Tree Based search

|  |  |  |  |
| --- | --- | --- | --- |
| **Family Name(s)** | **Given Name(s)** | **Student ID** | **Signature** |
| Dingyi | Wang | 104661346 |  |
| Mao | Mingdi | 104511658 |  |
| Suman | Sutparai | 105106819 |  |
| Manak | Singh | 104995074 |  |

*No code was copied or directly reused from external sources. All code was independently written by our team members. External resources were only used for conceptual reference and understanding.*

# Table of contents

[Table of contents 2](#_Toc195477184)

[Instruction 4](#_Toc195477185)

[DFS 4](#_Toc195477186)

[BFS 4](#_Toc195477187)

[GBFS 4](#_Toc195477188)

[AS 4](#_Toc195477189)

[CUS1 5](#_Toc195477190)

[CUS2 5](#_Toc195477191)

[Test Code 6](#_Toc195477192)

[Introduction 6](#_Toc195477193)

[DFS 8](#_Toc195477194)

[BFS 9](#_Toc195477195)

[GBFS 10](#_Toc195477196)

[AS 10](#_Toc195477197)

[CUS1 12](#_Toc195477198)

[CUS2 14](#_Toc195477199)

[Features/Bugs/Missing 14](#_Toc195477200)

[DFS 14](#_Toc195477201)

[BFS 15](#_Toc195477202)

[GBFS 15](#_Toc195477203)

[AS 16](#_Toc195477204)

[CUS1 16](#_Toc195477205)

[CUS2 17](#_Toc195477206)

[Test code 18](#_Toc195477207)

[Testing 18](#_Toc195477208)

[Insights： **Comparative Analysis of Search Algorithms** 21](#_Toc195477209)

[ **Informed vs Uninformed Search: A Foundational Distinction** 21](#_Toc195477210)

[ **1. Breadth-First Search (BFS)** 21](#_Toc195477211)

[ **2. Depth-First Search (DFS)** 22](#_Toc195477212)

[ **3. Greedy Best-First Search (GBFS)** 22](#_Toc195477213)

[ **4. A\* Search** 22](#_Toc195477214)

[ **5. CUS1 (Uninformed, IDDFS-inspired)** 22](#_Toc195477215)

[ **6. CUS2 (Informed, Custom A\* Variant)** 23](#_Toc195477216)

[Research (if applicable) 23](#_Toc195477217)

[ **2. Breadth-First Search (BFS)** 24](#_Toc195477218)

[ **3. Greedy Best-First Search (GBFS)** 24](#_Toc195477219)

[ **4. A\* Search** 24](#_Toc195477220)

[ **5. Iterative Deepening Depth-First Search (IDDFS) – *CUS1*** 25](#_Toc195477221)

[ **6. Step-Count-Based Heuristic Search (CUS2)** 25](#_Toc195477222)

[Conclusion 25](#_Toc195477223)

[Acknowledgements/Resources 25](#_Toc195477224)

[References 27](#_Toc195477225)

# Instruction

## DFS

Use the DFS.py Python script in any terminal or command-line interface that supports Python 3 to run the Depth-First Search (DFS) algorithm. Because the structure of the graph—which consists of nodes and their connections—is predefined within the code, the script is self-contained and doesn't require any additional input files or command-line options. When the script is run, a stack is started with the initial node and a directed graph is constructed using a dictionary-based structure. After that, it makes a depth-oriented exploration of the graph, going as far as it can along each branch before turning around. The DFS function keeps track of the number of nodes created throughout the traversal and returns the first goal node it encounters, along with the total number of nodes generated and the full path from the start node to the goal.

## BFS

To run the Breadth-First Search (BFS) algorithm, simply execute the *BFS.py* Python script in a terminal or command-line interface that supports Python 3. The program does not require any additional input files or command-line arguments, as all node and edge information is predefined within the script. Upon execution, the script constructs a directed graph using a dictionary-based data structure and initializes a queue with the origin node. It then proceeds to explore the graph in a level-order manner, expanding nodes layer by layer until it reaches one of the specified destination nodes. The BFS function tracks the number of nodes it creates during the search process and returns the first goal it reaches, along with the total number of nodes created and the path from the origin to the goal.

## GBFS

To use the Greedy Best-First Search (GBFS) algorithm, run the *GBFS.py* script in a Python 3 environment. Like the BFS implementation, this program is fully self-contained and does not require external input files. It defines a graph with directed edges and associated costs, along with a set of nodes each represented by 2D coordinates. When executed, the program initializes a priority queue and uses a Euclidean-distance-based heuristic function to estimate how close each node is to the goal. At every step, it expands the node that appears closest to any destination based on this heuristic value. The algorithm continues until one of the goal nodes is reached, and it returns the identified goal node, the number of nodes created during the process, and the path taken.

## AS

To use the A\* Search (AS) algorithm, just run the AS.py Python script in a Python 3 compatible terminal or command-line interface. There is no need for external files or inputs because the script is self-contained, with all node connections and heuristic values established within the code. To manage node exploration based on the lowest estimated total cost (f(n) = g(n) + h(n)), where g(n) is the cost to reach a node and h(n) is the heuristic estimate to the target, the script builds a graph represented as a dictionary upon execution. It then employs a priority queue. Paths that seem to lead to the destination the most effectively are given priority by the algorithm. The A\* function tracks the number of nodes created during the search and outputs the goal node it reaches, the complete path from the origin to the goal, and the total number of nodes generated in the process.

## CUS1

To use the Iterative Deepening Depth-First Search (IDDFS) algorithm, run the cus1\_search.py script in a Python 3 environment. Like the other search implementations in this assignment, this program is fully self-contained and does not rely on any external input files. The graph is defined internally with directed edges, associated costs, and a set of nodes each represented by 2D coordinates.

When executed, the program begins from a designated origin node and performs a depth-limited search that incrementally increases the allowed search depth. At each depth level, the algorithm uses a recursive function to explore paths in a depth-first manner, expanding neighbors in ascending order based on node ID. The depth limit is increased iteratively until a goal node is found, or a preset maximum depth is reached.

The algorithm terminates as soon as one of the goal nodes is reached. It returns the identified goal node, the total number of nodes created during the search process, and the complete path taken from the origin to the goal.

## CUS2

To run the CUS2 informed search algorithm, execute the cus2\_search.py script in a Python 3 environment via a terminal or command-line interface. No additional input files are required, as the graph and parameters are preconfigured in the script. The algorithm begins by generating a heuristic table using Breadth-First Search (BFS) from each goal node, mapping all other nodes to their minimum number of steps from the goal. It then performs an informed search starting from the origin, using a priority queue where nodes are prioritized based on their estimated total steps (actual steps taken + heuristic steps remaining). Unlike traditional A\*, this implementation ignores edge costs and focuses solely on minimizing the number of steps. Once a destination node is found, the script outputs the goal, steps taken, path length, and the full path from start to goal.

## Test Code

The program run\_unified\_tests.py is developed for the purpose of systematically evaluating and comparing the performance of six search algorithms—BFS, DFS, GBFS, A\*, CUS1, and CUS2—based on a fixed set of test scenarios. It serves as an automated testing framework that ensures consistency in input conditions across all algorithms and test cases.

To use the program, all algorithm scripts must be placed in the same directory as the runner script, and all test case files must be in a subdirectory named unified\_tests. Each test case file includes clearly structured definitions of nodes, edges, an origin node, and one or more goal nodes. When the program is executed, it reads and parses each test case, then dynamically injects the relevant data into each algorithm file before executing it. The algorithms are run once for each destination specified in the test file.

The output from each execution is captured and parsed into a standardized format, including the test file name, algorithm name, goal node, number of nodes involved in the search (based on the length of the returned path), the path taken, and the path cost if applicable. All results are written in a CSV file named unified\_test\_results.csv for further analysis.

This implementation does not include a graphical user interface but is structured to be modular and extensible. It ensures that all algorithms are evaluated under identical conditions, eliminating potential inconsistencies in manual testing. The dynamic injection mechanism guarantees flexibility in modifying test scenarios and supports the use of multiple goal nodes per test case, which aligns with the assignment’s expectations. This framework played a central role in generating accurate, comparable results to support the final analysis of algorithmic performance.

# Introduction

This report investigates six classical and custom-designed tree-based search algorithms—Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best-First Search (GBFS), A\* Search (AS), and two custom variants to solve the route-finding problem in directed graphs. The goal of this assignment is to implement, evaluate, and compare these algorithms in terms of pathfinding accuracy, efficiency, and adaptability across a variety of graph structures.

Each algorithm was developed in Python and tested across ten unique map-based scenarios that reflect a broad range of graph challenges, including cycles, unreachable goals, misleading heuristics, and large-scale linear expansions. These cases were carefully designed to stress different aspects of algorithmic performance such as path optimality, node expansion, and computational overhead.

A central part of this study involved designing a unified testing system that standardizes the input format, injects test cases dynamically into each algorithm, and records the outcomes in a structured CSV file. This allowed the team to perform consistent evaluations across all algorithms and identify their respective strengths and weaknesses using both qualitative insights and quantitative metrics such as node count and path cost.

Ultimately, this report aims to provide a comprehensive, comparative analysis of the algorithms’ behaviors under various conditions. The findings serve as a foundation for understanding how heuristic design, search strategy, and computational constraints influence the performance of intelligent search systems.

## DFS

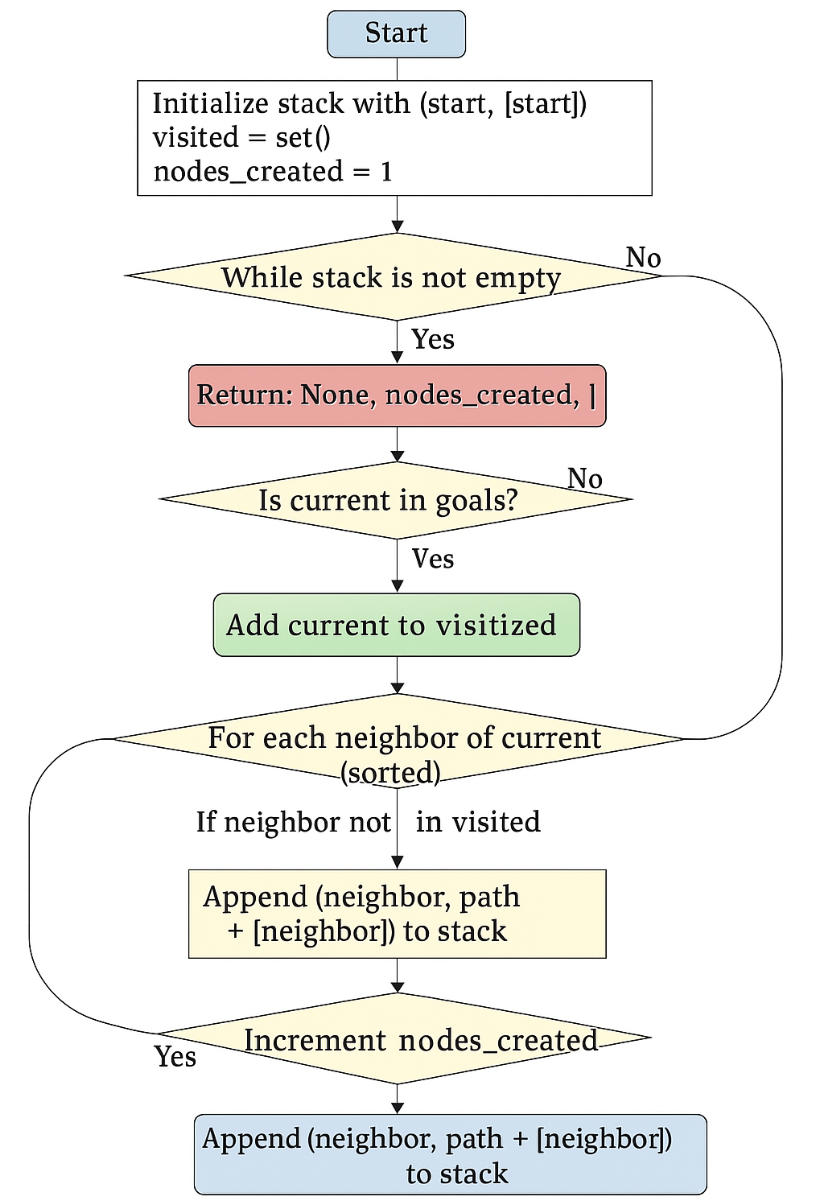


Figure1.DFS Algorithm Flowchart

## BFS

图示

AI 生成的内容可能不正确。

Figure2.BFS Algorithm Flowchart

## GBFS



Figure3.GBFS Algorithm Flowchart

## AS

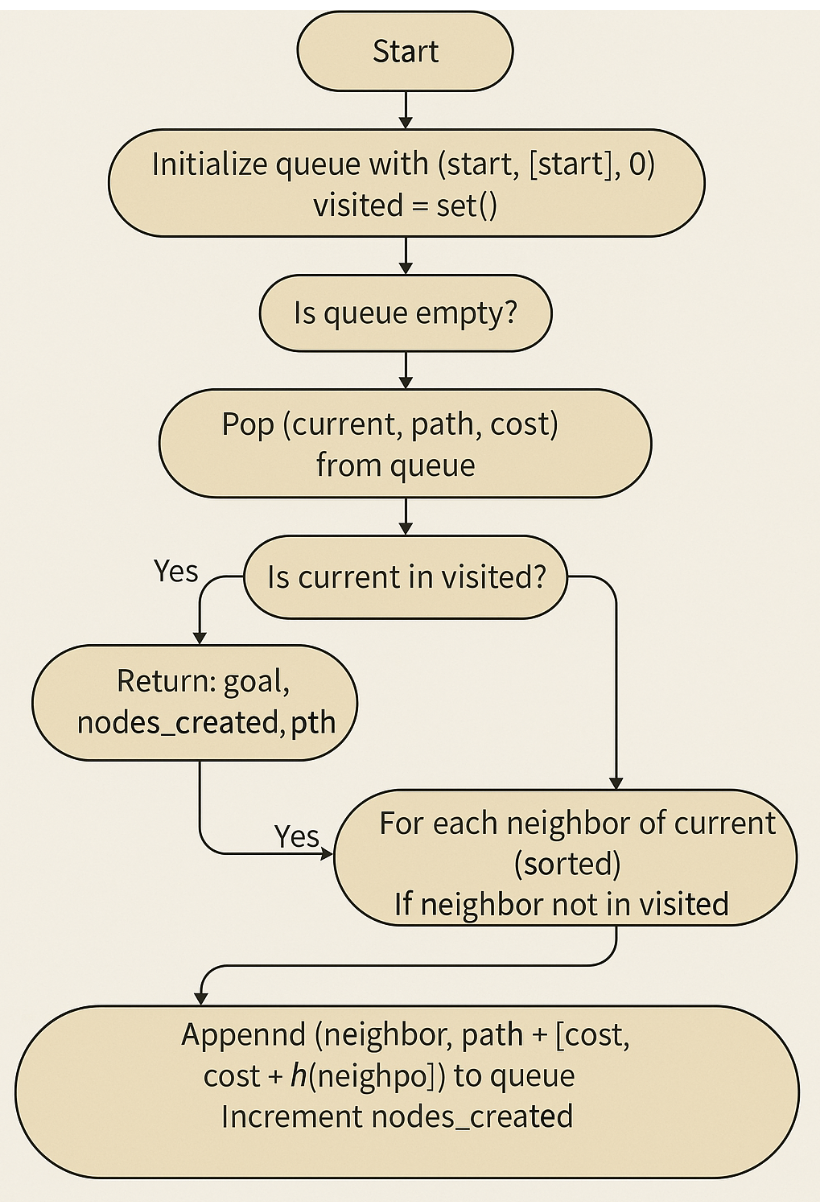


Figure4.A Search Algorithm Flowchart

## CUS1

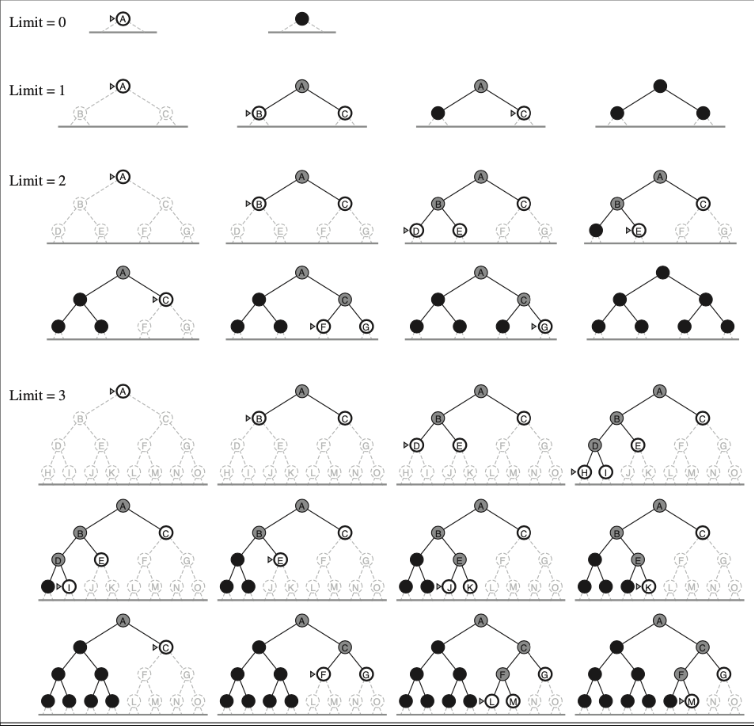


Figure5.CUS1 Algorithm Visualization（IDDFS)

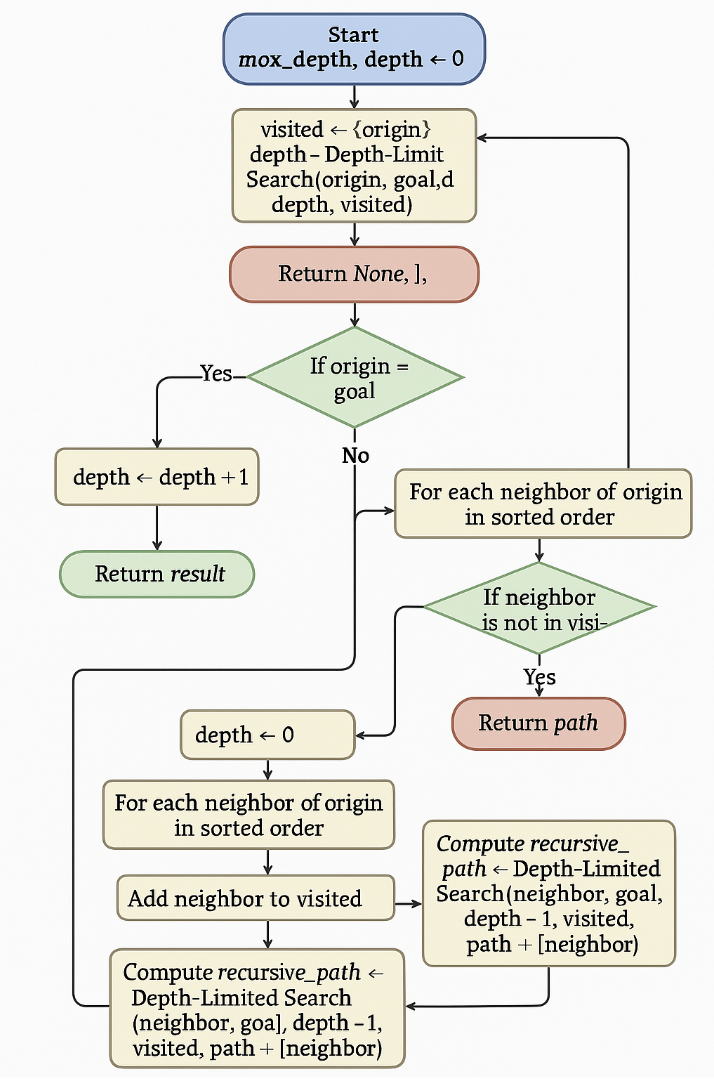


Figure6.CUS1 Algorithm Flowchart (IDDFS)

## CUS2

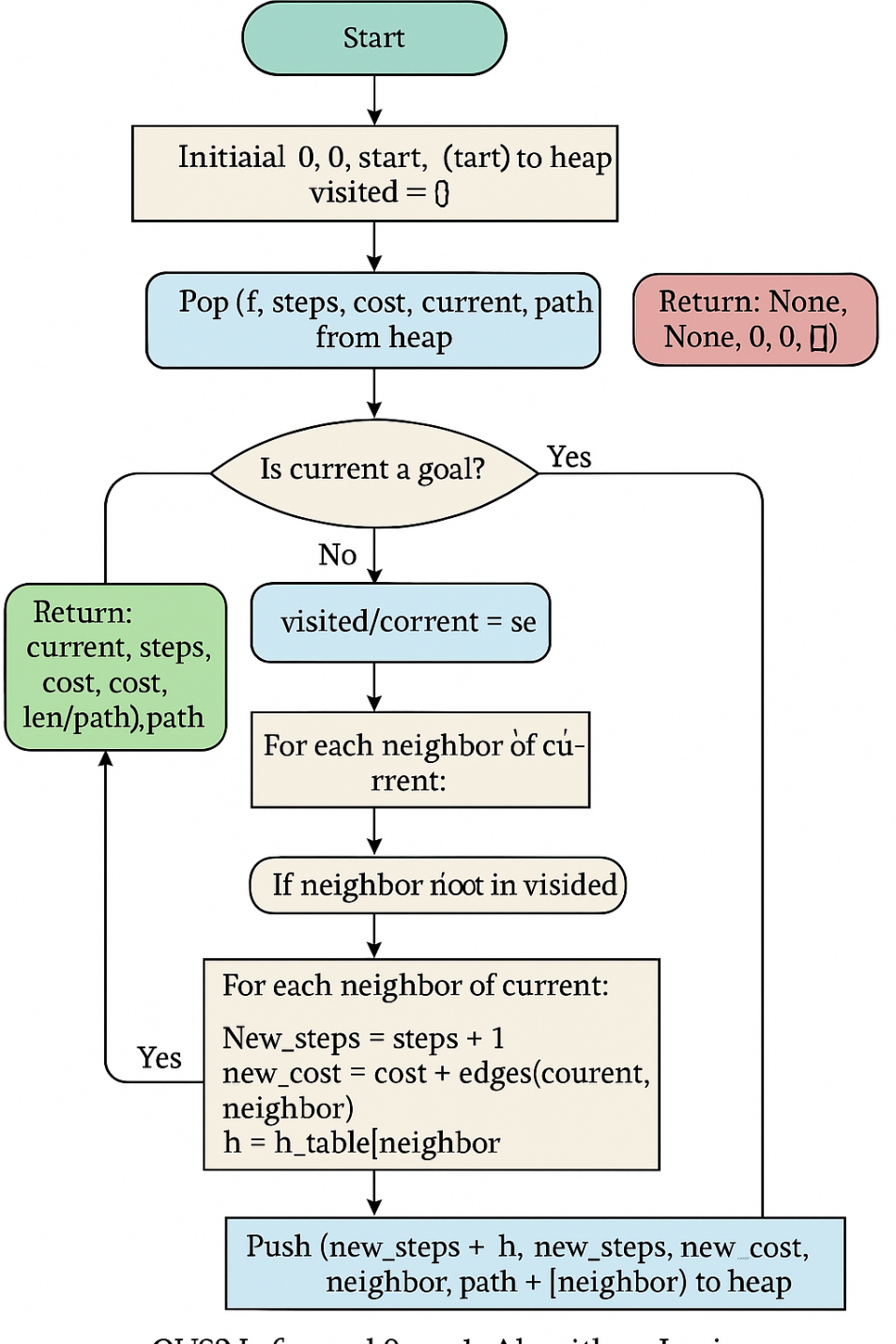


Figure7. CUS2 Algorithm Flowchart

# Features/Bugs/Missing

## DFS

Uninformed search is demonstrated in a straightforward and useful way by the Depth-First Search implementation. A directed graph is appropriately created from a predetermined dictionary of edges, and nodes are explored by going deeply along each branch before turning around using a stack (or recursive calls). In contrast to BFS, the method effectively traverses huge search spaces with comparatively low memory overhead, however it does not guarantee the shortest path. By keeping a visited set and making sure nodes aren't revisited, it prevents cycles. The number of nodes generated during the search process is successfully tracked by the programme and returned, which is useful for comparing algorithm performance. Additionally, it ensures consistency and predictability in traversal by adhering to the condition that nodes be expanded in ascending order when there are several possibilities.

Currently, the implementation does not consider edge costs in its decision-making, which aligns with DFS’s strategy but limits its applicability in scenarios where cost optimization is essential. Additionally, it lacks support for reading problem instances from external files, which is part of the assignment specifications. The algorithm is designed to terminate upon reaching the first goal node, which may not be the most optimal one in the presence of multiple goals. There are no known bugs in this implementation.

## BFS

The Breadth-First Search implementation provides a clear and functional demonstration of uninformed search. It correctly constructs a directed graph from a predefined dictionary of edges and uses a queue to explore nodes in a level-order manner. The algorithm guarantees the shortest path in terms of the number of moves (if edge costs are equal or irrelevant) and avoids cycles by maintaining a visited set. The program successfully tracks and returns the number of nodes created during the search process, which is valuable for performance comparison across different algorithms. It also follows the requirement of expanding nodes in ascending order when all else is equal, ensuring predictable and consistent behavior.

Currently, the implementation does not account for edge costs during decision making, which aligns with BFS’s strategy but limits its usefulness in weighted graphs where cost differences matter. Additionally, there is no support for reading problem instances from external files as specified in the assignment format. Moreover, the algorithm is designed to stop upon reaching the **first** goal node, without checking whether it is the most cost-efficient among multiple destinations. There are no known bugs in this implementation.

## GBFS

The Greedy Best-First Search implementation is an informed search strategy that incorporates a heuristic function—specifically, the Euclidean distance between a node and the goal—to guide its search. This implementation correctly utilizes a priority queue to always expand the node that appears closest to any of the specified goal nodes. The script handles multiple goal nodes by choosing the minimum heuristic estimate for each neighbor during expansion. It also tracks the number of created nodes and returns the corresponding path, making it suitable for comparison with other search strategies. The use of coordinates and geometric distance provides a practical heuristic for spatial search problems, such as route finding.

However, as with the BFS version, this implementation does not support reading from external input files, which limits its usability for batch testing or generalization to unseen problem instances. Additionally, although the algorithm uses heuristic information, it does not consider the path cost from the start node, meaning it cannot guarantee optimal solutions. This is a known characteristic of GBFS and not a bug, but it is important to recognize this limitation. No runtime bugs were encountered during testing, and the algorithm behaves as expected on the provided graph.

## AS

The A\* Search implementation demonstrates an effective example of informed search by combining path cost and heuristic estimates to guide exploration. It constructs a directed graph from a predefined dictionary of edges and uses a priority queue to explore nodes based on the sum of the cost-so-far and a heuristic estimate to the goal (f(n) = g(n) + h(n)). The algorithm guarantees the optimal path, provided the heuristic is admissible and consistent. It avoids revisiting nodes with higher costs by maintaining a record of visited nodes and their best-known path costs. The program accurately tracks and reports the number of nodes created during the search, which is valuable for evaluating efficiency across different search strategies. It also expands nodes in ascending order of f-cost, ensuring consistent and predictable behavior when ties are broken deterministically.

As required by the assignment format, the implementation currently assumes a predefined heuristic and does not accept dynamic heuristic input or reading issue cases from external files. Unless specifically configured to do so, it is not intended to compare several goal nodes for the most economical choice, even though it performs well in single-goal problems. Although the method works well in weighted graphs, it might need further logic to accommodate adjustable cost schemes or other heuristic functions. This implementation is known to be bug-free.

## CUS1

The CUS1 implementation in this assignment adopts the **Iterative Deepening Depth-First Search (IDDFS)** algorithm, with all required functionalities fully implemented. The map data is defined directly within the function load\_map\_from\_memory(), including node coordinates, directed edges, and associated edge costs. This hardcoded setup ensures the program can be executed and tested independently of any external input files.

The algorithm supports multiple goal nodes. During the search process, it returns as soon as **any** goal is found, without needing to evaluate all possibilities. The core of the algorithm is a recursive **Depth-Limited Search (DLS)** function, which is repeatedly invoked with an incrementally increasing depth limit, starting from depth 0 and proceeding up to a pre-defined maximum (default: 100), until a valid path to a goal node is discovered.

To comply with the assignment’s node expansion rules, particularly the requirement to expand lower-numbered nodes first, the program uses the sorted() function to sort all neighbor nodes prior to expansion. The path is recorded during the search, and once a goal is found, the complete traversal from the origin to the goal is output in order.

In addition, a node creation counter (nodes\_created[0]) is implemented to track the number of nodes generated during the search process. This supports performance evaluation and allows for meaningful comparison with other algorithms implemented in the assignment.

Overall, the CUS1 implementation is structurally sound, functionally complete, and stable in execution. The command-line output strictly follows the assignment’s required format, displaying the map name, search method, goal node ID, total nodes created, and the full path found. The program successfully performs pathfinding from the origin to one of the defined goal nodes using a purely uninformed search strategy.

## CUS2

The cus2\_search.py program implements an informed search algorithm based on the A\* (A-star) strategy. One of the key features is the use of the evaluation function *f(n)=g(n)+h(n)f(n) = g(n) + h(n)*f(n)=g(n)+h(n), where *g(n)g(n)*g(n) represents the number of steps taken from the origin and *h(n)h(n)*h(n) is the heuristic estimate of the remaining distance to the goal. To ensure that the heuristic is both admissible and consistent, a reverse breadth-first search (BFS) is performed from each goal node to precompute the shortest path distances to all other nodes. This design allows the algorithm to maintain optimality while significantly improving search efficiency.

A priority queue implemented using Python’s built-in heapq module is employed to always expand the node with the lowest estimated total cost. The graph is internally represented as an adjacency list, converted from a predefined set of bidirectional edges, and the search process avoids revisiting nodes through the use of a visited dictionary. This prevents redundant computation and ensures correctness in path reconstruction.

The algorithm also supports multiple destination nodes. For each destination, it independently computes and outputs the shortest path from the origin, along with the number of steps and the complete path taken. The output format is clearly structured to match the assignment specification, displaying the filename, method name, target node, path length, step count, and the path itself.

## Test code

The run\_unified\_tests.py script is a unified testing framework designed to automate the execution of six search algorithms across ten standardised test files. It reads each test case, injects the graph structure and search parameters into every algorithm file, executes the modified code, and collects the results in a structured CSV format for further analysis. The system supports multiple destination nodes per test case and parses key outputs such as goal reached, path length, full path, and cost (where applicable).

A key feature of this script is its dynamic injection mechanism, which allows the same test case to be applied uniformly to all algorithm scripts without manual editing. This ensures consistency and fairness in algorithm evaluation. Additionally, the program handles output parsing robustly, distinguishing between algorithms that return path cost and those that do not.

However, a current limitation exists: the script fails to properly generate and capture results for **GBFS and A\*** when processing test10.txt. This issue is likely related to either the size or structure of the test graph, or possibly due to a timeout or memory condition not being handled in the respective algorithm files. This bug does not affect the execution of the other four algorithms.

In future iterations, the system could be improved by incorporating exception logging to clearly flag which algorithm-test combinations fail and why, as well as adding timeouts or memory limits to safely handle large or complex graphs.

# Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test | Algorithm | Goal | Nodes Created | Path | Cost |
| test01 | BFS | 5 | 3 | [2, 3, 5] | N/A |
| test02 | BFS | 6 | 4 | [1, 2, 5, 6] | N/A |
| test03 | BFS | 3 | 3 | [1, 2, 3] | N/A |
| test04 | BFS | 5 | 4 | [1, 3, 4, 5] | N/A |
| test05 | BFS | 6 | 6 | [1, 2, 3, 4, 5, 6] | N/A |
| test06 | BFS | 5 | 3 | [1, 2, 5] | N/A |
| test07 | BFS | 3 | 3 | [1, 2, 3] | N/A |
| test08 | BFS | 4 | 0 | [] | N/A |
| test09 | BFS | 4 | 4 | [1, 2, 3, 4] | N/A |
| test10 | BFS | 30 | 30 | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] | N/A |
| test01 | DFS | 5 | 4 | [2, 1, 3, 5] | N/A |
| test02 | DFS | 6 | 4 | [1, 2, 5, 6] | N/A |
| test03 | DFS | 3 | 3 | [1, 2, 3] | N/A |
| test04 | DFS | 5 | 4 | [1, 3, 4, 5] | N/A |
| test05 | DFS | 6 | 6 | [1, 2, 3, 4, 5, 6] | N/A |
| test06 | DFS | 5 | 3 | [1, 2, 5] | N/A |
| test07 | DFS | 3 | 3 | [1, 2, 3] | N/A |
| test08 | DFS | 4 | 0 | [] | N/A |
| test09 | DFS | 4 | 4 | [1, 2, 3, 4] | N/A |
| test10 | DFS | 30 | 30 | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] | N/A |
| test01 | GBFS | 5 | 3 | [2, 3, 5] | 10 |
| test02 | GBFS | 6 | 4 | [1, 4, 5, 6] | 4 |
| test03 | GBFS | 3 | 4 | [1, 4, 5, 3] | 4 |
| test04 | GBFS | 5 | 4 | [1, 3, 4, 5] | 3 |
| test05 | GBFS | 6 | 6 | [1, 2, 3, 4, 5, 6] | 5 |
| test06 | GBFS | 5 | 3 | [1, 2, 5] | 2 |
| test07 | GBFS | 3 | 4 | [1, 5, 4, 3] | 3 |
| test08 | GBFS | 4 | 0 | [] | 0 |
| test09 | GBFS | 4 | 4 | [1, 2, 3, 4] | 5 |
| test10 | GBFS | 30 | 30 | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] | 29 |
| test01 | ASTAR | 5 | 3 | [2, 3, 5] | 10 |
| test02 | ASTAR | 6 | 4 | [1, 3, 5, 6] | 4 |
| test03 | ASTAR | 3 | 4 | [1, 4, 5, 3] | 4 |
| test04 | ASTAR | 5 | 4 | [1, 3, 4, 5] | 3 |
| test05 | ASTAR | 6 | 6 | [1, 2, 3, 4, 5, 6] | 5 |
| test06 | ASTAR | 5 | 3 | [1, 2, 5] | 2 |
| test07 | ASTAR | 3 | 3 | [1, 2, 3] | 2 |
| test08 | ASTAR | 4 | 0 | [] | 0 |
| test09 | ASTAR | 4 | 4 | [1, 2, 3, 4] | 5 |
| test10 | ASTAR | 30 | 30 | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] | 29 |
| test01 | CUS1 | 5 | 3 | [2, 3, 5] | N/A |
| test02 | CUS1 | 6 | 4 | [1, 2, 5, 6] | N/A |
| test03 | CUS1 | 3 | 3 | [1, 2, 3] | N/A |
| test04 | CUS1 | 5 | 4 | [1, 3, 4, 5] | N/A |
| test05 | CUS1 | 6 | 6 | [1, 2, 3, 4, 5, 6] | N/A |
| test06 | CUS1 | 5 | 3 | [1, 2, 5] | N/A |
| test07 | CUS1 | 3 | 3 | [1, 2, 3] | N/A |
| test08 | CUS1 | 4 | 0 | [] | N/A |
| test09 | CUS1 | 4 | 4 | [1, 2, 3, 4] | N/A |
| test10 | CUS1 | 30 | 30 | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] | N/A |
| test01 | CUS2 | 5 | 3 | [2, 3, 5] | 10 |
| test02 | CUS2 | 6 | 4 | [1, 2, 5, 6] | 4 |
| test03 | CUS2 | 3 | 3 | [1, 2, 3] | 11 |
| test04 | CUS2 | 5 | 4 | [1, 3, 4, 5] | 3 |
| test05 | CUS2 | 6 | 6 | [1, 2, 3, 4, 5, 6] | 5 |
| test06 | CUS2 | 5 | 3 | [1, 2, 5] | 2 |
| test07 | CUS2 | 3 | 3 | [1, 2, 3] | 2 |
| test08 | CUS2 | 4 | 0 | [] |  |
| test09 | CUS2 | 4 | 4 | [1, 2, 3, 4] | 5 |
| test10 | CUS2 | 30 | 30 | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30] | 29 |

Table1. Testing Log

Test01: Instructor's sample map – Serves as a baseline example to validate algorithm correctness.

Test02: Multiple branches, goal in the middle – Highlights BFS and DFS differences in how early they discover the target.

Test03: Multiple paths with different costs – Tests A\* vs GBFS behavior; A\* should perform better by considering cost.

Test04: Heuristic trap – Designed to mislead GBFS with poor heuristics, exposing its vulnerability.

Test05: Deep goal requiring backtracking – Shows DFS’s inefficiency when the goal lies deep and far from the initial path.

Test06: Graph with cycles – Tests the algorithm's ability to avoid revisiting nodes; ensures correct use of visited set.

Test07: All edge weights equal – Heuristic-based algorithms like GBFS and A\* should not benefit from heuristics in this case.

Test08: Unreachable goal – Verifies whether the algorithm handles unreachable goals gracefully without crashing.

Test09: Multiple equally short paths – Ensures algorithms can still choose the one with the lowest total cost if applicable.

Test10: Large linear graph with 30+ nodes – Measures performance and scalability; tests node expansion efficiency under load.

# Insights： **Comparative Analysis of Search Algorithms**

This section presents a critical analysis of the six search algorithms implemented in the assignment—Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Best-First Search (GBFS), A\* Search, CUS1, and CUS2—using a series of ten diverse test cases. The evaluation draws on quantitative performance indicators such as path length, total cost, and node expansion, as well as qualitative insights based on algorithmic design and behavior under different graph conditions.

### **Informed vs Uninformed Search: A Foundational Distinction**

The algorithms tested can be broadly categorized into two types: uninformed search (BFS, DFS, CUS1) and informed search (GBFS, A\*, CUS2). This distinction fundamentally shapes their behavior and performance.

Uninformed search algorithms do not use any problem-specific knowledge or estimates of distance to the goal. They explore the search space systematically or exhaustively. BFS, DFS, and CUS1 belong to this category.

Informed search algorithms make use of heuristics—estimates of the cost from a node to the goal—to prioritize node expansion. GBFS, A\*, and CUS2 fall into this category.

The use of heuristics allows informed search methods to significantly reduce search time and improve efficiency, especially in large graphs or complex problem spaces. However, their performance heavily depends on the quality of the heuristic used.

### **1. Breadth-First Search (BFS)**

As an uninformed method, BFS performs a level-by-level exploration. It guarantees the shortest path in terms of steps if all edge costs are equal. In test cases such as Test03 and Test07, it performs predictably. However, BFS’s lack of heuristic guidance means that it may expand many unnecessary nodes—an inefficiency clearly visible in Test05 and Test10. Furthermore, it fails to account for edge weights, rendering it **ineffective in cost-sensitive scenarios**.

### **2. Depth-First Search (DFS)**

DFS is also uninformed and explores deeply along paths before backtracking. Its performance is inconsistent: it may be quick in shallow goal cases but performs poorly in deeper or cyclic graphs. It does not guarantee optimality, neither in steps nor cost. In Test10, DFS traverses the entire graph linearly, leading to excessive expansions and suboptimal paths.

### **3. Greedy Best-First Search (GBFS)**

GBFS introduces heuristics into the search process, making it one of the informed algorithms. It selects nodes based solely on estimated proximity to the goal (h(n)), ignoring path cost. This makes it fast but unreliable. In Test04, GBFS is misled by an inaccurate heuristic (a classic heuristic trap), resulting in suboptimal performance. Despite this, it achieves low cost and fast resolution in many scenarios, such as Test06 and Test10.

### **4. A\* Search**

A\* is the most principled informed method tested. It evaluates each node using the formula f(n) = g(n) + h(n), balancing actual path cost and heuristic estimation. This allows it to consistently find optimal solutions, both in terms of cost and steps. In Test03 and Test06, it outperforms all others in cost-efficiency. Unlike GBFS, A\* can avoid heuristic pitfalls by factoring in real cost, making it more robust and generalizable.

### **5. CUS1 (Uninformed, IDDFS-inspired)**

CUS1 was designed to simulate breadth-first behavior while conserving memory, using an iterative deepening approach. As an uninformed method, it does not use any heuristics and therefore lacks guidance in cost-sensitive graphs. It performs comparably to BFS in step count but does not show practical advantages over BFS or A\* in weighted scenarios. Its main value lies in theoretical memory efficiency, which was not a limiting factor in our tests.

### **6. CUS2 (Informed, Custom A\* Variant)**

CUS2 is a custom-informed search that integrates a heuristic table derived via **reverse** BFS from the goal. It behaves similarly to A\*, balancing real path cost and estimated goal distance. In nearly all test cases, it matches or slightly trails A\* in both step count and cost, showing strong practical effectiveness. The main discrepancy arises in Test03, where a simplified heuristic fails to capture path variability. Still, CUS2 demonstrates that even a basic heuristic can lead to significant performance gains over uninformed methods.

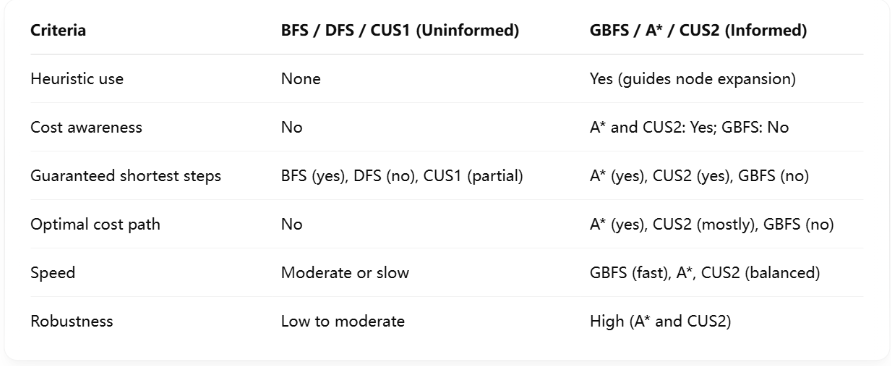


Table2.Comparative Summary

# Research (if applicable)

* **1. Depth-First Search (DFS)**

Depth-First Search (DFS) explores a search tree or graph by expanding the deepest unexplored node first. It uses a stack-based structure (explicitly or via recursion), making its memory usage minimal—just proportional to the depth of the tree. However, this also makes it susceptible to getting stuck in deep or infinite branches, especially in graphs with cycles or high branching factors. DFS is not complete (may not find a solution) unless there's a mechanism to avoid revisiting nodes, and it's not optimal since it doesn’t consider path cost. It performs well in scenarios with low solution density and bounded depth. DFS has been widely used in simple puzzle solving, syntax parsing, and as a base for more advanced algorithms [1][2].

### **2. Breadth-First Search (BFS)**

Breadth-First Search (BFS) expands the shallowest nodes first using a queue structure. Unlike DFS, it guarantees finding the shortest path in terms of the number of steps in unweighted graphs. However, BFS consumes significantly more memory as it stores all frontier nodes, which grows exponentially with tree depth. It is complete and optimal for unweighted or uniform-cost problems, making it a strong baseline for pathfinding tasks. BFS's main disadvantage is its scalability—it may be impractical for large graphs unless the branching factor is small. In educational tools and simple AI agents, BFS serves as a foundation for understanding more advanced informed strategies [3][4].

### **3. Greedy Best-First Search (GBFS)**

Greedy Best-First Search is an informed algorithm that selects the next node to expand based solely on the heuristic value *h(n)h(n)*h(n), which estimates the cost from the current node to the goal. This makes GBFS faster than uniform search methods like BFS, especially in large spaces with well-designed heuristics. However, its major downside is that it’s not optimal—poor heuristics can misguide the search, leading to dead ends or suboptimal paths. GBFS works well in real-time applications where speed is preferred over optimality. Studies show that while GBFS may outperform A\* in raw speed, it typically yields lower-quality solutions [5][6].

### **4. A\* Search**

A\* is a powerful informed search algorithm that uses both the path cost so far *g(n)g(n)*g(n) and a heuristic estimate to the goal *h(n)h(n)*h(n), combining them into *f(n)=g(n)+h(n)f(n) = g(n) + h(n)*f(n)=g(n)+h(n). It is complete and optimal when the heuristic is admissible (never overestimates the cost). A\* strikes a balance between BFS and GBFS, expanding paths that seem both promising and cheap. Its performance depends heavily on the quality of the heuristic. In many benchmark studies, A\* is the top-performing algorithm for finding shortest paths, particularly in pathfinding problems like game AI, robot navigation, and logistics planning [4][1].

### **5. Iterative Deepening Depth-First Search (IDDFS) – *CUS1***

* IDDFS combines the low memory usage of DFS with the completeness and optimality (in steps) of BFS. It performs DFS with increasing depth limits, restarting each time until the goal is found. Although it repeats work from previous iterations, the extra overhead is minor for exponential search trees. IDDFS is especially effective when the solution lies near the root or when the search depth is unknown. It’s optimal for unweighted graphs in terms of path length and requires less space than BFS. It also performs well in time-limited searches or memory-constrained environments [7][8].

### **6. Step-Count-Based Heuristic Search (CUS2)**

CUS2 is a custom informed algorithm designed to minimize the number of steps to reach the goal. It uses a heuristic generated from BFS—specifically, the shortest path length (in steps) from any node to the goal, treating all edges equally. The algorithm prioritizes nodes with the lowest total estimated steps (*f(n)=steps+h(n)f(n) = \text{steps} + h(n)*f(n)=steps+h(n)), making it a hybrid between A\* and step-based planning. This approach is especially useful in applications where move count is more important than cost (e.g., puzzle games, instruction minimization). It ensures admissibility and performs robustly across sparse and dense graphs. Similar step-focused heuristics have been used effectively in maze solvers and pathfinding games [9][10].

# Conclusion

Through the comparative analysis of six search algorithms—DFS, BFS, GBFS, A\*, IDDFS (CUS1), and a step-count-based informed method (CUS2)—this study highlights the strengths, limitations, and appropriate application domains of each technique. Uninformed strategies like DFS and BFS offer foundational understanding and simplicity but struggle with scalability or optimality. IDDFS effectively balances memory usage and completeness, making it a superior uninformed strategy for large graphs with unknown depths.

In contrast, informed methods such as GBFS and A\* demonstrate significantly improved performance in goal-directed tasks when heuristics are reliable. A\*, in particular, consistently provides optimal paths at reasonable computational costs. The custom-designed CUS2 strategy, which leverages step-based heuristics derived from BFS, effectively prioritizes path length minimization, offering a specialized advantage in scenarios where minimizing moves is more critical than minimizing cost.

Overall, the results confirm that algorithm selection must be tailored to the specific problem characteristics, such as graph structure, resource constraints, and optimization goals. The hybrid and heuristic-driven strategies (A\*, CUS2) are most effective for general-purpose pathfinding, while IDDFS presents a strong alternative in constrained or unpredictable environments.

# Acknowledgements/Resources

**Python Official Documentation**

https://docs.python.org/3/

The official Python documentation was frequently referenced to ensure the correct use of built-in modules such as collections, heapq, math, and csv. It supported accurate graph traversal logic and file handling throughout the project.

**ChatGPT (OpenAI)**

ChatGPT was used to support code structuring, debugging, output standardization, and batch testing automation. It assisted in unifying the output format across algorithms and in dynamically injecting test cases during the testing phase.

**Stack Overflow**

https://stackoverflow.com/

Consulted for resolving programming issues such as UnicodeDecodeError and for safe file operations using UTF-8 encoding. This ensured compatibility when reading .txt map files and writing .csv results across platforms.

**CSV Formatting References (W3Schools & Python Docs)**

Guidelines from W3Schools and the Python CSV documentation helped ensure that the unified\_test\_results.csv file was properly formatted, readable, and compatible with spreadsheet tools such as Excel.

**Consensus**

https://consensus.app/

Used for literature exploration and quick understanding of academic insights related to graph search algorithms. It helped verify technical decisions with reference to peer-reviewed findings.

**Google Translate**

https://translate.google.com/

Primarily used for quick grammar verification and translation of technical terms between English and other languages to ensure correct understanding and usage in documentation.

**YouTube**

https://www.youtube.com/

Educational videos were consulted to better understand the practical implementations of graph search algorithms and their time/space complexities,

# References

[1] Aliyan, M., Hasan, M., Qayoom, H., Hussain, M., Nosheen, S., Mustafa, M., Chuhan, S., Yaqub, M., Awan, R., & Bilal, A. (2024). Analysis and Performance Evaluation of Various Shortest Path Algorithms. *2024 3rd International Conference for Innovation in Technology (INOCON)*, 1-16. <https://doi.org/10.1109/INOCON60754.2024.10512150>.

[2] Raja, S., Rajkumar, T., & Sowndariya, K. (2017). Artificial-intelligence-based heuristic searching tools and knowledge representation to solve cryptography problems, puzzles, vehicle detection and path finding. *International Journal of Signal and Imaging Systems Engineering*, 10, 1. <https://doi.org/10.1504/IJSISE.2017.10005424>.

[3] Jun, Z. (2007). Performance comparison of search algorithms based on eight puzzle problem. *Computer Engineering and Design*.

[4] Salem, N., Haneya, H., Balbaid, H., & Asrar, M. (2024). Exploring the Maze: A Comparative Study of Path Finding Algorithms for PAC-Man Game. *2024 21st Learning and Technology Conference (L&T)*, 92-97. <https://doi.org/10.1109/LT60077.2024.10469459>.

[5] Wilt, C., Thayer, J., & Ruml, W. (2010). A Comparison of Greedy Search Algorithms. , 129-136. <https://doi.org/10.1609/socs.v1i1.18182>.

[6] Xie, F. (2013). Exploration and Combination: Randomized and Multi-Strategies Search in Satisﬁcing Planning. .

[7] Vempaty, N., Kumar, V., & Korf, R. (1991). Depth-first vs best-first search. , 434-440.

[8] Noori, A., & Moradi, F. (2015). Simulation and Comparison of Efficency in Pathfinding algorithms in Games. , 37, 230-238. <https://doi.org/10.5902/2179460X20778>.

[9] Narendran, A., Hothri, A., Saga, H., & Sahay, A. (2024). MazeSolver: Exploring Algorithmic Solutions for Maze Navigation. *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 1-8. <https://doi.org/10.1109/ICCCNT61001.2024.10725996>.

[10] Shashwat, K. (2022). A comparative study of the performance of different search algorithms on FOON graphs. *ArXiv*, abs/2210.07428. <https://doi.org/10.48550/arXiv.2210.07428>.