantly influences the search behavior and the number of expanded cells. The optimal strategy depends on the specific characteristics of the search space and the location of the goal.

§3 Part 3 - Repeated Backward A* vs Repeated Forward A*

The Repeated Forward A* algorithm is a variant of the A* search algorithm designed for pathfinding in dynamic or partially unknown environments. It operates by repeatedly planning a path from the start to the goal, adapting to changes in the environment as they are discovered.

§3.1 Implementation Details

The implementation of Repeated Forward A* involves several key components. The `astar` class represents a node in the search space, encapsulating its coordinates, g-value (cost from the start), h-value (heuristic estimate), parent node, and obstacle status. The Manhattan distance is used as the heuristic function, ensuring admissibility and consistency.

```
class astar:
    def __init__(self, x, y, is_obstacle=False):
        self.x = x
        self.y = y
        self.g = math.inf
        self.h = 0
        self.parent = None
        self.is_obstacle = is_obstacle
    # ... (other methods)
```

The `gothroughastar` function implements the core A* logic. It initializes an open list (priority queue) and a closed set. The start node's g-value is set to 0, and its h-value is calculated. The algorithm then enters a loop, iteratively expanding nodes with the lowest f-value (g + h).

```
def gothroughastar(grid, start, goal, tie_break='larger'):
    rows = len(grid)
    cols = len(grid[0]) if rows > 0 else 0
    open_list = BinaryHeap()
    closed_set = set()
    # ... (initialization)
    while not open_list.is_empty():
```

```
        _, current = open_list.pop()
        # ... (rest of the loop)
```

Tie-breaking is handled based on the `tie_break` parameter, favoring either larger or smaller g-values. This choice can significantly impact the algorithm's behavior. Favoring larger g-values encourages a wider search, while favoring smaller g-values leads to a depth-first-like search.

§3.2 Tie-Breaking Strategies

The choice of tie-breaking strategy in Repeated Forward A* is crucial for its performance. Favoring larger g-values encourages the algorithm to explore cells further away from the start, potentially leading to a wider search. This strategy can be beneficial in scenarios where the goal is located in a direction requiring a long path.

Conversely, favoring smaller g-values encourages the algorithm to explore cells closer to the start, resulting in a depth-first-like search. This approach might lead to finding a path quicker, especially when the goal is relatively close to the start. However, it may also increase the risk of getting trapped in local minima or dead ends.

The `gothroughbackwardastar` function implements the Repeated Backward A* search algorithm. This function is structurally very similar to `gothroughastar` but operates in the reverse direction. Instead of starting from the start node and searching towards the goal, it starts from the goal node and searches towards the start node. This reversal of search direction can significantly impact the algorithm's performance, especially in environments where the branching factor differs between the start and goal regions.

```
def gothroughbackwardastar(grid, start, goal, tie_break='larger'):
    rows = len(grid)
    cols = len(grid[0]) if rows > 0 else 0
    open_list = BinaryHeap()
    closed_set = set()
    # ... (initialization)
    while not open_list.is_empty():
        _, current = open_list.pop()
        # ... (rest of the loop)
```

§3.3 Test Case and Explanation

To illustrate the behavior of Repeated Backward A*, consider a simple test case with a small grid:

```
Grid:
0 1 0
0 0 0
1 0 0

Start: (0, 0)
```

```
Goal: (2, 2)
```

§3.3.1 Explanation

In this grid, '0' represents an open cell, and '1' represents an obstacle. Applying Repeated Backward A*, the search starts from the goal (2, 2). The algorithm explores neighboring open cells, calculating their g and h values. The heuristic guides the search towards the start (0, 0). The path is reconstructed by tracing back from the start to the goal.

The key advantage of Backward A* in this scenario (and others) is that if the goal has a much lower branching factor, the search space is limited from the beginning. In this simple example, we can see that the goal only has two possible neighbors, whereas the start has two as well. In larger mazes, this difference can become much more pronounced.

The performance of Backward A* depends on the specific maze structure. In mazes with a low branching factor near the goal, Backward A* can be more efficient than Forward A*. Conversely, if the start has a lower branching factor, Forward A* is preferred.

Tie-breaking also plays a crucial role. Favoring larger g-values encourages exploration of cells farther from the goal, potentially leading to a wider search. This can be beneficial when the optimal path requires traversing a larger area. Conversely, favoring smaller g-values promotes a more focused, depth-first-like search, which can be faster if the optimal path is relatively direct. However, it may increase the risk of getting stuck in dead ends or local minima.

In larger mazes, the impact of tie-breaking can be more significant. The choice of tie-breaking strategy should align with the expected path characteristics and the desired balance between exploration and exploitation. This test case demonstrates the basic steps of Repeated Backward A*, and highlights the importance of tie-breaking strategies in influencing the search process and the resulting path.

§3.4 Repeated Forward A*

§3.4.1 Implementation

The implementation of Repeated Forward A* relies on several key components:

Node Representation: The 'astar' class encapsulates a node in the search space, storing its coordinates, g-value (cost from the start), h-value (heuristic estimate to the goal), parent node (for path reconstruction), and obstacle status. This object-oriented representation facilitates efficient management of nodes during the search process.

Heuristic Function: The Manhattan distance serves as the heuristic function, providing an admissible and consistent estimate of the distance between two nodes. This choice ensures that the algorithm finds the optimal path if one exists.

Priority Queue: A binary heap acts as the priority queue, efficiently managing the open list of nodes to be explored. The heap structure ensures that the node with the lowest f-value ($g + h$) is always at the top, enabling quick retrieval.

Search Logic: The ‘gothroughastar‘ function encapsulates the core A* search logic. It initializes the open list and closed set, sets the start node’s g-value to 0, calculates its h-value, and then iteratively expands nodes with the lowest f-value. The function also handles tie-breaking based on the ‘tie_break‘ parameter, which determines whether to favor larger or smaller g-values when f-values are equal.

§3.4.2 Tie-Breaking Strategies

The tie-breaking strategy plays a crucial role in the performance of Repeated Forward A*. Favoring larger g-values encourages exploration of nodes farther from the start, potentially leading to a wider search that is beneficial when the goal lies in a direction requiring a longer path. Conversely, favoring smaller g-values promotes a depth-first-like search, which can be faster if the goal is close to the start but may increase the risk of getting trapped in local minima or dead ends.

§3.5 Repeated Backward A*

§3.5.1 Implementation

The ‘gothroughbackwardastar‘ function implements the Repeated Backward A* algorithm. It shares a similar structure with ‘gothroughastar‘ but operates in the reverse direction, starting from the goal node and searching towards the start. This reversal can significantly impact performance, especially in environments where the branching factor (number of neighboring nodes) differs significantly between the start and goal regions.

§3.5.2 Advantages in Specific Scenarios

Repeated Backward A* can be particularly advantageous when the goal node has a lower branching factor than the start node. By starting the search from the goal, the algorithm effectively prunes the search space early on, as it explores fewer neighbors at each step. This can lead to significant improvements in efficiency, especially in larger mazes or graphs where the difference in branching factors is substantial.

§3.6 Comparison and Observations

Comparing Repeated Forward A* and Repeated Backward A* highlights the importance of considering the environment’s structure and the relative branching factors of the start and goal regions. In scenarios where the goal has a lower branching factor, Repeated Backward A* tends to be more efficient, as it explores a smaller portion of the search space. Conversely, when the start has a lower branching factor, Repeated Forward A* often performs better.

The choice of tie-breaking strategy further influences the behavior of both algorithms. Favoring larger g-values promotes a wider search, which can be advantageous when the

optimal path requires traversing a larger area. Favoring smaller g-values encourages a more focused, depth-first-like search, which may lead to faster solutions in some cases but also carries the risk of getting stuck.

§3.7 Conclusion

The selection between Repeated Forward A* and Repeated Backward A* depends on the specific characteristics of the problem and the environment. Analyzing the branching factors of the start and goal regions can provide valuable insights into which algorithm is likely to be more efficient. Additionally, the choice of tie-breaking strategy should align with the expected path characteristics and the desired balance between exploration and exploitation. By carefully considering these factors, we can effectively leverage the strengths of each algorithm to achieve optimal pathfinding performance.

§4 Part 4 - Heuristics in Adaptive A*

A heuristic function $h(s)$ is considered consistent if it satisfies the following properties:

1. $h_{\text{goal}} = 0$
2. $h(s) \leqslant c(s, a) + h(\text{succ}(s, a))$

§4.1 Part 4.1 - Manhattan Distance Consistency

The Manhattan distance in our grid world is defined as

$$h(s) = |x_s - x_{\text{goal}}| + |y_s - y_{\text{goal}}|$$

When $s = s_{\text{goal}}$,

$$\begin{aligned} h(s_{\text{goal}}) &= |x_{\text{goal}} - x_{\text{goal}}| + |y_{\text{goal}} - y_{\text{goal}}| \\ &= 0 \end{aligned}$$

We see that the first condition is satisfied, so we move to the next condition: the triangle inequality. Without loss of generality, assume the agent's current state is s, and the next state will be s', directly to the right of s. The y component of the Manhattan distance doesn't change, but the x component will become $|(x_s + 1) - x_{\text{goal}}|$. Therefore, $h(s') = h(s) \pm 1$. The maximum value attainable can be therefore described by the inequality $h(s') \leqslant h(s)+1$. This assumption can be generalized to all 4 cardinal directions that are represented in the gridworld, with moving left changing the x component to $|(x_s - 1) - x_{\text{goal}}|$, and moving up or down changing the y component instead.

Recalling that $c(s, a) = 1$ in our gridworld, we see that $h(s') \leqslant h(s) + c(s, a)$, so the Manhattan distance is consistent.

§4.2 Part 4.2 - Heuristic Consistency in Adaptive A*

We need to show that Adaptive A*'s heuristic function is both admissible and consistent. We proceed to first show that it is admissible, essentially showing that it never overestimates the actual cost. Let $h^*(s)$ be the actual cost to reach the goal from state s Using the heuristic update formula and the definition of shortest path,

$$h_{\text{new}}(s) = g(s_{\text{goal}}) - g(s)$$
$$g(s_{\text{goal}}) = g(s) + h^*(s)$$

Solving this system of equations, we see that $h^*(s) = h_{\text{new}}(s)$. Therefore, the heuristic matches the cost exactly, and doesn't overestimate it, and is therefore admissible.

We next show that the heuristic is consistent. Suppose the agent is in state s and after updating, it is in state s'. We know that $h_{\text{new}}(s) = g(s_{\text{goal}}) - g(s)$, which becomes $h_{\text{new}}(s') = g(s_{\text{goal}}) - g(s')$. Substituting this into the triangle inequality,

$$\begin{aligned} g(s_{\text{goal}}) - g(s) &\leqslant c(s,a) + g(s_{\text{goal}}) - g(s') \\ -g(s) &\leqslant c(s,a) - g(s') \\ g(s') &\geqslant g(s) + c(s,a) \end{aligned}$$

The last inequality always holds true because of A*'s path expansion guarantee:

$$g(s') = \min(g(s'), g(s) + c(s,a))$$

Since these steps are all reversible, the trinagle inequality holds for $h_{\text{new}}(s)$, making it a consistent heuristic.

In the course of running Adaptive A*, action costs can increase. If we relax the assumption that the gridworld is unchanging, these action cost increases can be brought forth by a change in the maze that blocks a previously unblocked path. This means that $g(s)$ has to be recalculated for some states and may increase. However, this doesn't change the admissibility of $h_{\text{new}}(s)$ because it is exactly the true cost of moving from state s to the goal.

§5 Part 5 - Adaptive A*

We implement and test the Adapttive A* algorithm in 'adaptiveastar.py'. The algorithm is designed to adapt to changes in the environment by updating the heuristic function based on the observed costs. The results of the Adaptive A* algorithm are compared to the standard A* algorithm to evaluate its performance in dynamic environments. The Adaptive A* algorithm is expected to be more robust and efficient in scenarios where the environment changes over time, as it can adjust its heuristic estimates to reflect the updated costs. We get that, across the 50 maze test cases, we get an average of 3639.4 expanded nodes, and an average runtime of 0.03351 seconds.

§6 Part 6 - Statistical Analysis

We will proceed by comparing Repeated Forward A* and Repeated Backward A*. Our null hypothesis will be that there is no difference in performance between the two algorithms. Our alternative hypothesis will be that there is a statistically significant difference in performance between the two algorithms.

Suppose we have n runs of each algorithm, searched on the same n test mazes that are guaranteed to be solvable, and run on the same computer for each test maze. Using a large number for n will result in a more consistent answer and reduce sampling noise, and we expect the results of our experiment to converge to the true performance difference as n grows larger. We will record both the runtimes and the number of expanded cells for both algorithms. We will then use this data to undergo a paired t-test for both the runtimes and number of expanded cells.

We will proceed by first analyzing the runtimes The paired t-statistic is

$$t = \frac{\bar{d}}{s_d/\sqrt{n}}$$

Here, $\bar{d}, s_d$ are the mean difference between the algorithms' runtimes, and s_d is the standard deviation of the differences in their runtimes. The differences are taken for each test case. This analysis can be repeated for the each algorithm's number of expanded cells.

For each t-statistic, we can choose a significance level such as $\alpha = 0.05$. We can find the p-value associated with the t-statistic by looking it up on a t-distribution table using $n - 1$ degrees of freedom. If the p-value is greater than α, then we fail to reject to reject the null hypothesis, meaning that there is no statistically significant difference between the performance of the two algorithms. On the other hand, if the p-value is less than α, then we fail to rejct the alternative hypothesis, meaning that there is a statistically significant difference between the two algorithms' performance.