SWINBURNE UNIVERSITY OF TECHNOLOGY

SCHOOL OF SCIENCE, COMPUTING AND ENGINEERING TECHNOLOGIES

Assignment 2A – Tree Based Search

COS30019 - Introduction to AI

Team Members:
Minh An Nguyen (104844794)
Duc Thang Nguyen (104776473)
Duc Tri Tran (105089392)
Quoc Phi Long Pham (104771041)

Tutor:
Gia Hy Nguyen

Semester 1, 2025

Contents

1 Instruction

Running an Individual Search

To execute a single search instance on a selected graph, open a command prompt, navigate to the project folder and run the following command:

```
python search.py <filename> <method>
```

Replace `<filename>` with the path to your graph file and `<method>` with one of the supported search algorithms (`BFS`, `DFS`, `AS`, `GBFS`, `CUS1` for UCS, or `CUS2` for BULB). For example:

```
python search.py PathFinder-Test.txt DFS
```

This will load the specified graph, execute the chosen search algorithm, and output the results (including the goal reached, the number of nodes generated, and the solution path) to the console.

Running the Search Experiment

The `run_search_experiments.py` script integrates three main functions: Generating test graphs, running search experiments on the generated graphs, analyzing and visualizing the results.To run the entire search experiments in one step run the following command:

```
python run_search_experiments.py --all
```

To generate graphs only:

```
python run_search_experiments.py --generate
```

To run experiments only:

```
python run_search_experiments.py --run
```

To analyze results only:

```
python run_search_experiments.py --analyze
```

For more custom options in running the search experiments:

```
python run_search_experiments.py --help
```

Output Files and Logging

- All results from individual searches and full experiment runs are printed to the console and saved in structured files within the `data` directory.
- Detailed logging is enabled throughout the execution. Log files (with timestamps) are written to the `logs` folder and include experimental steps, performance metrics, and any encountered errors.

2 Introduction

2.1 Problem Overview

The Route Finding Problem is a classic challenge in computer science and artificial intelligence with the objective is to find the most efficient path from a starting point to one or more destinations on a graph. In this project, we tackled this problem by implementing and testing various tree-based search algorithms. In our program, a search tree is generated by representing each state (or graph node) as an instance of a Node, which contains pointers to its parent, the action that led to it, and the cumulative path cost incurred so far. This design facilitates backtracking from any given goal node to the root in order to reconstruct the solution path. In addition, our implementation uses a graph abstraction where nodes are connected via weighted edges, and a separate parser module converts the text file specification into a working graph model.

We implemented four classic algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best-First Search (GBFS), and A* Search. DFS and BFS represent uninformed strategies, GBFS and A* employ heuristic function, with A* combining cost and heuristic for optimality. We also developed two custom strategies: Uniform Cost Search (UCS) ensures optimality by always expanding the lowest-cost node, and BULB (Beam search Using Limited discrepancy Backtracking) blends focused beam search with controlled exploration, offering improved performance in complex scenarios.

Together, these methods provide a robust and flexible approach to solving the Route Finding Problem, demonstrating both foundational understanding and thoughtful algorithmic design.

2.2 DFS (Depth-First Search)

Explores as far as possible along each branch before backtracking. It uses a stack (LIFO) and is good for finding solutions deep in the search tree but isn't optimal or complete in infinite graphs.

Algorithm 1 Depth-First Search (DFS)

```
function DepthFirstSearch(problem)
    Initialize empty set visited
    Initialize stack with initial node of problem
    while stack is not empty do
        Pop node from stack as current_node
        if reached end goal then
            return current_node
        end if
        Add current_node to visited
        Get children of current_node
        for each child in children do
            if child not in visited then
                Add child to stack
            end if
        end for
    end while
    return null
end function
```

2.3 BFS (Breadth-First Search)

Explores all nodes at the current depth level before moving to the next level. It uses a queue (FIFO) and guarantees finding the shallowest goal node, making it optimal for unweighted graphs.

Algorithm 2 Breadth-First Search (BFS)

```
function BreadthFirstSearch(problem)
    Initialize empty set visited
    Initialize queue with initial node of problem
    while queue is not empty do
        Pop node from queue as current_node
        if reached goal then
            return current_node
        end if
        Add current_node to visited
        Get children of current_node
        for each child in children do
            Add child to queue
        end for
    end while
    return null
end function
```

2.4 Best-First Search

Best First Search is a general informed search algorithm that uses a priority queue to explore nodes based on an evaluation function f, always selecting the most promising node first. It serves as the foundation for Uniform Cost Search (UCS), A*, and Greedy Best First Search, which differ only in their specific evaluation functions .

Algorithm 3 Best First Search

```
function BestFirstSearch(problem, f)
    Initialize node with problem's initial state
    Initialize priority_queue as min-priority queue with evaluation function f to calculate priority value.
    Add node to priority_queue
    Initialize empty set visited
    while priority_queue is not empty do
        Pop node with lowest f-value from priority_queue as current_node
        if reached goal then
            return current_node
        end if
        Add current_node to visited
        Get children of current_node
        for each child in children do
            if child not in visited and child not in priority_queue then
                Add child to priority_queue
            else if child in priority_queue then
                new_f_value ← calculate new f-value of child using function f
                old_f_value ← get the old f-value of child in priority_queue
                if new_f_value < old_f_value then
                    Remove child from priority_queue
                    Add child to priority_queue
                end if
            end if
        end for
    end while
    return null
end function
```

2.5 GBFS (Greedy Best-First Search)

Expands the node that appears closest to the goal based solely on a heuristic estimate $h(n)$. It uses a priority queue but is considered ”greedy” as it doesn’t account for path cost already traveled, making it fast but not optimal or complete.

Algorithm 4 Greedy Best First Search

```
function GreedyBestFirstSearch(problem)
    Define $f \leftarrow h$ function of problem                ▷ f(n) = h(n): heuristic only
    return BestFirstSearch(problem, $f$)
end function
```

2.6 A* Search

Expands nodes based on a combination of path cost so far $g(n)$ and estimated cost to goal $h(n)$. It uses a priority queue ordered by $f(n) = g(n) + h(n)$ and finds the optimal (least-cost) path if the heuristic is admissible/consistent.

Algorithm 5 A* Search

```
function AStarSearch(problem)
    Define $h \leftarrow h$ function of problem
    Define $f(n) \leftarrow g(n) + h(n)$
    return BestFirstSearch(problem, $f$)
end function
```

2.7 UCS (Uniform Cost Search)

Expands the node with the lowest total path cost from the start $g(n)$. It uses a priority queue ordered by $g(n)$ and guarantees finding the path with the minimum cumulative cost.

Algorithm 6 Uniform Cost Search

```
function UNIFORMCOSTSEARCH(problem)
    Define $f(n) \leftarrow g(n)$                          ▷ f(n) = g(n): path cost so far only
    return BestFirstSearch(problem, f)
end function
```

2.8 BULB (Beam Search Using Limited Discrepancy Backtracking)

Performs beam search by keeping only `beam_width` promising nodes at each step, guided by a heuristic. It allows pruned nodes to be revisited later with an increased ”discrepancy” count (up to a limit `max_discrepancies`), providing a way to recover from poor heuristic choices while maintaining limited memory usage.

Algorithm 7 Beam search Using Limited discrepancy Backtracking (BULB)

```
function BULBSEARCH(problem, beam_width, max_discrepancies)
    Define $f\_func(n) \leftarrow g(n) + h(n)$
    Create initial_node as DiscrepancyNode with problem's initial state, f_func, and 0 discrepancies
    Initialize priority_queue ordered by (discrepancies, f_value)
    Add initial_node to priority_queue
    Initialize empty set visited
    while priority_queue is not empty do
        Pop node with lowest (discrepancies, f_value) from priority_queue as current_node
        if current_node in visited then
            continue
        end if
        if reached goal then
            return current_node
        end if
        Add current_node to visited
        Create successors as list of DiscrepancyNodes from children of current_node
        if successors is empty then
            continue
        end if
        Sort successors by f_value
        Set beam ← first min(beam_width, len(successors)) nodes from successors
        Set pruned ← remaining nodes from successors
        for each node in beam do
            Add node to priority_queue
        end for
        for each node in pruned do
            if discrepancies of node < max_discrepancies then
                Create backtrack_node as DiscrepancyNode with node, f_func, and node.discrepancies + 1
                Add backtrack_node to priority_queue
            end if
        end for
    end while
    return null
end function
```

3 Features

Our Route Finding program implements a comprehensive solution with modular architecture and multiple search algorithms. Below are the key features of our implementation.

3.1 Graph Parsing and Problem Setup

Files Involved:

- `src/parser/graph_parser.py`
- `src/graph/graph.py`
- `src/graph/node.py`

What It Does: The parser reads graph specification files with sections for Nodes, Edges, Origin, and Destinations. It uses flexible file path resolution to support different environments. The parsed data is used to construct a `Graph` object and instantiate `Node` objects for search processes, providing the foundation for all search operations.

3.2 Implementation of Tree-Based Search Algorithms

Files Involved:

- `src/search_algorithm/search_algorithm.py`

What It Does: We implemented six search algorithms from scratch:

- **Depth-First Search (DFS):** Explores deeply using a stack and a visited set to prevent loops.
- **Breadth-First Search (BFS):** Uses a FIFO queue to find the shallowest solution first.
- **Greedy Best-First Search (GBFS):** Uses a heuristic to quickly guide search toward the goal.
- **A* Search (AS):** Combines path cost and heuristic for optimal solutions.
- **Uniform Cost Search / UCS (CUS1):** Selects nodes based on lowest cumulative cost.
- **Beam Search with Limited Discrepancy Backtracking Search / BULB (CUS2):** A hybrid approach balancing focused search with controlled exploration.

All algorithms share a common interface that returns the solution node, nodes expanded count, and nodes created count.

3.3 Automated Evaluation Experiments

Files Involved:

- `src/experiment/generate_graphs.py`
- `src/experiment/run_experiments.py`
- `src/experiment/analyze_results.py`

What It Does: Our pipeline streamlines testing through three stages:

- **Graph Generation:** Creates random test graphs with configurable parameters.
- **Running Experiments:** Executes all search algorithms on the generated graphs, collecting metrics like runtime, nodes expanded, and memory usage, etc.
- **Result Analysis:** Processes data to calculate averages and success rates, generating visualizations for key metrics.

3.4 Utility Modules and Data Structures

Files Involved:

- `src/utils/utils.py`

What It Does: Provides essential support functions including Euclidean distance calculation and a custom `PriorityQueue` implementation that enables efficient node retrieval and priority updates for informed search algorithms.

3.5 Command-Line Execution

Files Involved:

- `search.py`
- `run_search_experiments.py`

What It Does: The program operates from a command-line interface, supporting:

- Execute single search operations by supplying a graph file and a search method

  Example: `python search.py PathFinder-Test.txt BFS`.

- Run the complete experiment pipeline (graph generation, experiments, and analysis) using flags such as `--all` in the `run_search_experiments.py` script.

Users can customize parameters like node counts, graphs per size, etc. (all details are displayed when using flag `--help`). Detailed logs are maintained to facilitate debugging and performance tracking.

3.6 Performance Measurement

Where It's Implemented:

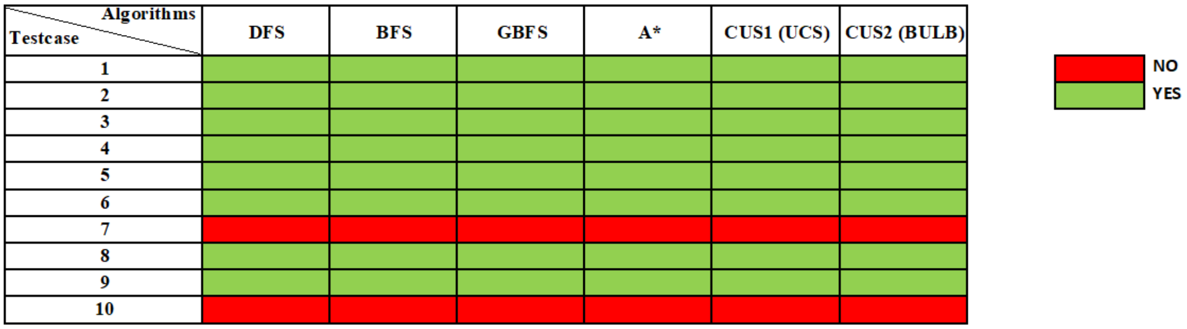
- Integrated into all experiment runs (see `src/search_algorithm/search_algorithm.py`)
- Experiment scripts monitor runtime and memory usage (using Python's `tracemalloc` in `run_experiments.py`)

What It Does: Each algorithm tracks metrics including nodes expanded, nodes created, computation time, and peak memory usage. The system conducts separate runs for runtime and memory tracking to ensure accurate performance measurement without interference.

4 Testing

4.1 Result for running testcases

Note: We put the testcase text files in `testcases/text` and there are visualizations for each testcase in `testcases/visualization`. Below is the table that shows the success of our algorithms in each test case.



Algorithms Testcase	DFS	BFS	GBFS	A*	CUS1 (UCS)	CUS2 (BULB)
1	🟩	🟩	🟩	🟩	🟩	🟩
2	🟩	🟩	🟩	🟩	🟩	🟩
3	🟩	🟩	🟩	🟩	🟩	🟩
4	🟩	🟩	🟩	🟩	🟩	🟩
5	🟩	🟩	🟩	🟩	🟩	🟩
6	🟩	🟩	🟩	🟩	🟩	🟩
7	🟥	🟥	🟥	🟥	🟥	🟥
8	🟩	🟩	🟩	🟩	🟩	🟩
9	🟩	🟩	🟩	🟩	🟩	🟩
10	🟥	🟥	🟥	🟥	🟥	🟥

🟥 NO
🟩 YES

4.2 Test Case 1: Loop Management Assessment

This test verifies that the search algorithms properly handle cycles and prevent infinite loops. It emphasizes the importance of visited-state tracking. The expected behavior is that UCS, A*, GBFS, and the custom BULB algorithm will quickly locate the exit through node 15, while DFS may explore deeper into the cycle before backtracking.

Outcome: Cost-based algorithms detected the exit with minimal node expansions, affirming proper loop detection. DFS expanded more nodes (showing deeper traversal) but ultimately avoided an infinite loop.

4.3 Test Case 2: Equal-Cost Path Evaluation

This test assesses tie-breaking when multiple paths have equal costs. Two paths exist, from node 2, one path through node 6 and another through node 7, each with a total cost of 15. The test requires that DFS and BFS follow their natural expansion order, UCS choose the lower node ID (node 6), and informed methods (A*, GBFS, BULB) favor the path with a more promising heuristic (node 7).

Outcome: UCS selected the path via node 6, while A*, GBFS, and BULB confirmed a preference for node 7 due to their heuristic evaluations. Tie-breaking rules based on node ordering were maintained as specified.

4.4 Test Case 3: Deceptive Heuristic Validation

This test examines how heuristic-dependent algorithms perform when presented with misleading geometric shortcuts. It presents a shortcut where the heuristic suggests a shorter route; however, the actual path cost is higher. The expectation is that algorithms incorporating path cost (UCS, A*, BULB) will find the optimal route, while GBFS, which relies solely on the heuristic, will be led astray.

Outcome: GBFS chose a more expensive route due to the deceptive heuristic. A* and BULB correctly balanced heuristic with accumulated cost to determine the best path, proving the importance of including cost information.

4.5 Test Case 4: Worst-Case Uninformed Search

This test evaluates the performance of uninformed algorithms under adverse conditions. In a linear graph with reverse node ordering, the test expects DFS and BFS to expand almost the entire graph, whereas UCS, A*, and BULB should more efficiently target the goal by using cost and heuristic data.

Outcome: DFS and BFS expanded all available nodes, demonstrating their vulnerability to unfavorable orderings. UCS, A*, and BULB found the optimal path with significantly fewer expansions, underscoring the benefits of cost and heuristic guidance.

4.6 Test Case 5: Elementary Route Discovery

This test verifies basic route discovery when multiple destination choices are available. It features several potential routes: cost-sensitive algorithms (UCS, A*, BULB) are expected to choose the path via nodes 2, 3, 6, and 9 due to lower overall cost, while BFS will systematically explore levels, and DFS will deeply follow a specific branch. GBFS is expected to select a route (2, 5, 8) based on geometric proximity.

Outcome: Each algorithm behaved according to its inherent strategy, confirming that the system differentiates between cost-based and traversal-based decision-making.

4.7 Test Case 6: Tie Resolution Assessment

This test assesses how the algorithms resolve ties when nodes have equivalent heuristic values and similar costs. With multiple nodes on different branches having identical evaluation values, the expected outcome is that all algorithms identify the correct path (2, 3, 6, 8) by expanding nodes in ascending order (with lower node IDs prioritized).

Outcome: All algorithms consistently expanded nodes in ascending order and correctly resolved ties by choosing the path through node 8, in accordance with the tie-breaking protocols specified.

4.8 Test Case 7: No Valid Path Handling

This test ensures the program responds gracefully when no valid path exists. Node 6 is isolated in this test case. The system should detect that the goal is unreachable and report ”No solution,” while also indicating the farthest reachable node.

Outcome: Each algorithm terminated the search quickly (within 3 node expansions) and reported that no valid path existed, demonstrating proper error handling.

4.9 Test Case 8: Goal Path Verification

This test confirms that the algorithms can detect and avoid seemingly better paths that are actually invalid, opting instead for a viable alternative. Although goal 4 initially appears promising based on heuristic and cost, there is no complete path to it. The predicted correct solution is the path 5, 7, 9, meaning that despite initial confusion, all algorithms should eventually settle on goal 9.

Outcome: All search methods eventually bypassed the invalid path to goal 4 and resolved the valid route to goal 9, as expected.

4.10 Test Case 9: Initial State as Destination

This test validates immediate termination when the origin is also the destination. In this scenario, node 2 is both the starting point and one of the destinations. The anticipated behavior is immediate identification of the goal state without further node expansion.

Outcome: Every algorithm returned the starting node immediately, with zero additional node expansions, thereby confirming correct goal state recognition.

4.11 Test Case 10: Empty Graph

This test validates that the program handles an empty graph correctly without crashing or entering undefined behavior. It involves a graph with no nodes or edges. The expected behavior is for all algorithms to immediately terminate, as there is no valid path to compute.

Outcome: All algorithms correctly detected that the graph is empty and printed as output without attempting any expansions.

5 Evaluation and Insight

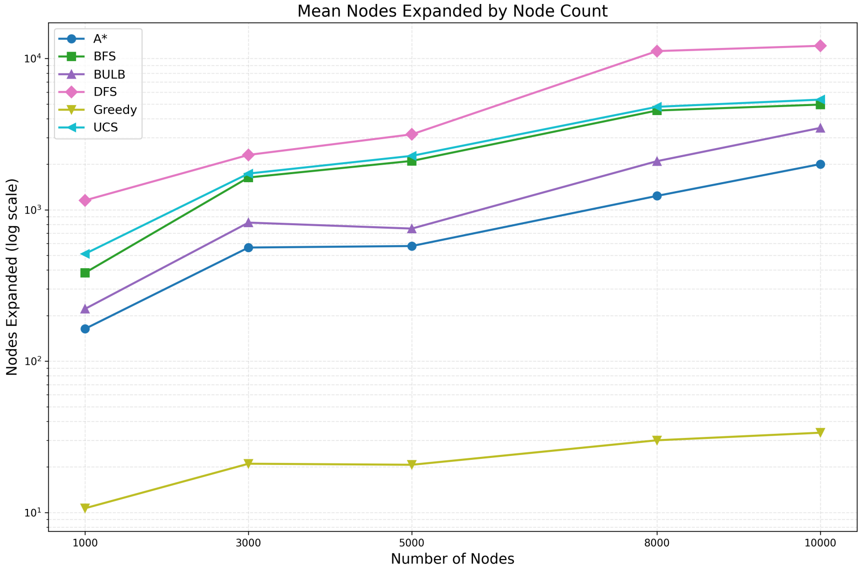
To evaluate the different search algorithms, we ran tests on a Macbook Air M1 by using the experiment script (discussed in section 6.2). For each algorithm applied to various weighted graph problems, we carefully measured several key factors:

- **Path Length:** How many steps were in the solution path found.
- **Path Cost:** The total weight of the path found.
- **Nodes Expanded:** How many nodes the algorithm looked at during its search.
- **Runtime:** How much time the algorithm took to finish.
- **Memory Usage:** How much computer memory the algorithm needed.

Our test graphs were set up with each node having between 10 and 20 connecting edges. To ensure our results were reliable, we created 3 different graphs for each size category. The final performance figures for each size are the average results from these three graphs. For the BULB algorithm specifically, we configured it with a `beam_width` of 50 and `max_discrepancies` set to 8.

We gathered all this data from our trials and processed it to create the visual charts presented later in this report. These charts help us understand the trade-offs: they show how each search strategy balances finding good, low-cost paths against the time and memory it takes, especially as the graphs become larger and more complex.

5.1 Node Expanded



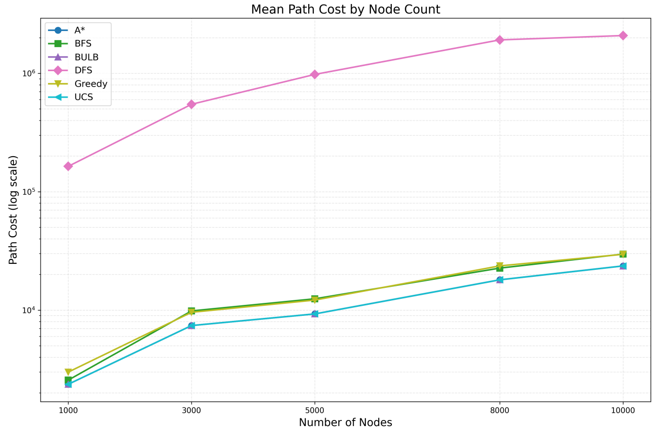
In our evaluation results, we observed that **GBFS** expands the fewest nodes across all test cases. This efficiency comes from its simple heuristic function $f(n) = h(n)$, which strictly follows the estimated distance to the goal without considering path cost.

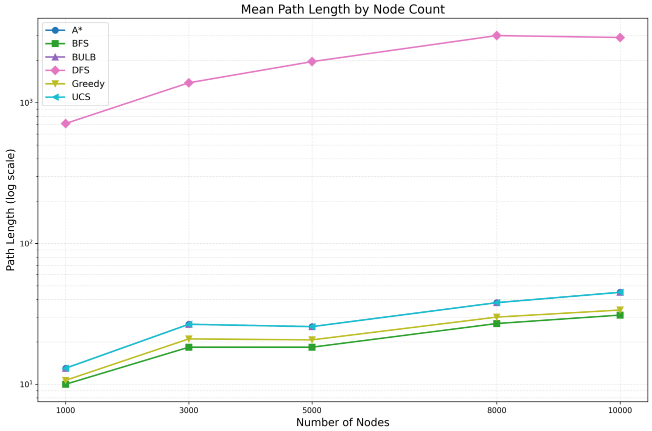
Next in performance is **A***, which expands more nodes than Greedy but fewer than other algorithms. We attribute this to **A***'s balanced approach using $f(n) = g(n) + h(n)$, considering both path cost and heuristic estimate. **BULB** shows moderate performance in our tests. While it effectively prunes some unpromising branches, it sometimes explores paths that initially seem unpromising but later prove efficient.

Both **BFS** and **UCS** expand significantly more nodes as they methodically explore the search space, **BFS** by breadth and **UCS** by considering path costs. Our results show these algorithms perform similarly in terms of node expansions.

DFS performs worst in our evaluation, expanding the most nodes across all graph sizes. Without heuristic guidance, **DFS** blindly explores paths deeply, often wasting effort on irrelevant branches.

5.2 Path Cost & Path Length


Mean Path Cost by Node Count


Mean Path Length by Node Count

- Path cost: In our evaluation of path costs, we found that **UCS**, **A*** and **BULB** perform similarly well, consistently finding near-optimal solutions across all graph sizes. This is expected behavior because **UCS** is guaranteed to find the lowest-cost path by exploring nodes in order of increasing path cost. **A*** combines this approach with a heuristic, which maintains optimality while improving efficiency. **BULB**'s limited discrepancy search strategy effectively explores promising paths first, leading to similar optimal results.

We observed that **GBFS** and **BFS** show moderate performance for path cost. **GBFS** sometimes misses optimal paths because it only considers the heuristic value (distance to goal) without accounting for the cost accumulated so far. **BFS** performs less efficiently in weighted graphs because it expands nodes based solely on their level in the search tree, ignoring edge weights entirely.

DFS clearly performs worst for path cost, with values significantly higher than all other algorithms. This occurs because DFS blindly explores deeply along arbitrary paths without considering cost or distance to goal, often finding extremely suboptimal solutions before reaching the target node.

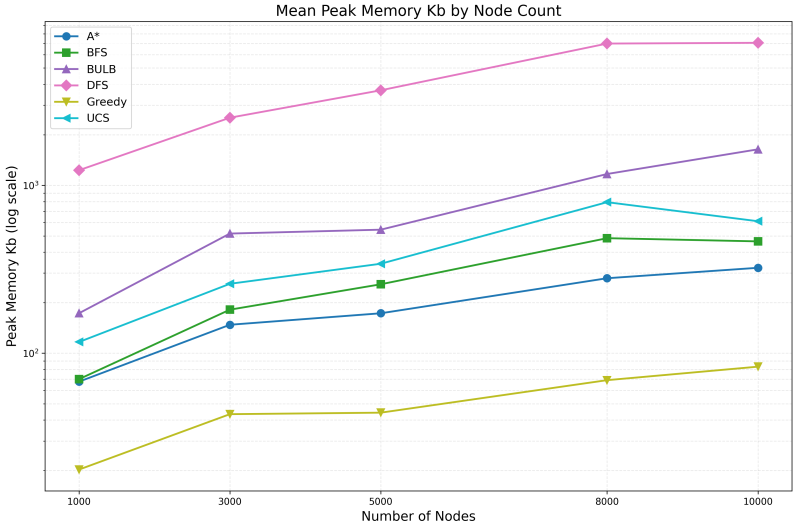
- Path length: When analyzing path length, we discovered that **BFS** consistently finds the shortest paths in terms of hop count. This result directly follows from **BFS**'s core property - it always explores all nodes at depth d before any node at depth $d+1$, guaranteeing the fewest-edge path in any graph.

GBFS performs close to **BFS** in path length because its heuristic (estimated distance to goal) tends to guide the search toward nodes that bring it closer to the destination in fewer steps. Although not guaranteed to find minimum-length paths, the heuristic often favors shorter routes.

UCS, **A*** and **BULB** show moderate path lengths because these algorithms prioritize minimizing total cost rather than minimizing the number of edges. In weighted graphs, the shortest path by edge count often differs from the optimal path by cost, explaining this trade-off in our results.

DFS again performs worst, producing significantly longer paths than all other algorithms. This poor performance stems from **DFS**'s tendency to follow deep paths regardless of direction, often taking numerous unnecessary detours before reaching the goal node."

5.3 Peak Memory Usage



In our evaluation of memory usage, we observed that **DFS** consumes significantly more memory than all other algorithms. While **DFS** is theoretically memory-efficient with space proportional to search depth $O(h)$, our results show otherwise. This unexpected outcome occurs because our test graphs are dense (10-20 edges per node), forcing **DFS** to store many long paths in memory before reaching the goal.

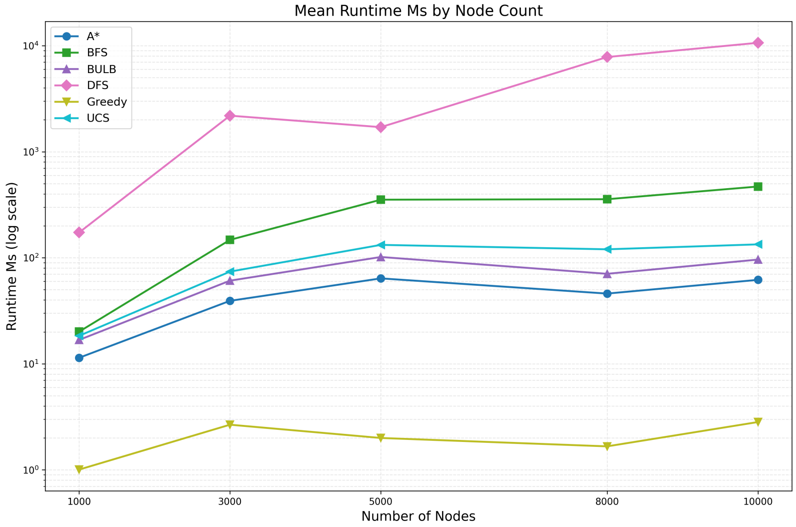
BULB ranks second in memory consumption, showing steady growth as graph size increases. This pattern reflects **BULB**'s hybrid search strategy that balances depth and breadth exploration, discarding some states between iterations but still requiring substantial memory for each deep search iteration.

UCS and **BFS** show moderate memory usage, with **UCS** slightly higher than **BFS** in most cases. We noticed **UCS** memory consumption decreases slightly at larger graph sizes (8,000-10,000 nodes), likely because it finds solutions earlier in these larger graphs, reducing its priority queue size. **BFS** must store all nodes at the current frontier, which grows steadily with graph size.

A* demonstrates more efficient memory usage than **BFS** and **UCS** while maintaining solution optimality. This efficiency comes from its informed search approach, which prioritizes promising paths and prunes less promising ones early.

GBFS clearly uses the least memory among all algorithms tested. We attribute this to its aggressive use of the heuristic function, which keeps only a small subset of most promising nodes in memory at any time. Unlike **A***, **GBFS** doesn't need to track path costs, further reducing its memory footprint.

5.4 Run Time



Regarding runtime performance, we observed that **DFS** is significantly slower than all other algorithms. Similar to our memory usage findings, **DFS**'s poor performance stems from its extensive exploration of paths in dense graphs, requiring substantial computational effort without effective pruning.

BFS shows the second highest runtime, growing steadily as graph size increases. This matches our expectations since **BFS** must process all nodes at each level before proceeding deeper, creating considerable computational overhead as the graph expands.

UCS demonstrates moderate runtime performance, positioned between **BFS** and the more efficient informed search algorithms. The priority queue operations in **UCS** add some overhead compared to the faster algorithms, but its directed exploration still outperforms the uninformed **BFS** approach.

BULB and **A*** show similar runtime profiles, with **A*** slightly faster across most graph sizes. Both algorithms benefit from heuristic guidance to reduce unnecessary exploration, explaining their improved efficiency over uninformed methods.

Greedy Best-First Search is clearly the fastest algorithm in our tests, with runtimes barely visible on the logarithmic scale compared to others. As with its memory efficiency, this exceptional speed results from **Greedy**'s single-minded focus on the heuristic function, exploring only the most promising paths without tracking alternative routes. However, this speed comes at the cost of solution optimality, as we observed in our path cost analysis

6 Research

6.1 Random graph generator

In our implementation, we've developed a method for generating random graph problems that maintain realistic structure while ensuring the Euclidean distance heuristic remains admissible.

```
class MultigoalGraphProblem:
    @classmethod
    def random(
        cls,
        num_nodes=10,
        min_edges_per_node=8,
        max_edges_per_node=16,
        grid_size=100,
        num_destinations=1,
        curvature=lambda: random.uniform(1.1, 2.0),
        max_distance_factor=2.0,
        ensure_connectivity=True,
    ):
```

6.1.1 Overview:

This function involves 3 main steps:

1. **Node and Goal Selection Strategy:** When creating random graph problems, We begin by generating `num_nodes` with random coordinates distributed across a 2D grid of `grid_size`. For selecting origins and destinations, we prioritize destinations from the furthest 90% of nodes relative to the origin, we create more challenging and realistic search problems while maintaining an element of randomness. This better mimics real-world scenarios where journeys typically involve significant distances.

2. **Edge Generation with Spatial Coherence:** For edge creation, we implement a proximity-based connection strategy that simulates real-world networks. Each node connects to between `min_edges_per_node` and `max_edges_per_node` nearby neighbors, prioritizing connections by distance. We also employ `max_distance_factor` to limit connection distances based on the average nearest-neighbor distance. This approach creates graphs resembling transportation networks, where nearby locations have direct connections while reaching distant locations requires traversing multiple edges.

3. **Ensuring Solvability:** Final step is to make sure all problems are solvable by checking if paths exist from the origin to all destinations. If not, we create the path with some specified mechanisms (which will be discussed later) to ensure the heuristic function always admissible.

6.1.2 Maintaining Heuristic Admissibility Mechanisms

We also implement several mechanisms that work together to maintain an admissible heuristic while generating random graph problems.

1. **Curvature Function:** We multiply the straight-line distance by a random factor greater than 1.0 (typically between 1.1 and 2.0) when setting edge weights:

```
    weight = distance(locations[node], locations[target]) * curvature()

```

This ensures the actual path cost is always greater than the Euclidean estimate, keeping our heuristic admissible.

2. **Closest Node Connection Strategy:** Our edge creation strategy prioritizes linking nodes to their closest neighbors. By connecting only nearby nodes, we create a graph that respects the underlying spatial relationships. This prevents creating unrealistic ”shortcuts” that could compromise the spatial relationships.

```
    potential_targets.sort(key=lambda x: distance(locations[node], locations[x]))
    targets = potential_targets[:num_edges]

```

3. **Maximum Distance Limitation**: We calculate a reasonable maximum connection distance based on the average nearest-neighbor distance times `max_distance_factor` (default 2.0) to establish reasonable connection limits and filter potential targets to only include those within the maximum distance. This ensures the graph topology respects physical constraints and further prevents distant connections that could create problematic shortcuts.

```
    avg_neighbor_distance = sum(all_distances) / len(all_distances)
    return avg_neighbor_distance * max_distance_factor

```

4. **Strategic Path Ensuring:** When ensuring paths exist between origins and destinations, we use a sophisticated approach:

   - We first identify all nodes reachable from the origin.
   - We then find the closest pair of nodes between reachable and unreachable sets.
   - We add an edge between these nodes, applying the curvature factor.
   - We recursively check if this creates a path to the destination.
   - Only as a last resort do we create a direct connection from the origin to the destination.

   This approach maintains spatial consistency by adding the most natural connections first. Only if necessary do we add direct paths from origin to destinations as a last resort.

Through these methods, we create graph problems that are both challenging and realistic, while ensuring they remain solvable with admissible heuristic search techniques like A*. The resulting graphs maintain a natural structure that resembles real-world networks, making them excellent test cases for evaluating search algorithms.

6.1.3 How the Heuristic Admissibility is Maintained

The Euclidean distance heuristic function in our code is defined as:

```
    def h(self, node):
        return min(
            distance(self.locations[node_state], self.locations[goal])
            for goal in self.goals
        )
```

This heuristic remains admissible in my random method because:

1. Edge weights are always greater than Euclidean distances due to the curvature multiplier (≥ 1.1)
2. Graph topology preserves spatial relationships by connecting closest nodes first
3. Distance constraints prevent unrealistic shortcuts across the graph
4. Connectivity is ensured with minimal distortion by connecting closest pairs between reachable and unreachable partitions

Together, these mechanisms ensure that the straight-line Euclidean distance will always be less than or equal to the actual path cost, which is the definition of an admissible heuristic.

6.2 Additional Feature: Experiment Script for Automatic Evaluation and Visualization

To aid in the evaluation of six search algorithms on weighted graphs, we implemented a specialized command-line experiment script `run_search_experiments.py`. This script automates the process of generating graphs, running algorithm tests, and visualizing the results, ensuring consistency and efficiency in our analysis. Here’s how it works:

6.2.1 Overview of the Experiment Script

The experiment script operates in three main steps:

1. **Graph Generation:** This step creates random weighted graphs with customizable parameters such as node count, edge density, and the number of graphs to generate for each configuration. The generated graphs are saved as files in a timestamped directory for future reference.
2. **Algorithm Evaluation:** Each of the six search algorithms is tested on the generated graphs, yielding metrics such as runtime, path cost, path length, number of nodes expanded, and memory usage. Memory tracking and runtime measurement are separately executed to minimize interference in results. Results are logged and saved in structured JSON files within a separate timestamped directory.
3. **Data Visualization:** Using the evaluation results, the script generates visualizations to compare algorithm performance on various metrics. Charts are created in linear or logarithmic scale and saved as image files in the analysis directory.

6.2.2 Key Features

- **Automation:** The script allows running all steps sequentially (`--all`) or individual steps (`--generate`, `--run`, `--analyze`) as needed.
- **Checkpoint System:** All experiment data (graphs, results, visualizations) is organized by timestamps to facilitate reproducibility and comparison.
- **Logging System:** The script features a robust logging system to provide transparency and aid debugging. Logs are written to both the console and timestamped log files in the `logs` directory. Each step of the experiment (graph generation, evaluation, and analysis) has its own log file to simplify tracking. Logs include details such as:
  - Start and end times of each step.
  - Parameters used during graph generation and evaluation.
  - Issues encountered, such as errors in graph loading or algorithm processing.
- **Customizability:** Users can control parameters such as graph size, edge density, specific algorithms, and output formats using command-line arguments.

6.2.3 Insights and Results

This comprehensive system allows us to gather a large amount of useful data and create detailed charts, which are analyzed in the ”Insight” section of the report. The insights gained from these results offer a deeper understanding of each algorithm’s performance, scalability, and efficiency, making this feature an invaluable element of our study.

7 Conclusion

The Route Finding problem presents a variety of challenges, including finding optimal paths, managing computational resources, and handling diverse graph structures. Based on our implementation and testing results, we conclude that the choice of the best search algorithm depends on the specific requirements of the problem, such as optimality, efficiency, and scalability.

7.1 Best Search Algorithm for Route Finding

Among the six algorithms implemented, **A* Search** often emerged as highly effective for this Route Finding problem when provided with a good heuristic. Its evaluation function, $f(n) = g(n) + h(n)$, strategically balances the actual path cost incurred so far $g(n)$ with an estimated cost to the goal $h(n)$. This guidance typically makes it computationally efficient compared to uninformed searches, and if the heuristic is admissible (and consistent for optimal efficiency), **A*** guarantees finding the optimal (lowest cost) path.

However, **Uniform Cost Search (UCS)** serves as a crucial baseline and a reliable alternative, particularly when designing an effective heuristic is challenging or not possible. **UCS** guarantees finding the optimal solution by systematically expanding the node with the lowest cumulative path cost $g(n)$. Its primary drawback is that, lacking heuristic guidance, it may explore significantly more nodes than **A***, potentially increasing computational time and memory usage, especially in large graphs with uniform edge costs near the start.

Beam search Using Limited discrepancy Backtracking (BULB), our custom implementation, demonstrated a compelling trade-off between performance and completeness. By limiting the search width (beam width), it can be significantly faster and use less memory than **A*** or **UCS** on large graphs. The limited discrepancy backtracking adds robustness, allowing it to recover from potentially poor heuristic choices that might mislead a standard beam search. This makes **BULB** a strong candidate for very large graphs or when heuristics are known to be imperfect. However, its performance is highly dependent on careful tuning of the `beam_width` and `max_discrepancies` parameters for the specific problem, and unlike UCS and A* (with an admissible heuristic), BULB does not guarantee finding the optimal solution.

7.2 Improving Performance

While our implementation meets the assignment requirements and performs well across test cases, there are several ways to improve performance further:

- **Dynamic Heuristic Adjustment:** The quality of A* and GBFS heavily depends on the heuristic function. Developing dynamic heuristics that adapt during runtime based on graph characteristics could improve search efficiency.
- **Memory Optimization:** For large-scale graphs, memory usage can become a bottleneck. Techniques like iterative deepening or external memory management could help reduce memory consumption while maintaining performance.
- **Parallelization:** Implementing parallelized versions of search algorithms could significantly speed up computation by exploring multiple branches simultaneously.
- **Beam Width Tuning:** The performance of BULB is sensitive to beam width selection. Automating this parameter based on graph density or other characteristics could enhance its adaptability.
- **Hybrid Algorithms:** Combining elements of informed and uninformed search strategies into hybrid algorithms could provide more robust solutions for complex graphs.

7.3 Final Thoughts

In conclusion, A* Search stands out as the most suitable algorithm for solving Route Finding problems due to its balance between optimality and efficiency. However, alternative approaches like UCS or BULB may be preferable in specific scenarios where memory constraints or heuristic reliability are critical factors. By incorporating advanced techniques such as dynamic heuristics, parallelization, and memory optimization, future iterations of this program could achieve even greater performance and scalability. This project has provided valuable insights into search algorithm design and application while laying a strong foundation for further exploration in advanced pathfinding techniques.

8 Acknowledgement

Russell And Norvig's "Artificial Intelligence - A Modern Approach" (aima-python): Gave information about code structures and implementation of the algorithms (Norvig, Russell, & Contributors, 2025).

Perplexity.ai: Assisted in the creation of this project by providing detailed explanations and insights, also assisted in writing the report (Perplexity AI, 2025).

Khoi Nguyen Pham: A friendly Swinburne student, we gained some idea creating the testcases while discussing with him (Pham, 2025).

Mr Hy Nguyen & Mr Bao Vo: Our tutor and lecturer who gave us much help through the process of this project (Nguyen, 2025; Vo, 2025).

Furcy, David and Koenig, Sven: The author of the original paper of the Beam search Using Limited discrepancy Backtracking (BULB). (Furcy & Koenig, 2005).

References

Furcy, D., & Koenig, S. (2005). Limited discrepancy beam search. In *Ijcai* (pp. 125–131).

Nguyen, H. (2025). *Our tutor.* Retrieved from https://www.linkedin.com/in/nguyengiahy/ (Accessed: 2025-04-11)

Norvig, Russell, & Contributors. (2025). *AIMA-Python: Python Code for Artificial Intelligence: A Modern Approach.* Retrieved from https://github.com/aimacode/aima-python (Accessed: 2025-04-11)

Perplexity AI, I. (2025). *Perplexity.ai.* Retrieved from https://www.perplexity.ai (Accessed: 2025-04-11)

Pham, K. N. (2025). *Our friendly friend in Swinburne University.* Retrieved from https://www.linkedin.com/in/nguyen-nevan-pham-317496321/ (Accessed: 2025-04-11)

Vo, B. (2025). *Our lecturer.* Retrieved from https://www.linkedin.com/in/bao-vo-28928344/ (Accessed: 2025-04-11)