Swinburne University of Technology

Intro to AI – COS30019

Assignment 2A

Tree Based Search

Marco Giacoppo (104071453)

Jason Tjahjono (104656753)

Xuan Huy Nguyen (103848944)

Tran Hoang Le (104486329)

Tuesday 8:30

2025 Semester 1

Table of Contents

Table of Contents.......... 2
Instruction.......... 3
Introduction.......... 5
Features, Bugs, and Missing Parts.......... 6
Testing.......... 7
Insights.......... 8
Research.......... 10
Conclusion.......... 10
Acknowledgements/Resources.......... 11
References.......... 11

Instruction

The Route Finding AI Agent solves the problem of finding an optimal path on a 2D graph from an "Origin" node to one or more "Destination" nodes. The program reads a text file containing nodes, edges, the origin, and destination(s), processes this data, and applies a user-selected tree-based search algorithm to compute the path. Supported algorithms include Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best-First Search (GBFS), A* (AS), and two custom algorithms (CUS1 and CUS2).

Setup

1. **Install Python:** Ensure Python or Python 3.x is installed on your computer. Download it from python.org if needed.
2. **Install Pyglet (Optional):** For graph visualization, install Pyglet using the command:

   ```
   pip install --upgrade --user pyglet
   ```

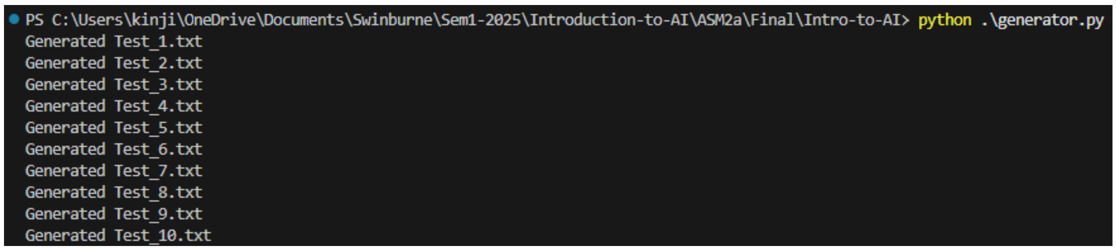
3. **Organize Files:** Place all program files in a single folder:
   - generator.py: Creates test case files.
   - search.py: Executes the search algorithms.
   - run_generator.bat: Automated testing (Windows only).

Running the Program

1. **Generate Test Cases:**
   - Open a terminal in the folder containing the files.
   - Run:

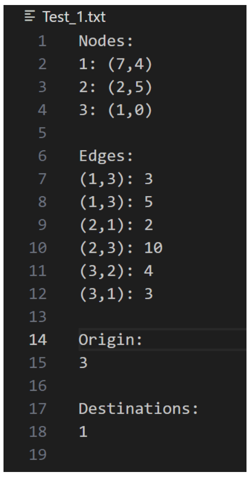
     ```
     python generator.py
     ```

   - This generates text files with test cases (nodes, edges, origin, destinations)
   - To customize test cases (e.g., number of nodes, edge density), edit the generate_test_case() function in generator.py to adjust parameters like graph size or connectivity.



```
PS C:\Users\kinji\OneDrive\Documents\Swinburne\Sem1-2025\Introduction-to-AI\ASM2a\Final\Intro-to-AI> python .\generator.py
Generated Test_1.txt
Generated Test_2.txt
Generated Test_3.txt
Generated Test_4.txt
Generated Test_5.txt
Generated Test_6.txt
Generated Test_7.txt
Generated Test_8.txt
Generated Test_9.txt
Generated Test_10.txt
```

Figure 1: Example output of generator.py showing a generated test file.



```
Nodes:
1: (7,4)
2: (2,5)
3: (1,0)

Edges:
(1,3): 3
(1,3): 5
(2,1): 2
(2,3): 10
(3,2): 4
(3,1): 3

Origin:
3

Destinations:
1

```

Figure 2: Example of the test_files layout

2. **To run the program:**

   python search.py <filename> <method> [--visualize]

- `<filename>`: path to the test case file.
- `<method>`: one of DFS, BFS, GBFS, AS, CUS1, or CUS2.
- `--visualize`: (optional) launches the GUI using pyglet.

3. **Batch testing (Windows Only):**
   - Double-click run_generator.bat or run it from the command prompt.
   - This script tests all search methods on all generated test files, outputting results (goal, node count, path, time taken, and cost where applicable) for comparison.
   - Requires Windows 10 or later.

Input file format

The text file must follow this format:

- Nodes: followed by lines like 1: (4,1) (node ID and x,y coordinates).
- Edges: followed by lines like (2,1): 4 (edge from node 2 to 1 with cost 4).
- Origin: followed by the starting node ID (e.g., 2).
- Destinations: followed by goal node IDs separated by semicolons (e.g., 5;4).

Note

- Ensure the Python compiler is added to your system's PATH for terminal commands to work.
- The visualization feature (via Pyglet) draws the graph and highlights the computed path, aiding debugging and analysis.
- The program supports command-line operation as required, tested on Windows 10.
- For issues with batch file execution, verify all files are in the same folder and Python is correctly installed.

Introduction

The Route Finding Problem involves finding optimal paths from a starting point to one or more goal destinations on a graph. Each node represents a location, and each edge represents a traversable path with a specific cost (can either be bi-directional or one-directional). Tree-based search algorithms explore these graphs to find valid paths, either blindly or using heuristics.

In this assignment, we implemented the following algorithms:

- **Uninformed methods**:
  - **Depth-First Search (DFS)**

    Depth First Search (DFS) is also a graph traversal algorithm, which expands nodes vertically; Prioritises going as deep as it possibly can, before backtracking to nodes in previous layers. Hence, this method utilises a stack, which operates on the Last in First Out (LIFO) principle. Because of its behaviour, it may find solutions without expanding all nodes, saving time and memory. This algorithm is especially good for finding solutions that are deep. However, this does not guarantee the shortest path is taken, as it returns the first solution found, without checking whether another branch has a shorter path [6].

  - **Breadth-First Search (BFS)**

    Breadth First Search (DFS) is a graph traversal algorithm, which expands nodes in the same depth before continuing to the next layer. Hence, this method utilises a queue, which operates on the First In First Out (FIFO) principle. It will do so until it finds the goal node. Advantages of this algorithm is that it will guarantee that the result is the shortest path (in an unweighted graph), as it checks layer by layer until a solution is found. Disadvantage of this algorithm is the time complexity, and memory usage. This algorithm checks all nodes in all layers until a solution is found, hence taking time and memory allocation, especially in a large graph with the solution being deep [6].

- **Informed methods**:
  - **Greedy Best-First Search (GBFS)**

    Greedy Best First Search (GBFS) is a search algorithm that chooses its path based on a heuristic [7]. A heuristic is defined as a method or rule that will return a value used to determine a path with the lowest cost. In the case of the problem above, we implemented a heuristic that calculates the euclidean distance of a node to the destination, which lets the algorithm decide which node to go to, by choosing one with the lowest heuristic. This means it is capable of finding solutions quickly, and saves memory by not expanding all nodes. However, this means that it might not return the most optimal path (path with lowest cost), since the smallest euclidean distance does not mean it will have the shortest path cost.

- **A* Search (AS)**

  A-Star is a search algorithm that also uses the same heuristic as GBFS, but also considers the actual cost to reach a node from the origin, $f(n) = g(n) + h(n)$. Hence it is somewhat of a more advanced version of GBFS. This method will return the most optimal path, assuming it is admissible (never overestimating the true cost) [8].

- **Custom methods**:
  - **CUS1**: Uniform Cost Search using cumulative path cost as g(n) and no heuristic.
  - **CUS2**: A variant of A* where g(n) is the number of steps, and h(n) is Euclidean distance.

Features, Bugs, and Missing Parts

Features Implemented:

- All 6 search algorithms as required
- Graph parsing from structured text files
- Repeated state checks for graph traversal
- Node and edge coordinate scaling for visualization
- GUI visualization of graph and path found using *pyglet* (optional)
- Command-line flags for debug and visualization
- Batch runner and graph test case generator

Missing or Not Implemented:

- Multi-goal sequential planning (e.g., visit all destinations in shortest total cost)

Known Bugs:

- None identified algorithm bugs during batch test runs
- Test case generator overwrites existing files with the same names

Special Notes:

- CUS1 is fully optimal but slower than A* due to lack of heuristic guidance
- CUS2 may perform slightly differently from A* due to step-based g(n)

Testing

To evaluate the performance and correctness of the implemented tree-based search algorithms, we used **20 automatically generated** test cases. This diverse testing approach allowed us to validate algorithm behavior under controlled scenarios as well as randomized, real-world-like graphs.

Automatically Generated Test Cases (Tests 1–10)

We created these test cases using a custom Python script (`generator.py`) that allowed fine control over:

- Number of nodes and destinations
- Coordinate range and edge costs
- Edge directionality (with optional bidirectional edges)

The configuration used to generate these 10 tests was:

```
num_nodes = 6

coord_range = (0, 10)

max_edges_per_node = 2

cost_range = (1, 10)

bidirectional_chance = 0.7

num_destinations = 2
```

This setup produced medium-complexity graphs with multiple routes, variable edge weights, and multiple goals, effectively stressing the decision-making capabilities of all six algorithms.

Overview of Observations:

- All algorithms successfully found paths when a valid route existed.
- In Test_1, DFS took a longer path, while others found a direct route.
- In Test_3 and Test_7, DFS explored more nodes before reaching the goal compared to A*, GBFS, and CUS methods.
- In Test_5 and Test_9, multiple algorithms arrived at different valid goals, indicating varied decision-making strategies.

The test logs confirmed the correctness and consistency of each algorithm and highlighted differences in strategy and performance.

Automatically Generated Test Cases (Tests 11–20)

Another set of 10 tests with more nodes were created to see how it would affect our algorithms:

The configuration used to generate these 10 tests was:

```
num_nodes = 50

coord_range = (0, 10)

max_edges_per_node = 2

cost_range = (1, 10)

bidirectional_chance = 0.3

num_destinations = 1
```

The results show the same observations as the previous 10 test cases. BFS returns paths with the least steps, while the informed algorithms might take longer paths but considers path cost. DFS is a "hit or miss", and UCS returns paths with least cost, not the shortest.

Insights

Our testing revealed meaningful differences in algorithm behavior:

- **DFS** is aggressive in its depth, often expanding unnecessary nodes, especially in graphs with misleading early branches. It sometimes reaches deeper goals that are further in cost. For example, in test 1, DFS takes a path of 2 ->6 -> 5 -> 4, instead of the fastest path of 2 -> 4.

- **BFS** reliably finds the shallowest path in terms of steps but may overexpand in dense graphs. It performed well in graphs with uniform edge costs.

- **GBFS** is fast and efficient when the heuristic is good, but risky when goal proximity is deceptive. It performed best when the graph layout aligned well with Euclidean distance.

- **A*** proved the most balanced — minimizing both cost and expansion — making it the most efficient and robust choice in almost all test cases.

- **CUS2**, which counts steps for g(n), often mimics A* but can slightly underperform in cost-efficiency. However, it's still heuristic-driven and generally performs well.

- **CUS1**, while consistently cost-optimal, tends to expand more nodes, especially when there are many cheap distractor paths before the best one.

In summary:

- DFS and BFS are faster compared to the other algorithms, but they tend to visit more nodes, hence trading time for memory.
- BFS is the most optimal search algorithm in terms of time and will always get the shortest path.

- CUS 1 will take more time than BFS and DFS and have more nodes visited, but will always return the path with least cost.
- CUS2 is a balance between A* and BFS, where it usually returns a path similar to A* with slightly less time taken. However, there was a case where it performed worse than both (test 19), as it took a significantly long path and higher path cost. Since it doesn't necessarily take the shortest path, nor the path with least cost, this algorithm isn't a great choice.

Our results may not have fully met our expectations, but that is understandable. Our code implementation would have been a big factor in our test cases. A possible explanation for how the informed algorithms took slightly more time is that it needed to calculate heuristics for all the nodes in the frontiers, making our implementation more inefficient.

We also observed that:

- All algorithms respected the repeated-state check constraint
- Tie-breaking was handled correctly (e.g., preferring smaller node IDs)

These findings reinforce the importance of algorithm choice depending on whether the priority is speed, optimality, or memory efficiency.

Research

As a research extension, we created `generator.py` to randomly build test cases and `run_batch.py` to automate evaluation. This allowed us to:

- Test algorithm stability under varied conditions
- Evaluate performance scalability
- Experiment with different graph densities and goal configurations

Future ideas include building a multi-goal optimization layer.

Conclusion

Six tree-based search algorithms, including DFS, BFS, GBFS, A*, and two custom strategies (CUS1 and CUS2), have been implemented and tested. This has given important information about the advantages and disadvantages of different search techniques in the context of route finding. Every algorithm was effectively built from the ground up, run via a command-line interface, and extensively tested with both manually written and randomly generated test cases.

Testing revealed that ignorant techniques such as DFS and BFS are easy to use and efficient in basic situations, but that they can be inefficient in terms of path cost or completeness depending on the graph structure. On the other hand, by using heuristics, well-informed approaches such as GBFS and A* demonstrated notable improvements in efficiency and optimality. Because it balanced the use of path cost and heuristic guidance, A* in particular was the most reliable in determining the least expensive path.

We developed our own algorithms, CUS1 and CUS2, to investigate other well-informed approaches. CUS1 is an A* inspired approach in which h(n) is the Euclidean distance to the closest goal and g(n) is the number of steps taken (instead of path cost). Shorter hop counts are prioritised in this method, which occasionally results in faster but not always less expensive routes. However, CUS2 uses Uniform Cost Search, which prioritises paths with the lowest cumulative cost but does not employ heuristics. This makes it optimal in terms of path cost, but when working with large graphs, it might be slower than A*. Both approaches were helpful in showing how changing the way cost is defined or removing heuristics can significantly change how search algorithms behave and perform.

All things considered, this assignment reaffirmed how crucial it is to choose the best search tactics depending on the context of the issue. The most dependable approach for finding the best route was A*. Additionally, to ensure algorithm correctness, particularly when handling cycles and unachievable goals, repeated state checks and robust test design were crucial.

Future enhancements might involve creating multi-goal plans (such as travelling to every location for the least amount of money), incorporating graphical debugging, and using these techniques on actual map data for additional research.

Acknowledgements/Resources

- **ChatGPT (OpenAI)** – Assisted in code explanations, debug guidance, and report formatting.
- **Python Documentation** – Reference for `heapq`, `subprocess`, `collections`, and `math`.
- **NetworkX** – Inspiration for graph generation structure
- **Swinburne Canvas & Lecture Notes** – Core algorithm knowledge base
- **Pyglet Documentation** – Helped with GUI/visualization setup

References

[1] Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.)

[2] Python Software Foundation, "3.7.3 Documentation," *Python.org*, 2019. https://docs.python.org/3/

[3] NetworkX, "NetworkX — NetworkX documentation," *networkx.org*. https://networkx.org/

[4] "Home — pyglet," *Pyglet.org*, 2024. https://pyglet.org

[5] OpenAI, "ChatGPT," *OpenAI*, 2025. https://openai.com/chatgpt

[6] Engineering, “When to Use Depth First Search vs Breadth First Search,” *Hypermode.com*, Jul. 23, 2024. https://hypermode.com/blog/depth-first-search-vs-breadth-first-search (accessed Apr. 11, 2025).

[7] “AI | Search Algorithms | Greedy Best-First Search,” *Codecademy*. https://www.codecademy.com/resources/docs/ai/search-algorithms/greedy-best-first-search

[8] m umar, “Exploring the A* Search Algorithm in Python: Pros, Cons, Applications, Use Cases, Time, and Space Complexity,” *Medium*, Aug. 19, 2023. https://webbazaar.medium.com/exploring-the-a-search-algorithm-in-python-pros-cons-applications-use-cases-time-and-space-44112c856134