Assignment 2A – Tree Based Search

Name	StudentID	Contribution
Denver J Cope	104738758	**Programming**: ASTAR, CUS1, GUI Reporting: Astar algorithm explanation, testing, Insights
Ananda Pathiranage Ruveen Thathsilu Jayasinghe	104317649	**Programming**: DFS, BFS, GBFS, CUS1, Graph.py, Search.py, GUI **Reporting**: DFS, BFS, GBFS, CUS1 algorithm explanation, testing, Insights and Research section
Fayiz Kallupalathingal	104658733	**Programming:** GUI Created test case test4.txt and test5.txt **Reporting:** Introduction
Rahat Alam	103810105	**Programming:** Cus2, GUI **Reporting:** CUS2 algorithm explanation, testing, Insights

Link to project repository: https://github.com/ruvxn/Pathfinder_AI

Table Of Contents

Introduction

This assignment is based on a Route-finding problem. The graph is in a hierarchical structure starting from the root node and branching out. While finding the route a search tree is created from the origin node and expands step by step. The graph includes a set of nodes connected by edges, each node represents a location in the graph and edges specify the cost of traverse from one node to another. The aim of this assignment is to create agents that can find the optimal route from the source node to the destination node. To solve this, our team had made use of standard search algorithms such as DFS (Depth-First Search), BFS(Breadth-First Search), GBFS (Greedy Best-First Search), A*(A-star Search) [1] .On top of that we have implemented two custom search algorithms developed from scratch.

Instructions

Instructions on how to use the program:

1. Open file directory in Terminal or Command Prompt.
2. Use the command structure to run tests:

   Python search.py <filename> <method>

   Example:

   Python search.py tests/test3.txt DFS
3. Check the output:

   If the program finds a path, the output will be,

   <filename> <method>

   Goal number_of_nodes_expanded

   path

   If the program doesn't find a path, the output will be,

   No path found

Instructions on how to run the Graphical version of the program:

1. In the terminal or command prompt navigate to the project directory and type ***"pip install -r requirements.txt"*** or manually install ***pygame***.
2. Use the command:

   python GUI.py <filename> <method>
3. Press the "enter/return" key for execution as prompted

Algorithms

Find pseudocodes attached in the pseudocodes folder for further breakdown.

1. **Depth First Search - DFS**

   Depth First Search (DFS) algorithm is an uninformed search method that follows the concept of reaching the maximum depth of a path in the map. Then takes steps back to find alternative paths. The main goal of DFS here is to find one of the destination nodes, starting from the designated origin node and expanding throughout a single path fully and then only trying other paths. DFS will repeat these steps until a destination is found or there are no more paths to discover.

2. **Breadth First Search - BFS**

   Unlike DFS, BFS explored all nodes at the current level prior to expanding any deeper. BFS guarantees the shortest path in terms of steps. In this project context, the main goal for BFS method is to start from origin node and visit all of the neighbor nodes level by level until a destination node is met.

3. **Greedy Breadth First Search - GBFS**

   GBFS is different from DFS and BFS as it is an informed search method. It considers heuristics to find the nodes based on closeness to the destination. In this project context the goal for GBFS is to find the neighbor closest to the destination based on heuristic (h(n)) value until the destination is met.

4. **ASTAR**

   A* builds upon Dijkstra's algorithm by introducing Heuristic weight, this rewards movement towards the goal node. Instead of exploring all possible nodes uniformly, A* prioritizes paths that appear closer to the destination, significantly improving efficiency.

   This is achieved using Euclidean distance (Straight-Line Distance) as a heuristic estimate for each node's proximity to the goal node. While this estimate doesn't account for obstacles/detours (like how a crow would fly directly to its target), it effectively guides the search towards the node without unnecessary exploration.

5. **Custom 1 – UCS**

   Uniform Cost search is implemented as Custom 1 algorithm. Its main goal is to find the path with lowest total cost. This is done using ***priorityqueue*** to expand to the cheapest cost path (Fleck, 2021**)** [1].

6. **Custom 2**

   Instead of concentrating on the total cost, the CUS2 algorithm is an informed search algorithm that seeks to arrive at the destination with the fewest number of moves. It employs a priority queue in which nodes are ranked according to step count, insertion order (with a counter for breaking ties), and, as a heuristic, Euclidean distance to the closest goal node.

   For every expansion, CUS2:

   1. determines whether the current node is a destination. To prevent going back, mark nodes as visited.
   2. investigates neighboring nodes that have been sorted, determines their heuristic, and modifies the priority queue.
   3. keeps the number of moves to a minimum while guaranteeing effective exploration towards the objective.

Features/Bugs/Missing

1. Depth First Search

Features:

- Since Depth First Search follows a last in first out structure, a ***stack*** was chosen as the best option for implementation.
- ***Visited set*** *is* utilized to avoid unnecessary loops or cycles where the search algorithm might cycle over and over through the same path.
- ***Sorted*** is used to achieve the requirement for chronological order when all else is equal.
- Use of ***Starmap*** and ***Lambda*** was by personal choice to streamline the code.

Bugs: No bugs were identified; the program simulated the behavior of DFS algorithm when executed (See DFS_test in testing for further clarification).

Missing: Could apply debugging prints to see functionality.

2. Breadth First Search

Features:

- Since BFS follows a first in first out structure, ***deque*** was used as it allows to remove from both the sides of a structure.
- ***Visited*** and ***Sorted*** implemented similar to DFS explained above.

Bugs: No bugs were identified; the program simulated the behavior of BFS algorithm when executed (See BFS_test in testing for further clarification).

Missing: Could apply debugging prints to see functionality.

3. Greedy Breadth First Search

Features:

- Informed search utilizing the ***Euclidean distance***. The choice of using Euclidean distance came with the understanding of graph traversal pattern (Sahani, 2020)[2] based on assignment diagram.
- Use of ***PriorityQueue*** for prioritizing lower heuristic values (h_score) for expansion.
- ***Visited*** and ***Sorted*** implemented similar to DFS explained above

Bugs: No exceptions made for when destination nodes might not be mentioned which will possibly raise an error during execution.

Missing: Could improve to calculate the h(n) to the closest goal dynamically at each node.

4. ASTAR

Features:

- Utilizes priority queue and sorts the Priorityqueue by estimating the distance using Euclidean distance, A* is optimal if the heuristic value never overestimates the true cost
- Unlike other algorithms, A* is both fast and accurate, saving time on not searching nodes that aren't in the direction of the goal (unlike DFS, BFS etc)

Bugs:

- Currently ASTAR doesn’t work with tests 1 and 4 however works for all others.

Missing:

- Currently missing a tiebreaker, if the heuristic values are equal there is no set way to break this tie.

- GUI, no graphical interface has been implemented so far.

5. Custom 1 – UCS

Features:

- Gives uniform but optimal path cost
- Use *PriorityQueue* to get the lowest total path cost.
- *Visited* and *Sorted* implemented like DFS explained above

Bugs: No major bugs noticed.
Missing: No major missing parts based on understanding.

6. Custom 2

Features:

- Ranks paths with less moves in a priority queue.
- Tie-breaker carried out in line with expansion order using insertion counters.
- Search towards the goal is guided by dynamic Euclidean distance heuristic.
- Neighbours arranged in order guarantee ascending node expansion.

Bugs:

- Not found any significant flaws; algorithm works as expected.
- For exact step-by--step tracking, add debug print statements.

Missing:

- Focuses just on achieving the first goal; does not manage several goals in turn.
- Beyond insertion order, there is no tiebreaker for equal heuristic values.

7. Graphical User Interface:

Features:

- Visualise node travel and final path using Pygame.
- Dynamic update of node colours: visited nodes, current paths, destination.

Bugs:

- When running test files with bi-directional edges, the final path colour is not shown properly (DFS_test).
- When running test files with over lapping nodes it is not visualised clearly to interpret two paths as different (example: CUS1_test path 2 – 4 and 2 – 6)

Missing**:** Meets expected requirements

Testing

1. DFS_test.txt

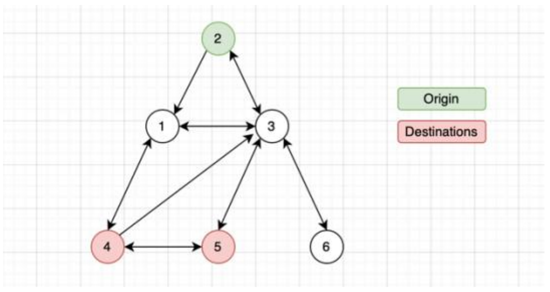


Figure 1: DFS_test graph structure

Reasoning behind the test design:
- Multiple paths between origin and destinations to test deep traversal.
- Dead ends to test back tracking.
- Testing sorted, visited and edge notation handling.*Refer to insights (DFS_test) for result breakdown.*

2. BFS_test.txt

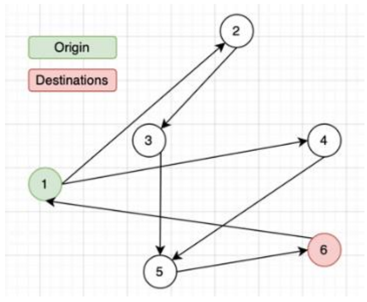


Figure 2: BFS_test graph structure

Shortest path (Expected Output) = 1,4,5,6

Reasoning behind the test design:
- Levels expansion testing with multiple neighbors.
- Testing the shortest path from origin to destination.
- Testing sorted, visited and edge notation handling.

Refer to insights (BFS_test) for result breakdown.

3. GBFS_test.txt

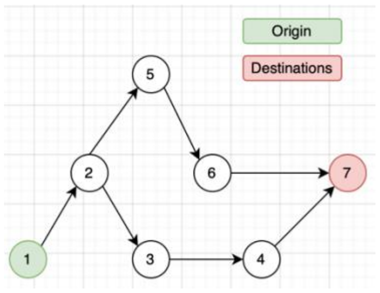


Figure 3: GBFS_test graph structure

Reasoning behind the test design: [2]
- Test heuristic-based decision by testing if it takes visually closest regardless of path cost.
- Non-linear node placement to test varying Euclidean distance.

Refer to insights (GBFS_test) for result breakdown.

4. ASTARTest.txt

This test was created in an attempt to trick the algorithm, by offering a path in which the Euclidean distance would be lower, however once this path was actually explored, it would show that it was actually not the shortest path. Resulting in it doubling back. (Green = the "trap" path, Blue = shortest path, 1 = origin, and 4 being the goal node)

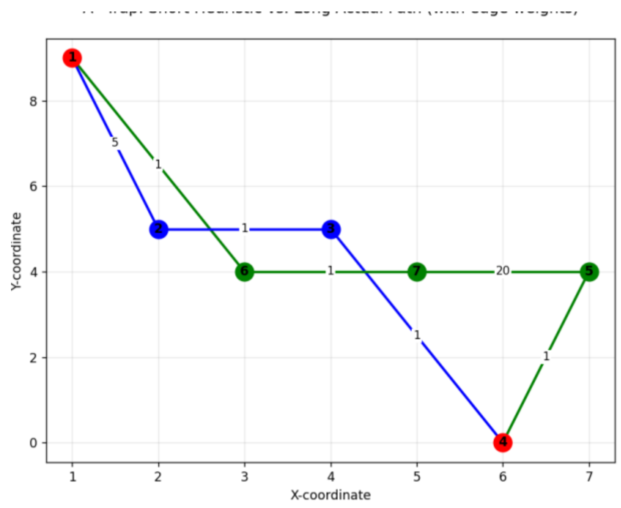


Figure 4: ASTAR test graph

5. CUS1_test.txt

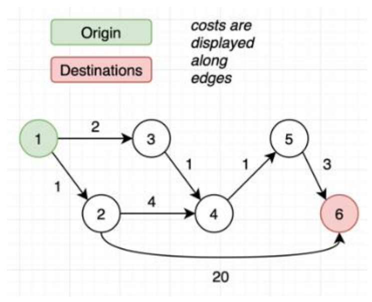


Figure 5: CUS1_test graph structure

Reasoning behind the test design:

- Provides two paths to destination,
  1. The first path is 1 – 3 – 4 – 5 – 6
  2. The second 1 – 2 – 6

There is a huge difference in number of steps as well as the cost of each path. The cost of the less steps path (Path 1) is 21, while the cost of the longer path is 7. This will test if UCS chooses the low-cost path over short path.

Refer to insights (CUS1_test) for result breakdown.

6. CUS2test.txt

This test aimed to check whether CUS2 is prioritizing paths by the number of moves rather than the total cost.
The graph includes a number of paths to the goal node; some of them are a bit more costly but shorter in the number of steps, and others are less costly overall but require more steps. The final path is highlighted in orange, the goal node in green, the start node in red, and the visited nodes in yellow.The aim is to verify that CUS2 successfully chooses the shortest path, avoiding longer alternative paths, even if they are less expensive, and expands nodes in step count order.

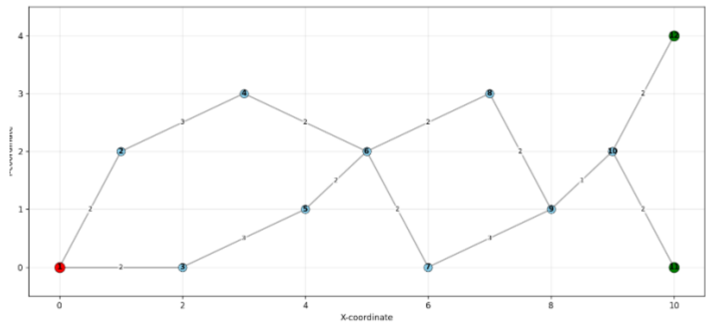
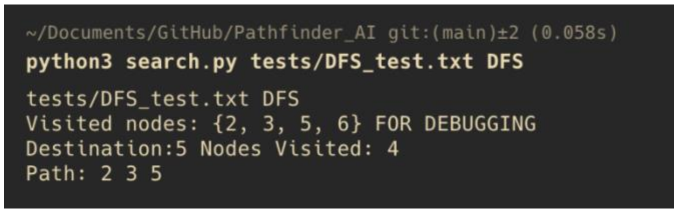


Figure 5: CUS1_test graph structure

Insights

1. **DFS Insights**



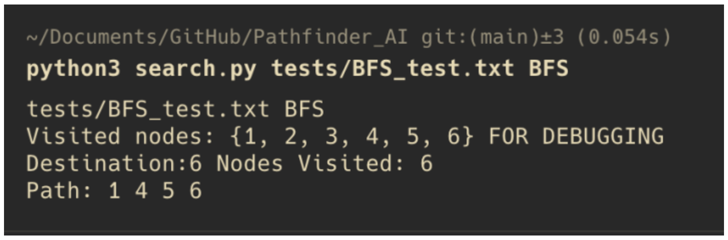
```
~/Documents/GitHub/Pathfinder_AI git:(main)±2 (0.058s)
python3 search.py tests/DFS_test.txt DFS

tests/DFS_test.txt DFS
Visited nodes: {2, 3, 5, 6} FOR DEBUGGING
Destination:5 Nodes Visited: 4
Path: 2 3 5
```

Figure 6: DFS_test output

Evaluating the DFS method implementation using the DFS_test test file uncovered that, DFS method follows the depth first traversal as it went a level deeper after reaching the goal. DFS is suitable to be used when the destination is deep within the graph and any path is acceptable as long as it reached the destination.

2. **BFS Insights**



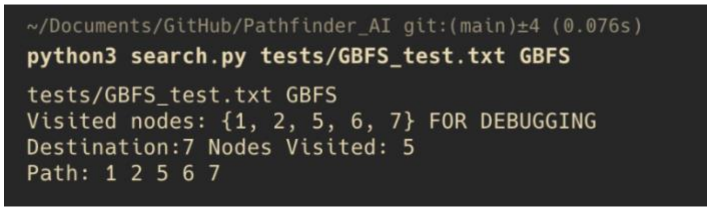
```
~/Documents/GitHub/Pathfinder_AI git:(main)±3 (0.054s)
python3 search.py tests/BFS_test.txt BFS

tests/BFS_test.txt BFS
Visited nodes: {1, 2, 3, 4, 5, 6} FOR DEBUGGING
Destination:6 Nodes Visited: 6
Path: 1 4 5 6
```

Figure 7: BFS_test output

Evaluating the BFS method implementation using the BFS_test test file uncovered that, BFS method follows a layered structure of expansion visiting all neighbors at a level before moving further to other levels. BFS method successfully returned the shortest path to the destination. Since BFS explores all nodes at one level it might be high time consuming when the number of nodes is high. Hence, the best place for BFS is where there are less nodes, but you require the shortest path.

3. **GBFS Insights**



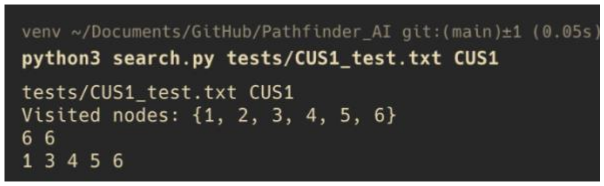
```
~/Documents/GitHub/Pathfinder_AI git:(main)±4 (0.076s)
python3 search.py tests/GBFS_test.txt GBFS

tests/GBFS_test.txt GBFS
Visited nodes: {1, 2, 5, 6, 7} FOR DEBUGGING
Destination:7 Nodes Visited: 5
Path: 1 2 5 6 7
```

Figure 8: GBFS_test output

Evaluating the GBFS method implementation using the GBFS_test test file uncovered that,GBFS followed the nodes that appeared to be closest to goal based on Euclidean distance. It ignored the alternative path which had higher heuristic values like the step from node 4 straight to node 7. Based on this result, GBFS is more suitable for situations where shorted path by cost is not required, and fast solution is favorable over cost effectiveness.

4. Custom 1 Insights



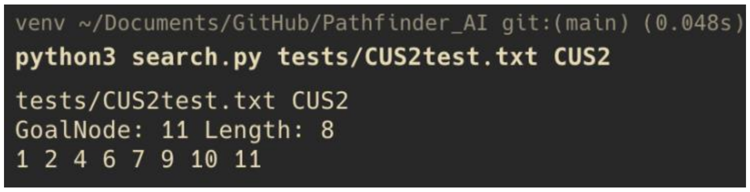
```
venv ~/Documents/GitHub/Pathfinder_AI git:(main)±1 (0.05s)
python3 search.py tests/CUS1_test.txt CUS1

tests/CUS1_test.txt CUS1
Visited nodes: {1, 2, 3, 4, 5, 6}
6 6
1 3 4 5 6
```

Figure 9: CUS1_test output

When comparing the results to the discussion in the testing section, we can confirm that UCS chose the low cost, more steps path over high cost, less steps path. Therefore, UCS used in custom1 algorithm is best suited for taking the minimum cost path, which could correlate to distance of road in a map, energy consumption of different paths and time consumption.

5. Custom 2 Insights



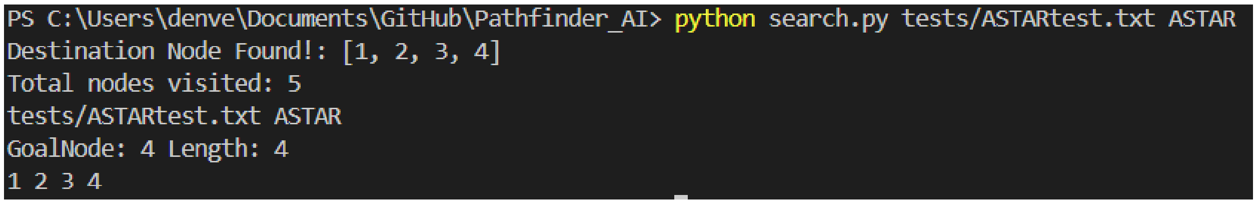
```
venv ~/Documents/GitHub/Pathfinder_AI git:(main) (0.048s)
python3 search.py tests/CUS2test.txt CUS2

tests/CUS2test.txt CUS2
GoalNode: 11 Length: 8
1 2 4 6 7 9 10 11
```

Figure 10: CUS2test output

Testing Cus2 shows that the algorithm efficiently favors routes that need the fewest moves to get to the destination node. Across multiple test case, Cus2 consistently expanded nodes in step count order while using the heuristic to keep the system moving in the direction of the goal. This allowed it to avoid longer paths and unnecessary node expansions. It is useful in situations where finding the goal quickly (in steps) is more important than the total cost of the path, even though it does not ensure the lowest total cost.

6. ASTAR Insights



```
PS C:\Users\denve\Documents\GitHub\Pathfinder_AI> python search.py tests/ASTARtest.txt ASTAR
Destination Node Found!: [1, 2, 3, 4]
Total nodes visited: 5
tests/ASTARtest.txt ASTAR
GoalNode: 4 Length: 4
1 2 3 4
```

Figure 11: ASTAR output

Testing using the ASTARTtest.txt shows that the program works as intended, it first tries the path that has the best estimate to get it closest to the node(using Euclidean distance) however once it goes down the "trap" node it sees that this will not be the shortest path, and results in it trying the alternative path. This can be seen in the" total nodes visited" is higher than the final path length.

Research

Through group discussion and confirming with class tutor a dynamic visual interpretation of the node traversal algorithm was selected as research material for this assignment. This was achieved using ***pygame*** due to its familiarity among team members [3]. Pygame suited perfectly with this project due to simple to implement minimal graphics, and the real time update capabilities, that allowed us to implement the graph traversal animations as well as showing the actual decision-making processes.

Taking inspiration from previous projects [3] as well as tutorials [4] we implemented animations that would showcase the color changes that contrasts the decision-making process. A bug that we encountered during execution was that when the program was executed the animation happened quicker than the pygame pop-up window opened, which created an issue for viewing the animation. Hence, to fix that we implemented a user input for traversal [4].

For execution of this program use the format explained in the instructions section.

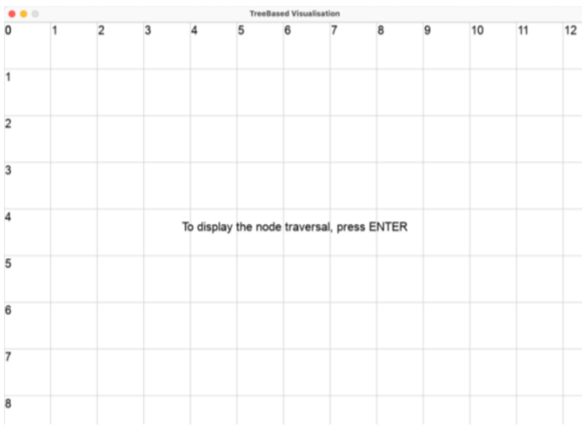


Figure 12: The start screen

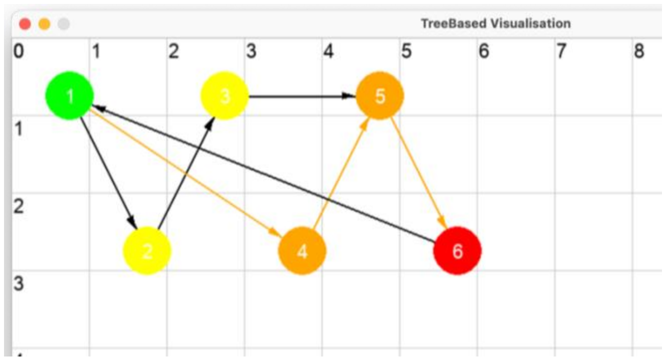


Figure 13: The BFS execution on GUI
Green – Origin
Red – Destination
Yellow – Travelled Nodes
Orange – Path

Acknowledgment and Resources

[1] **AIMA-Python –** Inspiration and understanding for algorithm implementation.
Link to AIMA-Python repository: https://github.com/aimacode/aima-python

[2] **ChatGPT** – Brainstorming best test case for testing GBFS functionality.

Prompt: "what should I consider when I'm creating a test file to test the GBFS algorithm functionality in a node traversal graph program that has origin, nodes with edges connecting them, and destinations"
Response resourced: "Since GBFS uses the heuristic to make decisions:
Nodes should be placed non-linearly so that Euclidean distances vary.
Ensure that shortest-cost path ≠ lowest heuristic path, to see if GBFS only chooses based on heuristic."

[3] **pygame** – previous pygame experience and used fundamental structures utilized.
Link to previous project repository: https://github.com/ruvxn/path_finder

[4] **Tutorials and Projects –** Design and implementation inspirations
Link to repository: https://github.com/techwithtim/A-Path-Finding-Visualization - color coding inspirations.
Link to video: https://youtu.be/JtiK0DOeI4A?si=2DXjl-8sIsoBuvdl - User key input for execution

References

[1] Fleck, M. (2021). *Uninformed Search 4*. CS 440/ECE 448: Artificial Intelligence. University of Illinois at Urbana-Champaign. https://courses.grainger.illinois.edu/cs440/fa2021/lectures/search4.html
[2] Sahani, G. R. (2020, July 24). *Euclidean and Manhattan distance metrics in machine learning*. Medium. https://medium.com/analytics-vidhya/euclidean-and-manhattan-distance-metrics-in-machine-learning-a5942a8c9f2f
[3] Reducible – Understanding the A* algorithm Indepth https://www.youtube.com/watch?v=88I6IidylGc
[4] Nicolas Swift (2017) Easy A* pathfinding. https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2
[5] GeeksforGeeks, "Python itertools.count() function", *GeeksforGeeks*, Nov. 17, 2022. [Online].Available: https://www.geeksforgeeks.org/python-itertools-count/. [Accessed: 13-Apr-2025].
[6] Python Software Foundation, "queue — A synchronized queue class", *Python 3.12.3 documentation*, Apr. 8, 2024. [Online]. Available: https://docs.python.org/3/library/queue.html. [Accessed: 13-Apr-2025].