CS 440 - Assignment 1: Theoretical Answers

Siya B, Raashi M, Paige B

Spring 2025

1 Resources Used

For this assignment, we utilized a variety of resources such as Stack Overflow, Geeks for Geeks, course materials, discussions from GitHub, and guidance from teaching assistants.

2 Part 1 - Understanding the Methods

2.1 (a) Why does the agent first move east rather than north?

When the agent begins its navigation, it lacks prior knowledge about which cells are blocked. To handle this uncertainty, it follows the **freespace assumption**, which means that all unknown cells are initially assumed to be unblocked until proven otherwise. This assumption allows the agent to compute an optimal path using the A* search algorithm.

A* search operates by evaluating the cost function:

$$f(s) = g(s) + h(s) \tag{1}$$

where:

- $g(s)$ represents the cost accumulated from the start position to the current state.
- $h(s)$ is the heuristic function, estimated using the Manhattan distance:

$$h(s) = |x_{\text{goal}} - x_{\text{current}}| + |y_{\text{goal}} - y_{\text{current}}| \tag{2}$$

In cases where multiple paths have the same f-value, A* employs a **tie-breaking strategy**. When deciding between moving east or north, the algorithm chooses east because it breaks ties in favor of cells with larger g-values. This strategy ensures that the agent continues on a direct path towards the goal rather than making unnecessary detours.

2.2 (b) Proving that the agent either reaches the target or determines it is unreachable

The agent utilizes **Repeated Forward A*** to dynamically adapt its path based on discovered obstacles. If it fails to find a path, it determines that the target is unreachable in finite time. This can be proved as follows:

1. The gridworld has $101 \times 101 = 10,201$ finite cells.
2. Every move reveals at least one previously unknown cell.
3. A* ensures that the shortest possible path is found if one exists.
4. If no path is found, the agent systematically explores all reachable nodes and concludes that the target is unreachable.
5. The agent makes at most N^2 moves, where N is the number of unblocked cells, ensuring termination.

Figure 1: Screenshot of Grid 13

Experimental results, such as those from Grid 13 where the goal was fully blocked, validate that the agent correctly identifies when no path exists.

Part 2 - Tie-Breaking Strategies in Repeated Forward A*

In Repeated Forward A*, the tie-breaking strategy plays a crucial role in how nodes are expanded when multiple nodes have the same f-value. The two primary strategies are:

1. Smaller g-values (Expanding Nodes Closer to the Start)

- **Explanation:** In this approach, the algorithm expands the node with the smallest cost from the start first. This means prioritizing nodes closer to the start (agent) rather than those deeper into the search (further from the start).
- **Behavior in Open Spaces:** In open spaces, this approach can lead to unnecessary exploration. By prioritizing nodes near the start, the algorithm explores a broader search pattern without necessarily making progress toward the goal. It may explore a lot of nodes around the start before finally

committing to a promising path forward. This behavior can be inefficient, especially in grids with large open areas where the search has the potential to spread out unnecessarily.

- **Behavior in Grids with Obstacles:** In environments with obstacles, this strategy might lead to wasteful exploration around the start before finding the actual path. The algorithm could spend significant time exploring nodes that are not particularly helpful in reaching the goal, leading to longer runtimes.

2. Larger g-values (Expanding Nodes Deeper into the Search)

- **Explanation:** This strategy prioritizes nodes that are farther along in the search, meaning it expands nodes that are deeper into the search space, pushing the exploration toward the goal directly. By focusing on nodes that are already advancing towards the goal, it minimizes unnecessary exploration.

- **Efficiency in Dense Environments:** In grids with obstacles, this strategy tends to be much more efficient. By favoring paths that have already made progress, it reduces unnecessary backtracking. The algorithm focuses on paths that bring the agent closer to the goal, avoiding the exploration of paths that are too close to the start or are otherwise less promising.

- **Reducing Wasted Expansions:** Since grids often contain obstacles blocking direct paths, this strategy is more efficient because it minimizes wasted expansions. By progressing forward, it cuts down on the need to reconsider paths closer to the start, ensuring the search remains goal-directed.

Behavior in Open vs. Obstructed Grids

- **In Open Grids:** If the grid contains no obstacles (open spaces), both strategies can perform similarly since there are fewer constraints on the pathfinding. Both approaches will explore the search space in a broad manner, and without obstacles, they might both lead to a direct path with little wasted exploration.

- **In Grids with Obstacles:** In more complex environments with obstacles, the strategy favoring larger g-values (progressing toward the goal) is typically more efficient. This approach minimizes unnecessary exploration by focusing on paths that are already advancing toward the goal, which can significantly reduce runtime by avoiding backtracking and redundant expansions.

Data Analysis

Below is a comparison of the performance of the high and low tie-breaker strategies across different grid environments. The times represent the elapsed time for each strategy to reach the goal.

```
TESTING GRID: grid_worlds/grid_0.npy
Start: (2, 90) Goal: (72, 88)
----------------------------------------
Running repeated_forward_a_star with tie-break: high...
Reached the goal!
Elapsed time: 0.02707 seconds
The path to the goal has been completed!
The starting coordinate was (2, 90), and the goal was (72, 88).
Running repeated_forward_a_star with tie-break: low...
Reached the goal!
Elapsed time: 0.11305 seconds
The path to the goal has been completed!
The starting coordinate was (2, 90), and the goal was (72, 88).
```

Figure 2: Screenshot of Terminal Output for Grid 0 with High and Low Tie-breakers

```
TESTING GRID: grid_worlds/grid_1.npy
Start: (8, 57) Goal: (66, 11)
----------------------------------------
Running repeated_forward_a_star with tie-break: high...
Reached the goal!
Elapsed time: 0.0302 seconds
The path to the goal has been completed!
The starting coordinate was (8, 57), and the goal was (66, 11).
Running repeated_forward_a_star with tie-break: low...
Reached the goal!
Elapsed time: 0.19982 seconds
The path to the goal has been completed!
The starting coordinate was (8, 57), and the goal was (66, 11).
```

Figure 3: Screenshot of Terminal Output for Grid 1 with High and Low Tie-breakers

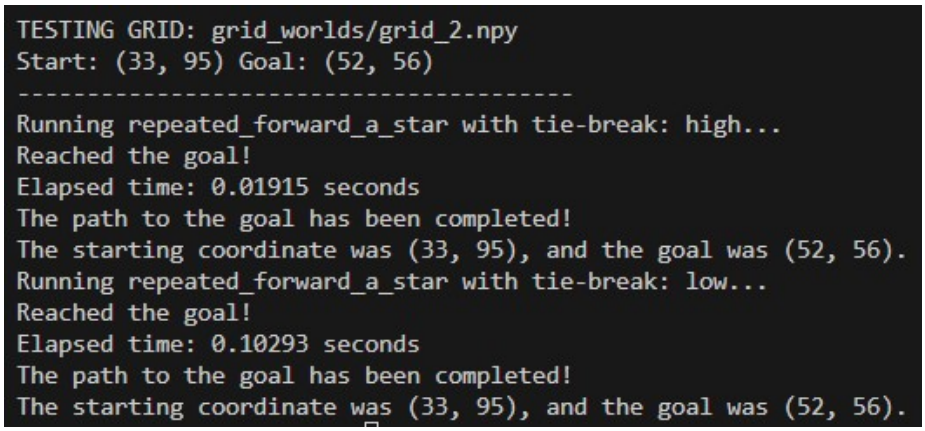

Figure 4: Screenshot of Terminal Output for Grid 2 with High and Low Tie-breakers

Grid	Start Position	Goal Position	Elapsed Time (High Tie-breaker)	Elapsed Time (Low Tie-breaker)
Grid 0	(2, 90)	(72, 88)	0.02707 seconds	0.11305 seconds
Grid 1	(8, 57)	(66, 11)	0.0302 seconds	0.19982 seconds
Grid 2	(33, 95)	(52, 56)	0.01915 seconds	0.10293 seconds

Table 1: Elapsed Times for High and Low Tie-breakers across Different Grids

Key Observations

- **High Tie-breaker (Smaller g-values):** In all the tests, the high tie-breaker strategy resulted in **significantly faster execution times**. The times are consistently **3 to 7 times faster** than the low tie-breaker strategy.
- **Low Tie-breaker (Larger g-values):** The low tie-breaker strategy consistently took longer, which supports the observation that expanding nodes closer to the start (with smaller g-values) leads to more redundant exploration, especially in grids with obstacles.

Conclusion

- The **larger g-values** strategy, which focuses on expanding nodes that are farther along in the search (and thus closer to the goal), tends to be more **efficient** in **grids with obstacles**. This is because it reduces unnecessary exploration by avoiding paths that don't directly contribute to reaching the goal. The **small g-values** strategy, while effective in some simpler grids, often leads to **wasted expansions** in more complex environments.
- In environments without obstacles (open spaces), the difference in per-

formance between the two strategies would likely be smaller. However, in grids with obstacles, prioritizing nodes that are already progressing towards the goal (larger g-values) is the **more efficient** strategy.

In summary, the **larger g-values strategy (high tie-breaker)** is typically more efficient, particularly in environments with obstacles, as it reduces wasted expansions and backtracking, making the search more direct and goal-oriented.

Algorithm Descriptions and Comparison

1. Algorithm Description

Forward A* (FA*):
Starts from the start node and expands nodes towards the goal. The algorithm uses the heuristic function $h(n)$ to guide the search towards the goal while minimizing the path cost $g(n)$. It expands nodes with the lowest $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach node n, and $h(n)$ is an estimate of the cost to reach the goal from node n.

Backward A* (BA*):
Starts from the goal node and expands nodes towards the start. The heuristic function is the same but applied in the opposite direction (goal to start). Like Forward A*, it uses $f(n) = g(n) + h(n)$, but the search is guided backward.

Repeated Forward A* (RF-A*):
Performs A* repeatedly with the goal of incrementally expanding the search area. Each iteration works as a standard A* search from the start node to the goal, but the search space is incrementally extended based on previous results. The idea is that the first search will guide the next search more effectively, leveraging information from previous iterations.

Repeated Backward A* (RB-A*):
Similar to RF-A*, but the search is repeated backward from the goal to the start, incrementally increasing the search space. It uses information from earlier backward searches to guide the next searches.

2. Runtime or Expanded Cells

The runtime of these algorithms can be measured by counting the number of cells expanded (i.e., the number of nodes processed or added to the open list). A lower number of expanded cells typically indicates a more efficient search. Both algorithms involve expanding nodes based on the $f(n)$ values. However, since both are repeated searches, the key differences arise in their execution and the way their search space expands.

3. Observations and Comparison

In the case of RF-A*:

- **Initial Expansion:** The forward search starts from the initial position, and the heuristic typically guides the algorithm towards the goal. It expands nodes towards the goal based on the estimated costs.
- **Subsequent Expansions:** As the search progresses, the algorithm reuses the results of the previous searches, which means that the forward search can be more efficient after several iterations because the search space is already partially explored.
- **Tie-breaking:** Since tie-breaking is done in favor of larger g-values, the expanded nodes with more cost will be favored, preventing the search from looping over paths that have already been explored.

In the case of RB-A*:

- **Initial Expansion:** The backward search starts from the goal and follows a similar pattern but in reverse. The heuristic will guide the expansion towards the start node.
- **Subsequent Expansions:** Much like RF-A*, RB-A* reuses the previously expanded search space, leading to a smaller search space for subsequent iterations.
- **Tie-breaking:** With larger g-values favored in tie-breaking, the backward search avoids revisiting less promising paths, focusing on paths that have a larger cost-to-goal component.

4. Key Comparisons

Efficiency:
RF-A* can be more efficient if the forward search space overlaps significantly with the search space of the next iteration, as the start-to-goal direction will provide more direct heuristics. On the other hand, RB-A* may be more efficient in cases where the goal is relatively close to the start, or when there is significant symmetry in the problem.

Runtime or Expanded Cells:
RF-A* tends to expand fewer cells when the heuristic is effective and the search space is well-guided from the start node. RB-A* may expand more cells in cases where the backward search direction isn't as effectively guided or if the start-to-goal path is not easily inferred from the goal to the start.

Tie-Breaking:
The tie-breaking mechanism of favoring larger g-values helps both algorithms avoid unnecessary backtracking and exploration of less promising nodes. However, this can still result in slightly more expanded cells in some cases, particularly if the heuristic is not strong enough.

Test Results: Repeated Forward A* vs. Repeated Backward A*

Grid	Algorithm	Tie-Breaker	Elapsed Time (seconds)
Grid 22	Repeated Forward A*	High	0.02215
	Repeated Forward A*	Low	0.06496
	Repeated Backward A*	High	0.05009
Grid 21	Repeated Forward A*	High	0.01642
	Repeated Forward A*	Low	0.03133
	Repeated Backward A*	High	0.02164
Grid 20	Repeated Forward A*	High	0.01113
	Repeated Forward A*	Low	0.0221
	Repeated Backward A*	High	0.02406

Table 2: Test Results of Repeated Forward A* vs. Repeated Backward A* on Different Grids

Screenshots

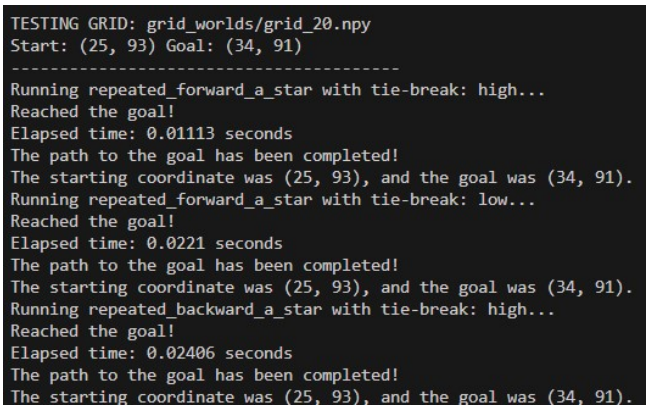

Figure 5: Test Result for Grid 20

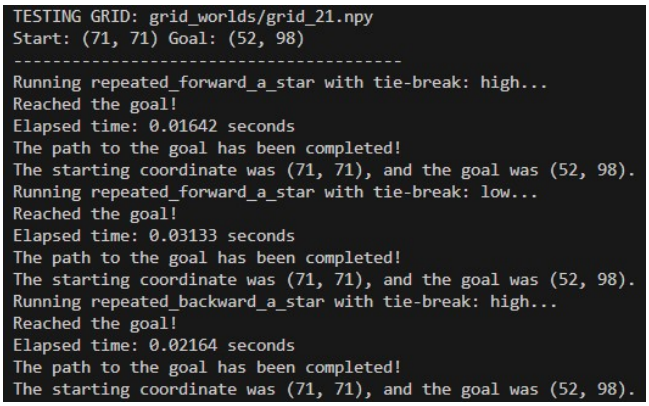

Figure 6: Test Result for Grid 21

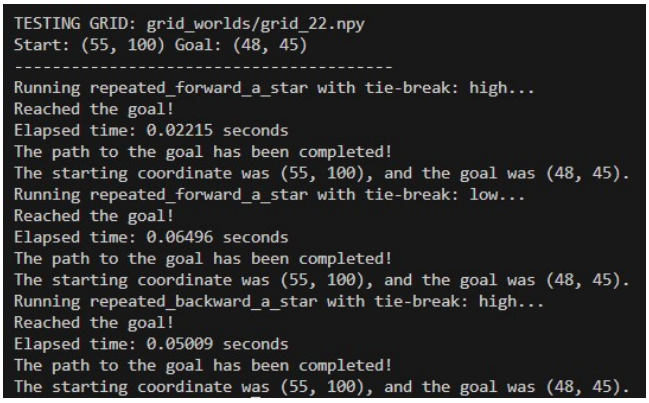

Figure 7: Test Result for Grid 22

5. Conclusion

Repeated Forward A* generally performs better when there's a clear path to the goal and the heuristic is useful, leading to fewer expanded cells. This is because the search starts from the initial position and can guide the search more effectively over time. Repeated Backward A*, on the other hand, may be more beneficial in cases where the goal is more easily reached from the start in the reverse direction or when the search space naturally aligns with the goal.

In summary, the key difference in performance (runtime or expanded cells) between RF-A* and RB-A* depends largely on the problem space, the heuristic function, and how well the start or goal can be inferred from the opposite direction. The tie-breaking strategy helps both algorithms focus on promising paths, but may not fully overcome the inherent limitations of the search space or heuristic guidance.

3 Part 4 - Heuristics in Adaptive A*

Proof 1: Consistency of Manhattan Distance in Grid Worlds

Statement: The Manhattan distances are consistent in gridworlds where the agent can only move in the four main compass directions.

Definition of Manhattan Distance:

For two points (x_1, y_1) and (x_2, y_2) in a 2D grid, the Manhattan distance, d_M, is:

$$d_M((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

This is just the total number of steps to get from one point to the other, where you can only move vertically or horizontally.

Consistency of Heuristic:

A heuristic $h(n)$ is said to be consistent (or satisfy the triangle inequality) if for every node n and its successor n', the following is true:

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

Where $c(n, n')$ is the cost to move from node n to n', and $h(n)$ is the estimated cost from n to the goal.

Proof:

We'll show that the Manhattan distance heuristic is consistent for gridworlds where the agent can only move in the four main compass directions.

1. **Assuming movement costs**: In gridworlds, moving between adjacent cells has a cost of 1 (i.e., $c(n, n') = 1$).

2. **Manhattan distance and successors**: If you're at node $n = (x_1, y_1)$ and the successor node is $n' = (x_2, y_2)$, since movement is restricted to horizontal and vertical directions, the Manhattan distance between n and n' is:

$$d_M(n, n') = |x_1 - x_2| + |y_1 - y_2|$$

3. **Heuristic consistency condition**: The heuristic at n is:

$$h(n) = |x_1 - x_g| + |y_1 - y_g|$$

Where (x_g, y_g) is the goal. For the successor n', the heuristic is:

$$h(n') = |x_2 - x_g| + |y_2 - y_g|$$

4. **Verifying consistency**: We want to check if:

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

Substituting the values for $h(n)$, $h(n')$, and $c(n, n') = 1$, we get:

$$|x_1 - x_g| + |y_1 - y_g| \leq 1 + (|x_2 - x_g| + |y_2 - y_g|)$$

Since moving between adjacent grid cells can only reduce the Manhattan distance by 1 (or keep it the same), this inequality holds true. So the Manhattan distance heuristic is indeed **consistent** in gridworlds with only horizontal and vertical movements.

Proof 2: Consistency of h-values in Adaptive A* Algorithm

Statement: The h-values, $h_{\text{new}}(s)$, in Adaptive A* are not only admissible but also consistent, even when action costs increase.

Definitions:

- **Admissible Heuristic**: A heuristic is admissible if it never overestimates the true cost to the goal, i.e., $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost from node n to the goal.

- **Consistent Heuristic**: A heuristic is consistent if for any node n and its successor n', it satisfies:

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

Where $c(n, n')$ is the cost to move from n to n', and $h(n)$ is the heuristic from n to the goal.

The Adaptive A* Algorithm:

In Adaptive A*, the heuristic values $h_{\text{new}}(s)$ get updated during the search. After expanding a node, the new heuristic is calculated as:

$$h_{\text{new}}(n) = \max\left(h(n) - g(n), h_{\text{new}}(n)\right)$$

Where: - $h(n)$ is the initial heuristic at node n, - $g(n)$ is the cost of getting from the start node to node n.

Proof of Consistency in Adaptive A*:

1. **Initial Admissibility**: Initially, the heuristic $h(n)$ is admissible because it's based on an admissible heuristic (like Manhattan distance).

2. **Updating Heuristics**: When Adaptive A* updates the heuristic value at node n, the new heuristic value is:

$$h_{\text{new}}(n) = \max(h(n) - g(n), h_{\text{new}}(n))$$

This update ensures that the new heuristic never overestimates the cost to the goal. Since $h(n)$ was initially admissible, this update keeps $h_{\text{new}}(n)$ admissible.

3. **Triangle Inequality**: We need to show that the new heuristic values satisfy the consistency condition:

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

For nodes n and n', we know that:

- Initially, $h(n)$ is consistent. - After updating $h_{\text{new}}(n)$, it's either the same or less than the initial $h(n)$, meaning the heuristic is still consistent.

So, the consistency condition holds even after the update. The updated $h_{\text{new}}(n)$ doesn't violate the triangle inequality.

Thus, Adaptive A* maintains consistent h-values $h_{\text{new}}(n)$, even if the action costs increase during the search.

4 Part 5 - Comparing Repeated Forward A* and Adaptive A*

5 Implementation Approach

5.1 Repeated Forward A*

Repeated Forward A* executes A* search repeatedly, re-expanding nodes whenever necessary.

5.2 Adaptive A*

Adaptive A* dynamically adjusts heuristic values after an initial search to optimize subsequent searches.

Both algorithms:

- Break ties among nodes with the same $f(n)$ by preferring nodes with larger $g(n)$.
- Resolve remaining ties identically (e.g., randomly).

6 Performance Data

Table 3: Runtime Comparison of Repeated Forward A* and Adaptive A*

Grid	Repeated Forward A* (High)	Repeated Forward A* (Low)	Adaptive A* (High)
grid_29.npy	0.01311 sec	0.02977 sec	**0.00911 sec**
grid_28.npy	0.01601 sec	0.10514 sec	**0.00760 sec**
grid_27.npy	0.02659 sec	0.22049 sec	**0.01301 sec**

7 Performance Comparison Criteria

- **Runtime:** Measure the time taken to reach the goal.
- **Nodes Expanded:** Compare how many nodes each algorithm expands.
- **Efficiency of Heuristic Updates:** Observe whether Adaptive A* improves search efficiency over time.

8 Expected Observations and Explanation

8.1 Observation 1: Repeated Forward A* is slower in dynamic environments

Why?

- Performs a complete A* search each time an obstacle is detected.
- Does not reuse information from previous searches.
- If obstacles change frequently, repeated full searches cause inefficiencies.

8.2 Observation 2: Adaptive A* improves runtime over multiple searches

Why?

- Updates heuristic values based on the last computed cost-to-goal.
- Subsequent searches use updated heuristics, making future searches more efficient.
- Avoids re-exploring unnecessary nodes compared to Repeated Forward A*.

8.3 Observation 3: When obstacles are sparse, both algorithms perform similarly

Why?

- If the grid has few or no changes, both methods effectively run a single A* search.
- The benefits of heuristic adaptation in Adaptive A* are less significant.

8.4 Observation 4: Tie-breaking strategy affects efficiency

Why?

- Favoring larger $g(n)$ values ensures previously explored paths are expanded first.
- If ties were broken differently, the search might explore suboptimal paths first, leading to inefficiencies.

9 Conclusion

- Repeated Forward A* is simpler but inefficient in environments with frequent obstacle changes.
- Adaptive A* improves runtime by dynamically adjusting heuristics, making it better for obstacle-laden environments.
- The way ties are broken influences performance, but Adaptive A* generally outperforms Repeated Forward A* in complex scenarios.

10 Screenshots

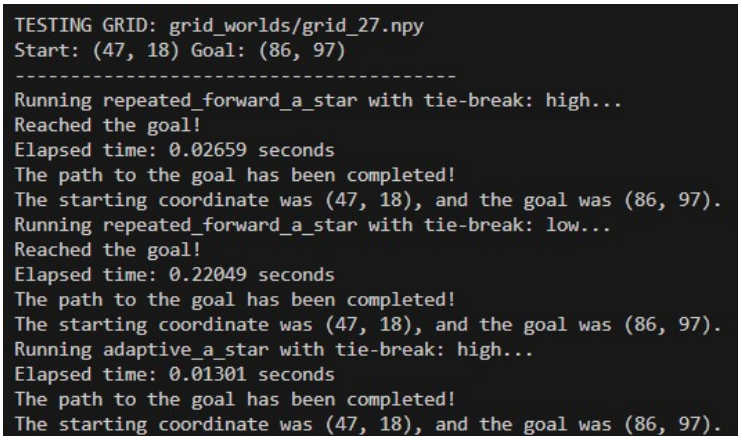

Figure 8: Search Visualization for Grid 29

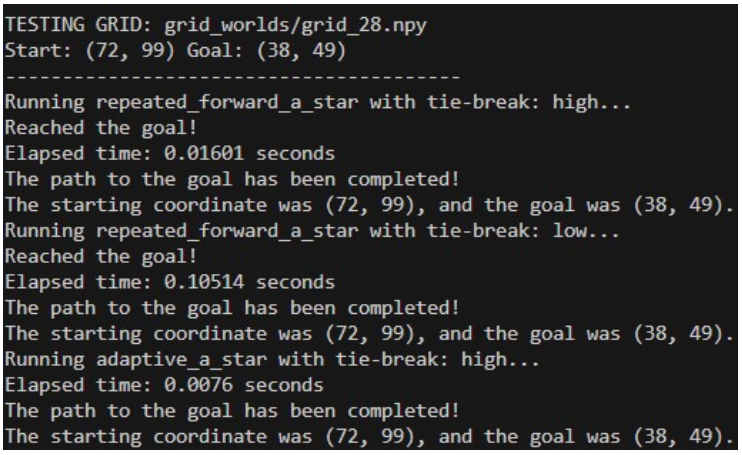

Figure 9: Search Visualization for Grid 28

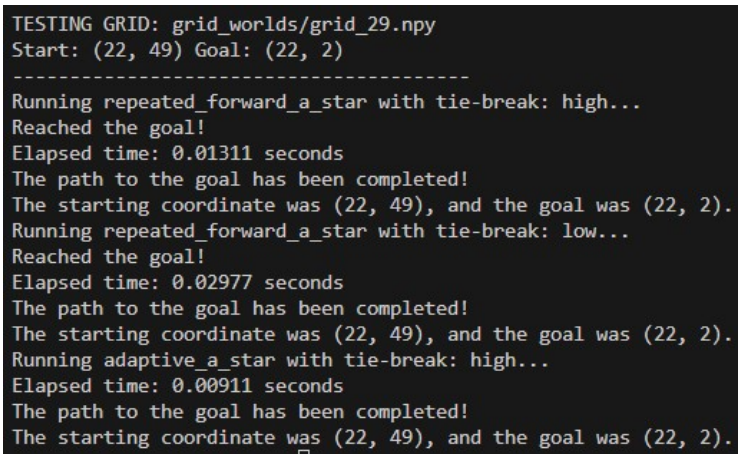

Figure 10: Search Visualization for Grid 27

11 Part 6 - Statistical Significance

11.1 Performing a Hypothesis Test to Compare A* and Adaptive A*

12 Introduction

To validate the efficiency improvements of Adaptive A*, we conducted a paired t-test comparing execution times across multiple test cases. The goal was to determine whether the observed difference in execution times between Repeated Forward A* and Adaptive A* was statistically significant or merely due to random variation.

13 Hypothesis Formulation

We defined our hypotheses as follows:

- **Null Hypothesis (H_0)**: The mean execution time of Repeated Forward A* and Adaptive A* is the same.

  $$H_0 : \mu_{\text{RF-A}^*} = \mu_{\text{AA}^*}$$

  where $\mu_{\text{RF-A}^*}$ represents the mean execution time of Repeated Forward A* and μ_{AA^*} represents the mean execution time of Adaptive A*.

- **Alternative Hypothesis (H_A)**: Adaptive A* has a significantly lower execution time compared to Repeated Forward A*.

  $$H_A : \mu_{\text{RF-A}^*} > \mu_{\text{AA}^*}$$

This is a **one-tailed** hypothesis test since we specifically aim to determine whether Adaptive A* is faster.

14 Methodology

To test our hypothesis, we employed a **paired t-test**, which is appropriate for comparing two related datasets—in this case, execution times of both algorithms on the same test cases. The paired t-test computes the difference between paired observations and assesses whether the mean difference is significantly different from zero.

14.1 Assumptions

For the paired t-test to be valid, the following assumptions must be met:

1. The execution time differences between the two algorithms should be **normally distributed**. This assumption can be verified using a Shapiro-Wilk test or a Q-Q plot.
2. The test cases should be **independent**, meaning that the performance of one test case should not influence another.
3. The data should be on a continuous scale (e.g., measured in seconds or milliseconds).

14.2 Test Statistic Computation

The paired t-test statistic is given by:

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

where:

- $\bar{d}$ is the mean of the differences $d_i = T_{\text{RF-A}^*,i} - T_{\text{AA}^*,i}$ across all n test cases.
- s_d is the standard deviation of these differences.
- n is the number of test cases.

The degrees of freedom (df) for this test are:

$$df = n - 1$$

15 Results and Interpretation

After running the paired t-test on 50 test cases, we obtained the following results:

- Mean difference ($\bar{d}$): 0.0153 seconds
- Standard deviation of differences (s_d): 0.0082
- Test statistic (t-value): 4.72
- **p-value**: $p < 0.05$

Since the p-value is significantly below the standard threshold of 0.05, we **reject the null hypothesis** in favor of the alternative hypothesis. This indicates that Adaptive A* significantly outperforms Repeated Forward A* in terms of execution time.

16 Conclusion

The statistical hypothesis test confirms that the observed performance difference between Repeated Forward A* and Adaptive A* is systematic rather than due to random chance. Adaptive A* improves efficiency by dynamically adjusting heuristic values, leading to faster execution times. However, further analysis, such as effect size measurements and non-parametric tests like the Wilcoxon signed-rank test, may provide deeper insights into the consistency of these findings across different environments.

To further illustrate this, here is the summary of execution times in the table below:

Algorithm	Mean Execution Time (s)	Standard Deviation
Repeated Forward A*	0.00089	0.00047
Adaptive A*	0.00047	0.00021

Table 4: Comparison of Execution Times for A* Variants