Assignment 2A - Group Report

Dwayne D’Souza - 105332012

Mitchell Walker - 103995277

Al Shahriar Miad - 104473978

Ruzmy Mohamed - 104799360

# Contents

[**Contents 1**](#_73w7kq1kc2lq)

[**1.Instructions 2**](#_nefphppp7qs0)

[**2. Introduction 2**](#_8k0auuc05e76)

[The Route Finding Problem 2](#_pt2urnxeoc9z)

[Graph and Tree Concepts 3](#_trsg7h92ausw)

[Search Algorithms Overview 3](#_ntnbett8r4ql)

[**3. Features/Bugs/Missing 4**](#_v7lbdng7z7ir)

[Implemented Features 4](#_c257t3u9tmmc)

[Bugs 4](#_j82gak1wi8xo)

[Notes on Implementation 4](#_zgsmf440e5up)

[**4. Testing: 4**](#_qdu4zyilyd34)

[**5. Insights 6**](#_ye2j3b6mnzml)

[**6. Research 7**](#_hsc7y16ro9mw)

[**7. Conclusion 8**](#_mq6wz97j0gfu)

[**8. Acknowledgements/Resources 9**](#_aphlutmu7zpu)

[**9. References 9**](#_1mt78f4vtio2)

# 

# 1.Instructions

To run our program, use the following command in a Windows 10 Command Prompt:

C:\Assignments>pip install networkx matplotlib

C:\Assignments> python search.py <filename> <method>

* <filename>: Path to the text file containing the route-finding problem
* <method>: One of dfs, bfs, gbfs, astar, ucs, iddfs.

Example:

C:\Assignments> python search.py PathFinder-test.txt gbfs

Output Format:

Loading graph from PathFinder-test.txt...

Graph loaded.

Selected search method: gbfs

Search algorithm created.

Starting search...

Search finished.

PathFinder-test.txt gbfs

4 5

2 1 4

Graph visualization saved as PathFinder-test\_gbfs\_path.png

* The program reads the problem file, applies the specified search method, and outputs the goal reached, number of nodes expanded, and the path taken as well as the name of the visualisation of the graph with the solution as a png.

# 2. Introduction

### **The Route Finding Problem**

The Route Finding Problem is a classic pathfinding challenge where an agent must find the optimal path from an origin node (O) to one or more destination nodes (D) on a directed weighted graph. The problem is fundamental to AI and has applications in navigation systems, logistics, network routing, and game AI.

In our implementation, the problem is represented as a 2D graph where:

* Nodes have specific coordinates (x,y)
* Edges connect nodes with associated costs
* Some edges are directional (can only be traversed in one direction)
* The agent starts at a designated origin node
* The goal is to reach any one of potentially multiple destination nodes

### **Graph and Tree Concepts**

**Graph**: A graph G = (V, E) consists of a set of vertices V and a set of edges E. In our implementation, vertices represent locations, and edges represent possible paths between locations with associated costs.

**Tree**: A tree is a special type of graph with no cycles. In search algorithms, we construct a search tree where:

* The root node is the starting state (origin)
* Child nodes represent states reachable from their parent
* Branches represent actions or moves
* Leaf nodes are terminal states or unexpanded nodes

**Search Space**: The set of all possible states accessible from the initial state by applying a sequence of actions. In our route finding problem, this encompasses all paths that could potentially be taken from the origin.

### **Search Algorithms Overview**

We have implemented six search algorithms:

1. **Depth-First Search (DFS)**: An uninformed algorithm that explores as far as possible along each branch before backtracking.
2. **Breadth-First Search (BFS)**: An uninformed algorithm that explores all neighbors at the present depth before moving to nodes at the next depth level.
3. **Greedy Best-First Search (GBFS)**: An informed algorithm that selects the path that appears to be closest to the goal according to a heuristic function.
4. **A\* (AS):** An informed algorithm that combines the cost to reach the node and the estimated cost to the goal.
5. **Iterative Deepening Depth-First Search (IDDFS)** [CUS1]: An uninformed algorithm that performs DFS with an increasing depth limit until a solution is found.
6. **Uniform Cost Search (UCS)** [CUS2]: An informed algorithm that expands nodes based on their path cost from the start node.

# 3. Features/Bugs/Missing

### **Implemented Features**

* Graph Parser: Successfully parses graph files in the specified format
* Abstract Search Interface: A common interface for all search algorithms
* Uninformed Search Algorithms:
* Depth-First Search (DFS)
* Breadth-First Search (BFS)
* Iterative Deepening Depth-First Search (IDDFS) as our first custom algorithm
* Informed Search Algorithms:
* Greedy Best-First Search (GBFS)
* A\* Search (AS)
* Uniform Cost Search (UCS) as our second custom algorithm
* Converts the search results into a graph visualization that shows the whole network and the solution found
* Output Format: Results displayed according to the assignment specifications
* Command-Line Interface: Command-line arguments handled correctly
* Error Handling: Robust error checking and graceful failure

### **Bugs**

* No known bugs in the current implementation. All algorithms successfully find paths when they exist.

### **Notes on Implementation**

* For A\* and GBFS, we use Euclidean distance as our heuristic function, which is admissible for this 2D space.
* When all else is equal, nodes are expanded in ascending order of their IDs as required.
* The program outputs the first destination path found when multiple destinations are specified.

# 4. Testing:

* evaluatepathfinding.txt
* Has the functionality to ensure the algorithms following evaluate the pathing to solve for a routing problem. We did this to test whether the BFS and DFS algorithms can handle basic pathing successfully
* Test Case result for **DFS** model:

19 25

1 2 3 4 10 9 5 6 7 16 15 14 13 12 11 22 21 20 8 18 19

* Test Case result for **BFS** model:

19 25

1 5 8 19

* simplegridpathing.txt
* The purpose of this test is to check that the algorithm chooses the best pathing method for multiple destinations.
* Test Case result for **DFS** model:

13 16

1 2 3 6 5 4 8 11 9 12 10 13

* Test Case result for **BFS** model:

13 16

1 2 3 6 10 13

* multitargetpathfinding.txt
* The purpose for this test is to ensure that the algorithm can navigate a larger more complex map with multiple destinations.
* Test Case result for **DFS** model:

13 17

3 1 2 7 6 5 4 20 19 17 15 8 9 10 12 11 13

* Test Case result for **BFS** model:

13 20

3 2 7 9 11 13

* cycle.txt
* The purpose of the cyclic graph is to detect if there are any infinite loops and/or cycles. It should ensure that DFS and IDDFS avoid any cycle through depth limits.
* Test Case result for **DFS** model:

8 8

1 2 3 4 5 7 8

* Test Case result for **IDDFS** model:

8 30

1 2 3 4 5 7 8

* multiclusternetwork.txt
* We use this to ensure the algorithm groups and points across a big spread out map.
* Test Case result for **DFS** model:

16 22

1 2 3 5 6 7 18 17 10 8 9 11 16

* Test Case result for **BFS** model:

5 3

1 2 5

* hierarchicalgridpathfinding.txt
* The purpose of this test is to see whether the algorithm can handle a clean well organised grid as optimally as possible with multiple destinations.
* Test Case result for **DFS** model:

21 28

1 2 4 8 10 5 9 15 22 14 16 23 21

* Test Case result for **BFS** model:

16 19

1 3 5 6 8 11 16

* circularnetwork.txt
* We do this test to validate whether the algorithm can find multiple paths from the centre to points around the edge, going through a circular route to reach the destinations, outputting ‘none’ if there aren’t any valid solutions.
* Test Case result for **DFS** model:

None 1

* Test Case result for **BFS** model:

None 1

* costly.txt
* We’re doing this test case to ensure that the A\* and UCS algorithms have chosen the cheapest path, not the shortest.
* Test Case result for **A\*** model:

6 8

2 1 4 6

* Test Case result for **UCS** model:

6 8

2 1 4 6

* astar\_test.txt
* We’re checking whether A\* can properly discovery guides are efficiently searching
* Test Case result for **A\*** model:

7 8

1 2 5 7

* iddfs\_test.txt
* This test case verifies if the IDDFS is exploring the node layer by layer, with the expansion count number being reflected with the number of repeated depth-limited searches
* Test Case result for **IDDFS** model:

8 36

1 2 3 4 5 6 7 8

# 5. Insights

Although A\*, BFS and GBFS models produced the same results, in general, A\* is considered to be the best, as it’s good for finding the shortest path with the cheapest path due to its GBFS heuristic speed and its UCS general optimisation, which the other models don’t possess, or don’t share both qualities of. Allowing it to be the most effective and efficient model out of the rest, as gets the job done using the least amount of resources.

* **DFS** output:

4 5

2 1 3 5 4

* **BFS** output:

4 5

2 1 4

* **GBFS** output:

4 5

2 1 4

* **A\*** output:

4 5

2 1 4

* **UCS** output:

4 6

2 1 4

* **IDDFS** output:

4 11

2 1 4

The data shown proves that A\* has one of the best results supporting our argument that it’s the most efficient and effective model from the rest.

# 6. Research

A significant improvement was implemented to enhance the program's output: visualising the calculated path. The original program outputted the path as a sequence of node IDs and the number of expanded nodes. While functional, this textual representation can be difficult to interpret, especially for larger or more complex graphs. It doesn't provide an intuitive sense of the path's shape, its relation to the overall graph structure, or how it compares visually to potential alternative routes.

To address this, the visualize\_path function was developed and integrated into the results display process. This function leverages two powerful Python libraries commonly used for graph manipulation and plotting:

networkx: Used to model the graph structure programmatically. It takes the parsed graph dictionary and node coordinates to build an internal graph representation.

matplotlib.pyplot: Used as the plotting engine to generate the actual visual output.

How it Enhances the Program:

Intuitive Understanding: It generates a PNG image file for each successful search. This image displays the entire graph (nodes and edges) faintly in the background. The specific path found by the algorithm is then highlighted with distinct colors (e.g., salmon nodes, red edges), making it immediately stand out. The start and end nodes of the path are further emphasized with unique colors (green and purple, respectively). This visual representation allows users to instantly grasp the path's trajectory and position within the graph.

Debugging and Verification: If an algorithm produces an unexpected or seemingly inefficient path, the visualization makes it much easier to spot. Seeing the path laid out visually can help identify potential issues in the graph definition (e.g., missing edges, incorrect costs) or in the algorithm's logic itself.

Algorithm Comparison: When running different search algorithms (e.g., BFS vs. A\*) on the same graph file, the generated visualizations provide a direct, qualitative way to compare the resulting paths. One can easily see differences in path length, directness, and exploration patterns.

Contextualisation: The visualization shows the found path in the context of the entire graph, including nodes and edges not part of the solution. This helps understand why a particular path was chosen over others.

**Reporting and Communication**: The saved PNG files serve as clear, shareable artifacts for reports, presentations, or discussions about the search results.

Implementation Details:

The visualize\_path function is called by display\_results only if a path is found for a destination.

It uses the node coordinates (nodes dictionary) provided in the input file to position the nodes spatially in the plot, making the visualization correspond to the intended layout.

It gracefully handles cases where matplotlib or networkx might not be installed, printing a message instead of crashing.

Output filenames are automatically generated based on the input filename and the search method used (e.g., graph1\_bfs\_path.png), making it easy to manage multiple results.

In summary, the addition of the visualize\_path function significantly improves the program's usability and analytical power by transforming abstract path data into an easily interpretable visual format.

# 7. Conclusion

Through this assignment, we gained a thorough understanding of both informed and uninformed tree - based search algorithms in relation to a route finding problem. Understanding the functionality and its significance into artificial Intelligence. From the assignment, we studied the following six distinct algorithms being DFS, BFS, IDDFS, GBFS, A\* and UCS, in which we implemented them to see how different methods impact performance in terms of pathing as optimality as possible, search depth, and node expansion.

From our conclusion, we found that A\* Algorithm is the best as in our test it was one of the most optimal from the other algorithms, and in general was the most effective and efficient because it uses a heuristic pathing system.

We can improve upon A\* through changing formula from euclidean distance which takes pathing from a straight line, to manhattans distance which follows the pathing of the grid between the two points making it faster and working even better with grids. We can also post-processes our paths, getting rid of the searching elements of the unnecessary zig zags, making the process a lot more direct, accurate and faster. We can also store A\* heuristic pathing in cache so it doesn’t have to recalculate and rediscover the pathing every single time.

**KEY TAKEAWAYS:**

* A\* was the best method since it discovered the least expensive routes and was quick.
* Although UCS showed good ability to find ideal pathways, its inclination to investigate a larger search space usually resulted in slower performance.
* Although BFS did not guarantee the lowest cost, it did guarantee the shortest path (in number of steps).
* The IDDFS was a satisfactory compromise between the depth of DFS and the completeness of BFS
* Although GBFS was quick and easy to use, it was heavily reliant on heuristic accuracy.
* Despite its speed, DFS did not guarantee the greatest efficiency or completeness.

# 8. Acknowledgements/Resources

* **Russell and Norvig's "Artificial Intelligence: A Modern Approach"**: This textbook provided the theoretical foundation for our understanding of search algorithms and their implementation. We referenced it extensively for algorithm concepts and pseudocode.
* **GeeksForGeeks**: This website provided clear explanations and example implementations of various search algorithms that helped us understand implementation details.
* **Python Documentation**: The official Python documentation was invaluable for understanding features like priority queues (heapq), collections (deque), and other data structures used in our implementation.
* **Stack Overflow**: Several specific implementation challenges were resolved with help from the Stack Overflow community, particularly regarding priority queue behavior and cycle detection strategies.
* **AI Course Materials**: The lecture slides and tutorial exercises from COS30019 provided the fundamental knowledge required to complete this assignment.

# 9. References

1. Russell, S. J., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
3. Skiena, S. S. (2008). The Algorithm Design Manual (2nd ed.). Springer.
4. Korf, R. E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. Artificial Intelligence, 27(1), 97-109.
5. Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1(1), 269-271.
6. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.