*Exploring Heuristic Efficiency in Pathfinding Algorithms:*

*A Comparative Study of A\* and Greedy\_BFS*

**Table of Contents**

1. **Abstract**
2. **Introduction**
3. **Methodology**
   1. Algorithm Overview
   2. Heuristic Functions
   3. Directional Weights
   4. Maze Design
4. **Implementation Details**
   1. A\* Search Algorithm
      1. Implementation of Heuristic Functions
      2. Implementation of Directional Weight
   2. Greedy Best-First Search Algorithm
      1. Implementation of Heuristic Functions
      2. Implementation of Directional Weight
5. **Experimental Results**
   1. A\* Search Results
      1. With Heuristics
      2. With directional Weight
   2. Greedy Best-First Search Results
      1. With Heuristics
      2. With directional Weight
6. **Discussion and Insights**
   1. Discussion
   2. Future Work
7. **Conclusion**
8. **References**

# 1. Abstract

In this project, I learn the logic behind Heuristic functions by a comparative analysis and visualization of two widely used heuristic-based pathfinding algorithms: A\* and Greedy Best-First Search (Greedy BFS). Both algorithms employ heuristics to guide their search towards a goal, but they differ significantly in their approach to evaluating nodes. in solving maze navigation problems. I study 3 heuristics methods i.e. **Manhattan, Euclidean, and Chebyshev**. They are applied to compare the algorithms’ performance in terms of path length, search length, and execution time. The report also analyses the impact of directional weights on both algorithms. My study through this project, also includes the understanding of impact of changing in weight values on path\_length and exploration\_length of each search algorithms under different heuristic methods. The project includes interactive visualizations created using Python libraries such as Pyamaze and Tkinter, alongside performance analysis using Matplotlib and Pandas to compare search lengths, path lengths, and exploration footprints. The results provide insights into the strengths and limitations of each algorithm in different search scenarios, contributing to a deeper understanding of heuristic-based search strategies.

# 2. Introduction

Pathfinding algorithms are essential for navigations, robotic movements, and AI game development. A\* Search and Greedy BFS are two popular algorithms for finding the path in a maze. Both use heuristics to guide their search, but their approaches differ significantly. Where A\* algorithm focus on optimal path, on the other hand, Greedy\_BFS focus on searching the path at earliest. The differences between these two algorithms are the key reasons to choose A\* and Greedy\_BFS. This project explores these differences by implementing the algorithms with multiple heuristics and directional weights.

The objectives of this study include:

1. Comparing the performance of A\* Search and Greedy BFS using Manhattan, Euclidean, and Chebyshev heuristics.
2. Evaluating the effect of directional weights on pathfinding.
3. Visualizing exploration and pathfinding processes using the pyamaze library.

Ultimately, this work contributes to the study of heuristic-based pathfinding by providing a comprehensive visual and analytical comparison of **A**\* and **Greedy BFS**. By working on this project, I able to understand how different heuristics affect the performance of these algorithms. It also insight me about the logic behind the heuristics and weights over algorithms and some answered my many questions as well.

# 3. Methodology

## 3.1. Algorithms Overview

**Let’s me introduce with the algorithms I had used in this project.**

### 3.1.1. A\* Search Algorithm*:*

A\* Search is an informed search algorithm that combines the advantages of Dijkstra's Algorithm and Greedy Best-First Search, as A\* combines the cost to reach a node (“G-cost”) and the estimated cost to the goal (“H-cost”) to determine the best path. This ensures optimal and complete pathfinding.

Total cost: T(c) = G(c) + H(c)

### 3.1.2. Greedy Best-First Search Algorithm:

Greedy BFS relies solely on the heuristic (H-cost) to prioritize nodes, often leading to faster but suboptimal solutions.

Greedy Best-First Search prioritizes nodes based on their heuristic cost; it aims to reach the goal as quickly as possible. While computationally simpler, it may not always find the optimal path but find path in fast time.

Total cost: T(c) = H(c)

## 3.2. Heuristic Functions

For this project I had used 3 heuristic functions. The heuristic function helps in estimating the distance from start(current) to goal. The calculation of this distance is different in all 3 functions. Let’s discuss it little deeper and understand each of them.

### 3.2.1. Manhattan Distance:

Manhattan Distance measures how far the two points are when you only move along the horizontal and vertical directions (no diagonal moves).

**Formula:**

Manhattan = {x1 − x2} + {y1 − y2} ; , (x1,y1) and (x2,y2) are the coordinates of the two points

**Example:**

If Point A is at (1, 2) and Point B is at (4, 6), the Manhattan distance is:

Manhattan = {1 – 4} + {2 – 6} = 3+4 = **7 units.**

A black background with white text

Description automatically generated

Code 1: Code for the Manhattan

### 3.2.2. Euclidean Distance:

It follows the straight-line distance between two points, like the shortest path between them if you could fly or walk directly. Euclidean distance calculates the "straight-line" or "as-the-crow-flies" [4]distance between two points.

**Formula:**



**Example:**

If Point A is at (1, 2) and Point B is at (4, 6), the Euclidean distance is:



A black background with colorful text

Description automatically generated

Code 2: Code for the Euclidean:

### 3.2.3. Chebyshev Distance:

This is useful when you're allowed to move in all directions (up, down, left, right, and diagonally), and it counts how far you need to go to reach the other point, taking diagonal moves into account.

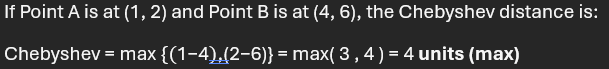
Chebyshev distance considers diagonal movement, so it measures the maximum number of steps you would need to move in any one direction to get from one point to another.

**Formula:**



Here, it picks the larger of the horizontal or vertical distance, assuming you can move diagonally.

**Example:**



**A black background with white text

Description automatically generated**

Code 3: Code for the Euclidean:

## A screenshot of a black and white screen Description automatically generated3.3. Directional Weights

Directional weights simulated variable terrain by assigning the costs to move in different directions (North, East, South, and West) and this cost will influencing while path selection. I had used 8 weight combinations to check and understand its impact on both algorithm while using all 3 heuristic functions.

0,0,0,0 is a neutral weight that means it have no effective at any heuristic function. 10 here is representing that if my agent moves in that direction then the cost of one step in that direction is 10, and -10 is vice versa. (+10) is penalty and (-10) is like a reward for an agent to move in that direction. [Visualization will be done in experimental setup: Section3.4.4]

## 3.4. The Maze Creation

A maze of predefined dimensions is created using the pyamaze library.

A black background with white text

Description automatically generated

Code 4: Create maze.py

In this project I had used 2 mazes. Both serve different purpose and use.

### 3.4.1. Maze\_1 for Heuristic\_Function.py:

Maze size 50 x 120, loopPercent = 35. This maze is created to use when A\* and Greedy\_BFS algorithms are experimenting with heuristic functions. To visualize the path length, it is important to have a little longer length so that it will be clearly visible the impact of heuristic functions over algorithms. To make it more complex, I used pattern = ‘vertical’ so that maze contain more vertical lines.

A black and white maze

Description automatically generated

**Fig 2: Maze\_1 50 x 120**

### 3.4.2. Maze\_2 for Directional\_Weight.py:

Maze size = 20 x 80, loopPercent = 30. To make algorithm process faster this maze is created having width only 20 and for proper visualization of path the length of the maze is 80. In this project I had shown only 8 directional weights but for my testing and learning I had done more than 200 test to see the effect of weight change over algorithms and how the path and search length count change according to the respective weight.

A black maze with white lines

Description automatically generated

**Fig 3: Maze\_2 20 x 80**

# 4. Implementation Details

In this section I will present you approach of my work.I have 2 algorithms, with 3 heuristic functions i.e. Manhattan, Euclidean and Chebyshev and 8 directional weights. For both the algorithms the pseudocode is somewhat similar just a logic is different, and the calculation of Heuristic functions is changed as per the use.

## 4.1. A\* Search Algorithm

A black screen with white text

Description automatically generatedFirst, necessary libraries and modules are imported. Heapq: for data structure, time: to calculate time, pyamze: for maze, tkinter: for visualization

Code 5: Necessary libraries are imported

A black background with colorful text

Description automatically generatedIn all A\* Code, different heuristic functions are used, this one is for Manhattan heuristic function.

Code 6: Heuristic function is added

***A screen shot of a computer code

Description automatically generatedget\_next\_cell*** function is created to move my agent to next cell on the direction (E, W, S, N) and check for wall presence (1,0). If any direction allows the moment of agent, means (no wall), then it returns to new position (next cell) and if the direction has restricted by wall, then it moves back to current position.

Code 7: get\_next\_cell function

***A screen shot of a computer

Description automatically generated***Code 8: Function for A\* search algorithm logic-I

For heuristic method here it is written as Manhattan heuristic and for others the heuristic function will changed.

A computer screen shot of a program code

Description automatically generated

Code 9: Function for A\* search algorithm logic-II

The A\_star\_search function runs the A\* algorithm by adding the starting position to a priority queue with its estimated cost. Then, explores neighboring cells. For each valid move, it calculates the cost and adds it to the queue if it's a better path. The search continues until the goal is reached. Afterward, it traces the path back from the goal to the start and returns the exploration order, visited cells, and the path to the goal. This exploration\_order visited and path\_to\_goal is needed by my agents to trace the path and for proper visualization.

A computer screen shot of a program code

Description automatically generated

Code 10: Tkinter code part

This code is written for visualization of result(data) by using tkinter. Although I can also use inbuild function of pyamaze module, which show the data on maze itself but this one look good, clearly visible and understandable as well. (Will show this window in result section.)

A screen shot of a computer program

Description automatically generated

Code 11: Main function to run and implement the whole logic

This code is the main function that runs the A\* search algorithm on a maze. It starts by creating a maze (50 x 120) and loading it from a CSV file i.e. ***Maze\_1 for Heuristic\_Function.csv***

The goal position is (1 ,1), and the A\* search is executed, measuring how long it takes. The function then calculates the path and exploration lengths.

Next, it sets up three agents:

A black background with white text

Description automatically generated

Finally, the heuristic information (like Manhattan distance) is displayed in a separate window and as text labels in the maze. The maze simulation is then run.

In a same way I design 3 python files for A\* Search Algorithm:

A black background with white text

Description automatically generated

For A\* algorithm these files cover the same logic, but the difference is about the heuristic function which I will explain below. In addition, there is one more python file that is



A computer screen shot of numbers

Description automatically generatedwhich help to visualize the goal path of all three files at one place.

This code is used only when I am doing experiments on directional weight. These weight (10) are changed and then by changing then the exploration and path lengths will be change which I will show you in **result section.**

Code 12: Directional Weight function

A screenshot of a computer screen

Description automatically generatedThis code is under the main function which run all heuristic functions with color coding.

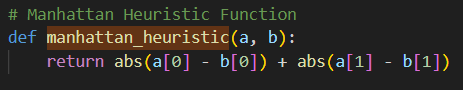
Code 13: Heuristic run

A computer screen with text on it

Description automatically generatedThis code will help to run all heuristic functions in A\* algorithm**.** This will trace the path to goal for each heuristic function. it performs the search and gets the exploration order, visited cells, and the path to the goal.

Then, it traces the optimal path for each heuristic, using different colors for each. The path length for each heuristic is also stored for later use. Code 14: Trace path for Heuristic in A\*

### 4.1.1. Implementation of Heuristic Functions

So, as I already mentioned, the code for all heuristic functions are different, else other things are same for all 3 files mentioned above. Let me show you, how and where I had used heuristic functions in all 3 files for A\* algorithm.

Code 15: Heuristic Implement Code-II

A screen shot of a computer

Description automatically generated

Code 16: Heuristic Implement Code-II

A screen shot of a computer

Description automatically generated

Code 17: Heuristic Implement Code-III

Code 1: Manhattan heuristic function calculates the Manhattan distance between two points.

Code 2: Call Manhattan heuristic function *in A\* search algorithm* that takes a maze, start, and goal positions, and uses a heuristic function to find the shortest path.

Code 3: Runs the A\* algorithm using the heuristic function to find the path from the start to the goal, and it returns the order of exploration, visited cells, and the path to the goal.

### 4.1.2. Implementation of Directional Weight

For **Directional Weight**, I created another 3 python files for A\* Search Algorithm which show the effect of change in weight on path length from start to goal, search length of algorithm and the graphical representation of the result.

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

Code 18: Implementation of Directional Weight Code-I

A screen shot of a computer code

Description automatically generated

Code 19: Implementation of Directional Weight Code-II

A screenshot of a black board with white text

Description automatically generatedThe directional\_weights is a dictionary, that assigns a cost of 10 to each direction (North, East, South, and West). This means that moving in any of these directions will cost 10 units. The second code uses these weights to calculate the cost of moving from one cell to the next. When exploring a new cell, it adds the appropriate cost based on the direction of movement to the current path cost.

These 10 units is same for all but for the experiment purpose I had used 8 different directional weights. That are as follow:

## 4.2. Greedy Best-First Search Algorithm

Other things are same in greedy\_BFS except the implementation part. So, lets quicky summarize it here.

A screen shot of a computer

Description automatically generated

Code 20: Function for Greedy\_BFS algorithm logic

The greedy\_bfs\_search function implements the Greedy\_BFS algorithm using the Manhattan heuristic. In a same way the other heuristic functions are implemented in the greedy\_BFS algorithm. It starts by setting the start and goal positions. Then, it uses a ***priority queue*** to explore the maze, prioritizing cells based on their estimated distance to the goal (calculated by the heuristic). The algorithm explores neighboring cells in four directions (North, East, South, West), and for each valid move, it adds the new cell to the queue if it hasn't been explored. Once the goal is reached, it reconstructs the path from the start to the goal by tracing back through the visited cells. The function returns the exploration order, visited cells, and the path to the goal. The other part of this algorithm is quite similar to A\* Algorithm only so let’s move on next part about the implementation of heuristic functions and directional weights.

### 4.2.1. Implementation of Heuristic Functions

The implementation of each heuristic function in greedy\_BFS is also same as A\* Algorithm. The logic of using the h(n) is different which I had explained in **section 3.2.**I design 3 python files for A\* Search Algorithm:

**2.1 Greedy\_BFS Manhattan Heuristic.ipynb**

**2.2 Greedy\_BFS Euclidean Heuristic.ipynb**

**2.3 Greedy\_BFS Chebyshev Heuristic.ipynb**

For greedy\_BFS algorithm these files cover the same logic, but the difference is about the heuristic function which I will explain below. In addition, there is one more python file that is

**2.4 Greedy\_BFS Combine Visualization of goal path.ipynb**

This file is used to visualize the goal path of all three heuristic functions at one place.

### 4.2.2. Implementation of Directional Weight

For **Directional Weight**, I created another 3 python files for A\* Search Algorithm which show the effect of change in weight on path length from start to goal, search length of algorithm and the graphical representation of the result.

**A black background with white text

Description automatically generated**

A black background with white text

Description automatically generated

Code 21: Implementation of Directional Weight Code-I

A screen shot of a computer code

Description automatically generated

Code 22: Implementation of Directional Weight Code-II

The direction weight of greedy\_BFS is same as A\* Algorithm.

# 5. Results

Both algorithms are divided into 2 parts for result.

1. Impact of ***Heuristic function*** on search algorithm.
2. Impact of change in ***Directional weight*** on search algorithm.

## 5.1. A\* Search Algorithm Performance

### 5.1.1. With Heuristics

* Manhattan Heuristic Function:

A maze with blue lines

Description automatically generated

Fig 5: Manhattan Heuristic function in A\* algorithm

* Euclidean Heuristic Function:

A maze with blue and white lines

Description automatically generated

Fig 6: Manhattan Heuristic function in A\* algorithm

* Chebyshev Heuristic Function:

A maze with blue lines

Description automatically generated

Fig 7: Manhattan Heuristic function in A\* algorithm

Visualization of path length for all 3 heuristic functions.

A black and yellow maze

Description automatically generated

Fig 9: Visualization of all Heuristic Functions

This visualization shows that all heuristic functions give same output in terms of path\_length.

A black and white table with numbers

Description automatically generated

Table 1: Tabular Comparison of Heuristic function in A\* Algorithm metrics

The table clearly shows the execution time of Manhattan function is more than other functions, also the search length is also more for Manhattan, but the path\_length is same for all heuristic functions.

### 5.1.2. With directional Weight

A black and white table with numbers

Description automatically generated

Table 2: A\* search results with different Directional Weights for each heuristic

The path\_length is similar for all heuristics when the weights are balanced (e.g., 1075 or 1187), but the search\_length varies. For example, with weights like (0, 10, 10, 0), all heuristics have the same path\_length (1075) and search\_length (9090). Some weight combinations result in "NA" values, indicating invalid or blocked paths. In general, the search\_length could to be higher when the directional weights vary or are imbalanced, and the path\_length is consistent for most scenarios.

## 5.2. Greedy BFS Algorithm Performance

### 5.2.1. With Heuristics

* Manhattan Heuristic Function:

A maze of lines and a red and blue sky

Description automatically generated with medium confidence

Fig 10: Manhattan Heuristic function in Greedy\_BFS algorithm

* Euclidean Heuristic Function:

A maze with different colored lines

Description automatically generated

Fig 11: Euclidean Heuristic function in Greedy\_BFS algorithm

* Chebyshev Heuristic Function:

A maze with different colored lines

Description automatically generated

Fig 12: Chebyshev Heuristic function in Greedy\_BFS algorithm

A maze with different colored lines

Description automatically generated

Fig 13: Visualization of path length for all 3 heuristic functions.

A black and white table with numbers

Description automatically generatedThis visualization shows that all heuristic functions give same output in terms of path\_length.

Table 3: Tabular Comparison of Heuristic function in A\* Algorithm metrics

The table clearly shows the execution time of Manhattan function is more than other functions, also the search length is also more for Manhattan, but the path\_length is same for all heuristic functions.

### 5.2.2. With directional Weight

A black and white table with numbers

Description automatically generated

Table 4: A\* search results with different Directional Weights for each heuristic

The path\_length is similar for all heuristics when the weights are balanced (e.g., 1075 or 1187), but the search\_length varies. For example, with weights like (0, 10, 10, 0), all heuristics have the same path\_length (1075) and search\_length (9090). Some weight combinations result in "NA" values, indicating invalid or blocked paths. In general, the search\_length could to be higher when the directional weights vary or are imbalanced, and the path\_length is consistent for most scenarios.

# 6. Discussion & Future Work

This project compared the A\* Search and Greedy Best-First Search (Greedy BFS) algorithms using different heuristics (Manhattan, Euclidean, and Chebyshev) and directional weights. Let’s have a final discussion about the project and future work I can add in this.

## 6.1 Discussion

**1. Heuristic Performance:** All heuristics found valid paths, but the **Manhattan heuristic** often led to longer search times and path lengths. Manhattan has some restricted moment in directions that cause it a little slower in execution and it gives a longer path, on the other hand **Euclidean** and **Chebyshev** heuristics generally performed faster, but they didn’t always find the shortest path. With my experiment I had found many times Manhattan perform well due to maze structure.

**2. Directional Weights:** Adding directional weights helped me to understand it role. A\* handled these variations well, while Greedy BFS has no impact on this due to this greedy behaviour towards reaching goal. In addition, for learning purpose I had done more than 200 experiments to understand the behaviour of this weight change and some time when I don’t get any result with some weight i.e. NA in some situations, I first though it is because of some error but thanks to some reference paper, google and ChatGPT, I understood it is a because the nature of maze sometime doesn’t match with the moment of agent with specific weight conditions. E.g. (0, -10, -10, 0).

**3. Visualization:** The use of **Tkinter** helped me to visualize the result about how each algorithm worked, making it easier to understand their differences in behaviour. The tabular form or getting matrix is nice way to sort out difference easier.

## 6.2 Future Work

In the future, several improvements can be made

1. **Dynamic Maze Generation** could simulate moving obstacles for real-time testing.
2. **Machine learning-based heuristics** could be explored to improve the algorithms.
3. **Parallel algorithms** could speed up searches in large mazes.
4. **Testing other algorithms**, like Dijkstra’s or Bidirectional Search, could provide more insights.
5. **Real-world applications** such as robot navigation could be explored for practical use cases.

# 7. Conclusion

This project analyzed the A\* Search and Greedy BFS algorithms in maze navigation, using different heuristics and directional weights.

1. *A Search*\* was more reliable for finding the best path, but it took longer, especially with the Manhattan heuristic.
2. **Greedy BFS** was faster but sometimes found suboptimal paths, particularly in complex mazes.

The project highlighted how heuristic choices, and directional weights impact the performance, and it also showed how these algorithms can be applied to different environments.

In conclusion, I enjoyed and learned a lot from this project. The A\* and Greedy\_BFS both algorithms have their strengths and weaknesses, they provide useful insights into how pathfinding works in various scenarios.

# 8. References

1. Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach.
2. Stentz, A. (1995). The focused D\* algorithm for real-time replanning.
3. pyamaze Documentation. Available at: <https://github.com/MAnwarHussain/python-amaze>.
4. <https://stackoverflow.com/questions/47485510/how-to-calculate-heuristic-value-in-a-algorithm>