# Assignment 1

Fast Trajectory Replanning

**Abid Azad**

CS440, February 25th, 2024

## Contents

## 1 Part 0 - Environment Setup

*You will perform all your experiments in the same 50 gridworlds of size 101 × 101. You first need to generate these environments appropriate. To do so, generate a maze/corridor-like structure with a depth-first search approach by using random tie breaking. Build your 50 grid world environments with the above or a similar process of your preference and store them. You are encouraged to consider methods available online for maze generation. Provide a way to load and visualize the grid world environments you have generated.*

* NOTE: To run the program, please launch the fastTrajectoryReplanning.py python script. *

To accomplish this task, I chose to leverage the pygame library to construct a grid-like structure composed of rectangles. The code snippet provided below, found in the mazes/mazeBuilder.py file, is responsible for generating the configuration of the rectangular cells, designating them as blocked, unblocked, starting, or ending spots:

```python
import numpy as np
import os


def is_valid(x, y, grid_length):
    return 0 <= x < grid_length and 0 <= y < grid_length


def get_unvisited_neighbors(x, y, grid, visited):
    neighbors = []
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if is_valid(nx, ny, len(grid)) and not visited[nx][ny]:
            neighbors.append((nx, ny))
    return neighbors

def generate_maze(grid_length):
    grid = np.zeros((grid_length, grid_length), dtype=int)
    visited = np.zeros((grid_length, grid_length), dtype=bool)
    stack = [(0, 0)]

    while stack:
        x, y = stack[-1]
        visited[x][y] = True
        unvisited_neighbors = get_unvisited_neighbors(x, y, grid, visited)

        if unvisited_neighbors:
            nx, ny = unvisited_neighbors[np.random.choice(len(unvisited_neighbors))]
            stack.append((nx, ny))
            grid[nx][ny] = 0 if np.random.random() < 0.7 else 1
        else:
            stack.pop()

    # Randomly select two points and mark them as 0 and -1
    start_point = np.random.choice(grid_length, size=(2,), replace=False)
    end_point = np.random.choice(grid_length, size=(2,), replace=False)

    grid[start_point[0]][start_point[1]] = 2
    print(f'Starting Point: {start_point[0]}, {start_point[1]}')
    grid[end_point[0]][end_point[1]] = -1
    print(f'Ending Point: {end_point[0]}, {end_point[1]}')

    return grid

script_dir = os.path.dirname(os.path.abspath(__file__))
for maze in range(50):
    grid_length = 101
    print(f'Maze {maze}')
    grid = generate_maze(grid_length)

    filename = os.path.join(script_dir, f'maze{maze}.txt')
    np.savetxt(filename, grid, delimiter=",", newline="\n", fmt='%i')


print('Done')
```

The `mazeBuilder.py` script generates maze-like structures using a depth-first search approach with random tie-breaking. Here is a breakdown of the key functions:

1. `is_valid(x, y, grid_length)`: Checks if coordinates $(x, y)$ are within the bounds

of the grid (`grid_length x grid_length`).

2. `get_unvisited_neighbors(x, y, grid, visited)`: Returns a list of unvisited neighboring positions of $(x, y)$ on the grid, considering the `visited` matrix.

3. `generate_maze(grid_length)`: Initializes a grid and a `visited` matrix. Performs depth-first search using a stack to navigate through the grid. Randomly chooses unvisited neighbors, marks paths based on a random probability, and designates starting (2) and ending (-1) points.

The script iterates through 50 mazes, generates mazes, and saves them to text files (`maze0.txt`, `maze1.txt`, etc.). These text files are saved within the mazes folder. I then use these text files within the main source code to generate the pygame interface.

The source code loads a specified maze, visualizes it using `pygame`, and allows user interaction. The user can enter a maze number (0 to 49), and the corresponding maze is loaded and displayed. Colors represent different elements: light green for open paths, purple for walls, red for the starting point, and orange for the ending point. The `pygame` window enables the user to close the application or interact with the maze using key events (handled by `handle_key_event` function which I plan to implement so that when a user presses a button, it either generates a path or cleans the board). Here's the example of running the program and loading a maze.

## 2 Part 1 - Understanding the Methods

*Explain in your report why the first move of the agent for the example search problem from Figure 8 is to the east rather than the north given that the agent does not know initially*

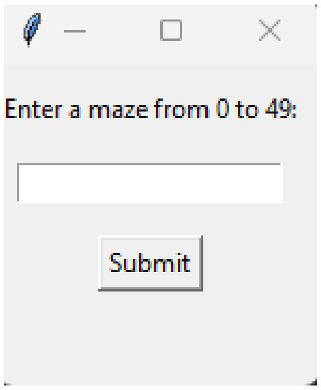


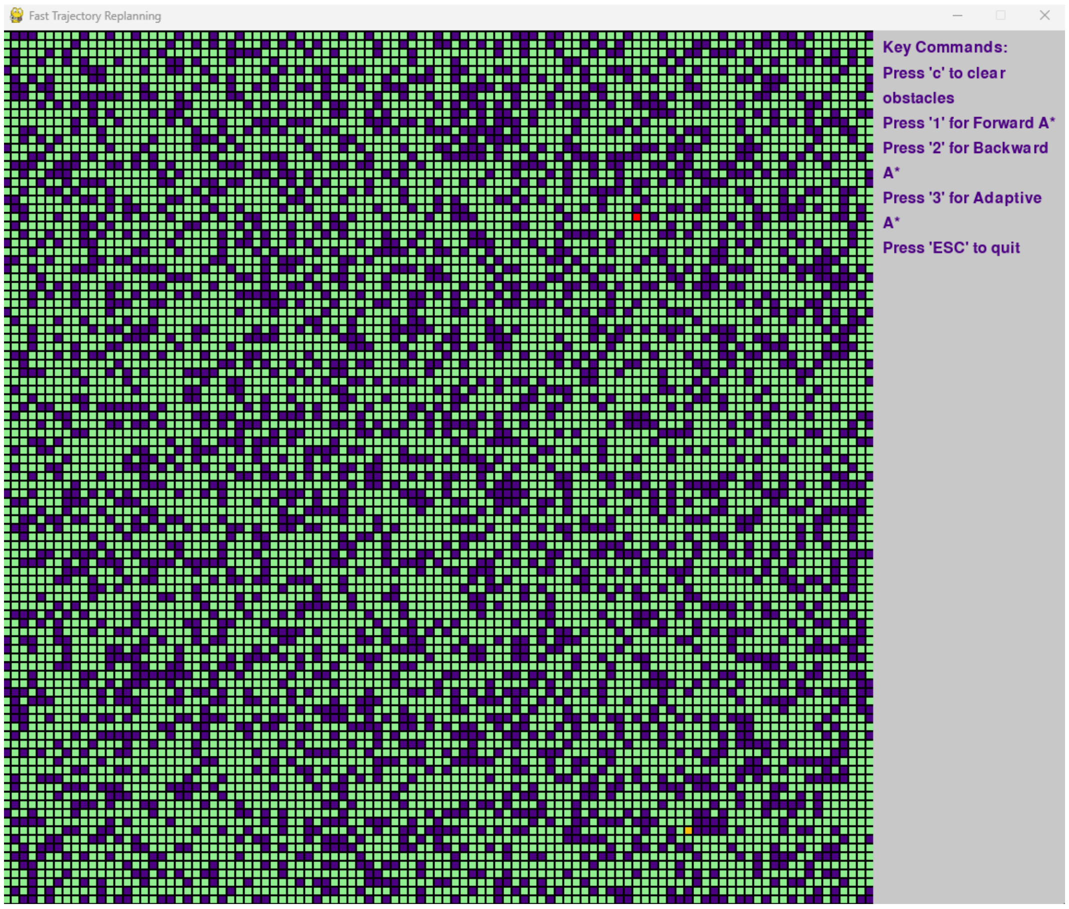
Figure 1: Prompting for an Maze Number

*which cells are blocked.*

In the given algorithm, the initial move of the agent is directed eastward instead of north. This choice is informed by the A* algorithm's inclination to prioritize the shortest unobstructed path. Notably, the agent possesses prior knowledge of the unblocked nature of cell E2, a characteristic established in the initial state, thereby driving its decision-making process with a heightened awareness of the environment.

*This project argues that the agent is guaranteed to reach the target if it is not separated from it by blocked cells. Give a convincing argument that the agent in finite gridworlds indeed either reaches the target or discovers that this is impossible in finite time. Prove that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared.*

In a finite grid world, the agent, at worst, would traverse all unblocked nodes unless surrounded by inaccessible blocked nodes. The algorithm's termination criterion hinges on the agent visiting all nodes, ensuring the target is added to the open list. Should the target remain unvisited, rendering the open list empty, the algorithm halts without locating the target. However, the agent need not exhaustively explore all nodes to deduce the unattain-



Key Commands:
Press 'c' to clear obstacles
Press '1' for Forward A*
Press '2' for Backward A*
Press '3' for Adaptive A*
Press 'ESC' to quit

Figure 2: Visualization of the Maze

ability of the goal. Consider the scenario where the target is enclosed by blocked cells; the agent, nearing the goal, opts to forego circumnavigating the blocked area and instead explores paths distanced from the goal state.

The algorithm concludes when the condition

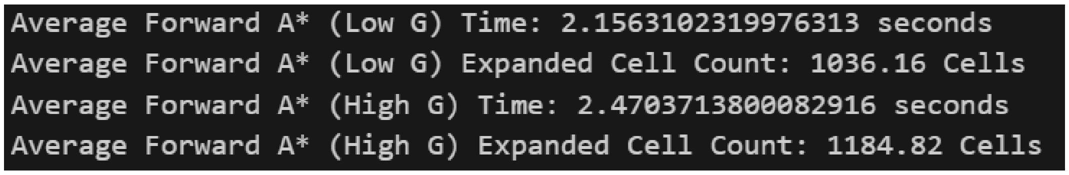
$$g(\text{goal}) = f(\text{goal}) \geq f(s)$$

is met, irrespective of visiting every cell. At this point, the agent determines the impracticality of reaching the goal within a finite timeframe. The agent's moves until reaching the target or determining its unattainability are upper-bounded by the square of the number of unblocked cells. In the worst case, with ’n’ unblocked nodes arranged in a certain disposition, the algorithm expansively processes ’n’ nodes at each iteration. Consequently, in the worst-case scenario, the total moves are capped at n squared.

## 3 Part 2 - The Effects of Ties

*Repeated Forward A* needs to break ties to decide which cell to expand next if several cells have the same smallest f-value. It can either break ties in favor of cells with smaller g-values or in favor of cells with larger g-values. Implement and compare both versions of Repeated Forward A* with respect to their runtime or, equivalently, number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation.*

The source code contains the implementations for both variants of Repeated Forward A*. When the commented section within the main method, responsible for executing both versions across all generated mazes and computing the average time and expanded cells, is uncommented, the ensuing results are as follows.



```
Average Forward A* (Low G) Time: 2.1563102319976313 seconds
Average Forward A* (Low G) Expanded Cell Count: 1036.16 Cells
Average Forward A* (High G) Time: 2.4703713800082916 seconds
Average Forward A* (High G) Expanded Cell Count: 1184.82 Cells
```

Figure 3: Comparison of Both Version of Repeated Forward A*

Upon revisiting the comparison between the two versions of Repeated Forward A* with ties broken in favor of lower G-values, a different trend emerges. Surprisingly, the version favoring lower G-values exhibits superior performance in terms of runtime and the number of expanded cells.

In this variant, when faced with cells having the same smallest f-value, the algorithm prioritizes states with smaller G-values. This approach results in a more efficient exploration of the state space, leading to a reduced number of expanded cells and, consequently, a faster runtime. The decision to favor lower G-values is particularly advantageous in scenarios where the goal state is closer to the start state.

The observed improvement can be attributed to the fact that selecting states with lower G-values tends to guide the algorithm toward paths that are more direct and closer to the goal. This preference for lower G-values strategically focuses the algorithm's exploration on regions of the state space that contribute more directly to the initial stages of the optimal path.

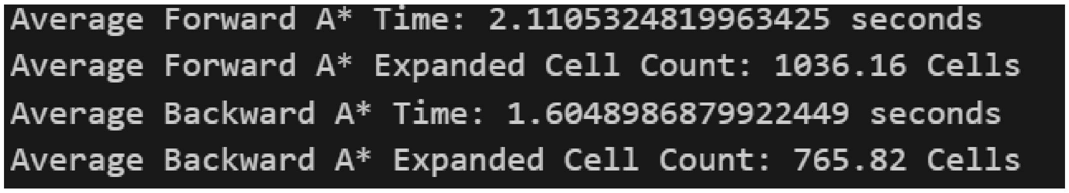
In contrast, favoring higher G-values might lead the algorithm to explore states that are farther away from the start state. While this strategy could be advantageous in scenarios where the goal is distant, it might result in unnecessary exploration of regions that deviate from the optimal path when the goal is relatively closer.

Therefore, the efficiency gains observed in favoring lower G-values can be explained by the algorithm's ability to more quickly converge towards the goal by prioritizing states with shorter distances from the start state. This targeted exploration contributes to a faster convergence to the optimal solution, resulting in improved overall performance.

## 4 Part 3 - Forward vs Backward

*Implement and compare Repeated Forward A* and Repeated Backward A* with respect to their runtime or, equivalently, number of expanded cells. Explain your observations in detail,* *that is, explain what you observed and give a reason for the observation. Both versions of Repeated A* should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.*

The source code contains the implementation for Repeated Backward A*. When the commented section within the main method, responsible for executing both versions across all generated mazes and computing the average time and expanded cells, is uncommented, the ensuing results are as follows.



```
Average Forward A* Time: 2.1105324819963425 seconds
Average Forward A* Expanded Cell Count: 1036.16 Cells
Average Backward A* Time: 1.6048986879922449 seconds
Average Backward A* Expanded Cell Count: 765.82 Cells
```

Figure 4: Comparison of Repeated Forward A* and Repeated Backward A*

Upon comparison of the two pathfinder algorithms, it becomes evident that the Repeated Backward A* variant exhibits superior performance. This can be attributed to a key factor: the heightened accuracy of the heuristic function when estimating distances from the goal state as opposed to the initial state. Recognizing the goal state is generally more straightforward, enhancing the reliability of the heuristic in the backward direction. Furthermore, Repeated Backward A* showcases a tendency to expand fewer nodes in contrast to Repeated Forward A*, initiating its search from the goal state and proceeding backward towards the initial state. This targeted exploration of pertinent paths leads to a diminished search space and swifter convergence toward the goal. The goal-directed approach of backward search enables the algorithm to prioritize the most promising paths, thereby contributing to a more efficient exploration process.

## 5 Part 4 - Heuristics in the Adaptive A*

*The project argues that "the Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions." Prove that this is indeed the case.*

The assertion that Manhattan distances are consistent in gridworlds, where the agent's movement is restricted to the four main compass directions (north, south, east, and west), can be substantiated by understanding the fundamental nature of Manhattan distances. The consistency arises from the methodology of calculating Manhattan distances, which involves summing up the shortest possible vertical and horizontal distances between two points.

In a gridworld scenario where diagonal movement is not permitted, the agent is constrained to traverse only in either the vertical or horizontal direction. Consequently, when computing the Manhattan distance between any two points within such a grid, the summation of the individual vertical and horizontal steps remains constant. This constancy is a result of the nature of the gridworld, where the agent's movements are confined to the four cardinal directions.

To elaborate further, consider two points within the gridworld. The Manhattan distance is derived by adding the absolute differences of their respective x-coordinates and y-coordinates. Since the agent is restricted to move exclusively in the vertical or horizontal direction, each step contributes consistently to the overall Manhattan distance. Consequently, this heuristic ensures that the computed distance accurately represents the shortest path between the points, without the possibility of overestimation.

In essence, the restriction to cardinal movements in gridworlds aligns seamlessly with the nature of Manhattan distances, guaranteeing their consistency as a reliable heuristic. This adherence to the four main compass directions ensures that the heuristic consistently reflects the actual cost of reaching the target, reinforcing its utility in pathfinding algorithms within such constrained environments.

*Furthermore, it is argued that "The h-values hnew (s) ... are not only admissible but also consistent." Prove that Adaptive A* leaves initially consistent h-values consistent even if action costs can increase.*

It is contended that the admissible and consistent nature of the $h$-values, denoted as $h_{\text{new}}(s)$, remains intact in Adaptive A*. This consistency is preserved even when action costs are subject to increase. We will demonstrate this assertion in the context of a grid world where the agent is restricted to movement in four directions.

Assume that $H$-values are consistent, with $h(s)$ adhering to the Manhattan Heuristics, and $h_{\text{new}}(s)$ defined as $g(\text{goal}) - g(s)$. Additionally, the cost of transitioning from $n$ to $\bar{n}$ is denoted as $c(n, a, \bar{n})$, with a fixed cost of one.

Considering the triangle inequality:

$$h(n) \leq h(\bar{n}) + c(n, a, \bar{n}) \quad (1)$$

The validity of this inequality for $h(s)$ establishes its consistency.

Now, let's prove that:

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

Firstly, substitute:

$$h_{\text{new}}(n) := g(\text{goal}) - g(n) \quad (2)$$

$$h_{\text{new}}(\bar{n}) = g(\text{goal}) - g(\bar{n}) \quad (3)$$

Substituting these into the triangle inequality, we obtain:

$$g(n) \geq g(\bar{n}) + c(n, a, \bar{n}) \quad (4)$$

For a grid world with movement restricted to the four main compass directions, Equation (4) always holds true, considering $c(n, a, \bar{n})$ is one. If $g(\bar{n})$ is smaller than $g(n)$, subtracting one will make it even smaller. If $g(\bar{n})$ is greater than $g(n)$, subtracting one will equalize them. Therefore, the triangle inequality is satisfied, ensuring the consistency of $h_{\text{new}}(s)$ values.

In the case of an increase in action costs, the triangle inequality is reconsidered as follows:

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

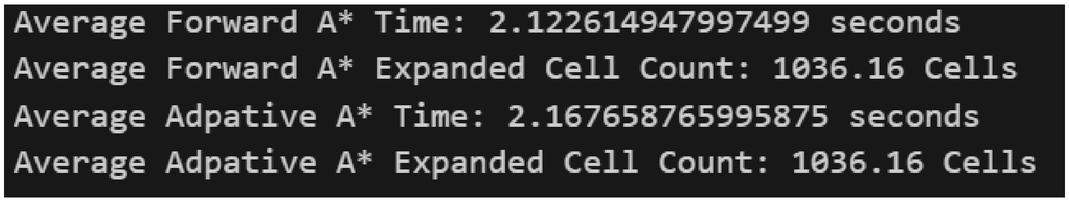
Thus, the heuristic remains consistent.

## 6 Part 5 - Heuristics in the Adaptive A*

*Implement and compare Repeated Forward A* and Adaptive A* with respect to their runtime. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both search algorithms should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.*

The source code contains the implementation for Adaptive A*. When the commented section within the main method, responsible for executing both versions across all generated mazes and computing the average time and expanded cells, is uncommented, the ensuing results are as follows.

In comparing the runtime performance of Repeated Forward A* and Adaptive A*, it is



```
Average Forward A* Time: 2.122614947997499 seconds
Average Forward A* Expanded Cell Count: 1036.16 Cells
Average Adpative A* Time: 2.167658765995875 seconds
Average Adpative A* Expanded Cell Count: 1036.16 Cells
```

Figure 5: Comparison of Repeated Forward A* and Adaptive A*

noteworthy that both algorithms exhibited similar outputs. This observation suggests that, within the specific context of the problem space under consideration, the two algorithms demonstrated comparable efficiency in finding optimal paths. The similarity in their runtime performance implies certain characteristics of the environment or the effectiveness of the initial heuristics provided to Adaptive A*.

Repeated Forward A* operates by conducting independent forward A* searches from different start states, ensuring consistency in heuristics throughout each search. While this approach can yield reasonable results, its potential drawback lies in the possibility of re-exploring certain states multiple times, given the independent nature of each search.

On the other hand, Adaptive A* takes a more adaptive approach, initially utilizing consistent heuristics provided by the user. After each search, the algorithm updates action costs if necessary and refines its heuristics based on the knowledge gained. This adaptability allows Adaptive A* to progressively enhance its heuristics over time, leading to a more focused exploration of the state space in subsequent searches. The algorithm's ability to adapt to changes in the environment provides a key advantage over Repeated Forward A*.

Despite the similar runtime outputs, it is essential to consider the potential scenarios where the algorithms may diverge in performance. In cases where the environment remains relatively static and the initial heuristics are well-informed, Repeated Forward A* may deliver competitive results. However, Adaptive A* is anticipated to outperform Repeated Forward

A* in dynamic environments or when there is room for improvement in the initial heuristics through experience.

The adaptability of Adaptive A*, evident in its ability to refine heuristics and adapt to changes in action costs over time, positions it as a promising choice for scenarios where the problem space is subject to variations. The consistency in runtime outputs may reflect a stable environment or highly effective initial heuristics, but it is crucial to acknowledge that Adaptive A* holds the potential for superior performance when facing evolving conditions or when continuous refinement of heuristics is beneficial. Both algorithms employ fair tie-breaking strategies, favoring cells with larger g-values and resolving remaining ties randomly.

## 7 Part 6 - Statistical Significance

*Performance differences between two search algorithms can be systematic in nature or only due to sampling noise (= bias exhibited by the selected test cases since the number of test cases is always limited). One can use statistical hypothesis tests to determine whether they are systematic in nature. Read up on statistical hypothesis tests (for example, in Cohen, Empirical Methods for Artificial Intelligence, MIT Press, 1995) and then describe for one of the experimental questions above exactly how a statistical hypothesis test could be performed. You do not need to implement anything for this question, you should only precisely describe how such a test could be performed.* Let's consider the question about comparing Repeated Forward A* and Repeated Backward A*.

To perform a statistical hypothesis test to determine whether the performance differences between Repeated Forward A* and Repeated Backward A* are systematic in nature or due to sampling noise, several steps need to be taken.

Firstly, I would define the null hypothesis (H0) and alternative hypothesis (H1). In this case, the null hypothesis could be that there is no systematic performance difference between Repeated Forward A* and Repeated Backward A*, while the alternative hypothesis is that there is a systematic performance difference between the two algorithms.

Next, I would decide on an appropriate statistical test. Since we are comparing the performance (e.g., runtime or number of expanded cells) of two algorithms, a common choice could be the t-test for independent samples. This test compares the means of two independent groups to determine whether there is a significant difference between them.

After selecting the test, I'll then gather data by running both algorithms on multiple test cases. It's important to ensure that the test cases are representative and cover a wide range of scenarios to minimize sampling bias.

Then I will calculate the test statistic (e.g., t-value) using the collected data. This statistic quantifies the difference in performance between the two algorithms. From there, I would determine the critical value or p-value threshold based on the chosen significance level (e.g., = 0.05). If the calculated test statistic exceeds the critical value or if the p-value is less than , then the null hypothesis is rejected in favor of the alternative hypothesis, indicating that there is a significant systematic difference in performance between the two algorithms.

Finally, I'll interpret the results and draw conclusions. If the null hypothesis is rejected, it suggests that the observed performance difference between Repeated Forward A* and Repeated Backward A* is not merely due to sampling noise but is systematic in nature. Possible explanations for the observed difference could include inherent differences in algorithm design or effectiveness in different problem domains.

In summary, by following these steps and conducting a statistical hypothesis test, I can determine whether the performance differences between Repeated Forward A* and Repeated Backward A* are systematic or due to sampling noise, providing valuable insights into the relative effectiveness of these algorithms.

## 8 References

1. GeeksforGeeks. "Binary Heap." *GeeksforGeeks*, 6 Feb. 2024, `www.geeksforgeeks.org/binary-heap/`.

2. Generalized Adaptive A*, `idm-lab.org/bib/abstracts/papers/aamas08b.pdf`.

3. Brownlee, Jason. "Statistical Significance Tests for Comparing Machine Learning Algorithms." *MachineLearningMastery.Com*, 8 Aug. 2019, `machinelearningmastery.com/statistical-significance-tests-for-comparing-machine-learning-algorithms/`.

4. Contribution of ChatGPT in assisting the development of the user interface for data visualization.