Assignment 2A Report

COS30019 – Introduction to Artificial Intelligence

Title: *Traffic-Based Route Guidance System – Tree-Based Search*

ASSESSMENT DETAILS

Unit title	Introduction to AI	Lab Group	Group 5	Office use only
Unit code	COS30019	Due date	13/4/2025	
Name of lecturer/tutor	Feixue Yan			
Class name	Studio 1 - 11, Tuesday 2:30pm			Faculty or school date stamp

STUDENT(S) DETAILS

	Student Name(s)	Student ID Number(s)
(1)	Anh Vu Le	104653505
(2)	Duc Thuan Tran	104330455
(3)	Lattaphonh Thipsengchanh	103848915
(4)	Harrishraj Rajandran	104332312
(5)		
(6)		

DECLARATION AND STATEMENT OF AUTHORSHIP

1. I/we have not impersonated, or allowed myself/ourselves to be impersonated by any person for the purposes of this assessment.
2. This assessment is my/our original work and no part of it has been copied from any other source except where due acknowledgement is made.
3. No part of this assessment has been written for me/us by any other person except where such collaboration has been authorised by the lecturer/tutor concerned.
4. I/we have not previously submitted this work for this or any other course/unit.
5. I/we give permission for my/our assessment response to be reproduced, communicated, compared and archived for plagiarism detection, benchmarking or educational purposes.

I/we understand that:

6. Plagiarism is the presentation of the work, idea or creation of another person as though it is your own. It is a form of cheating and is a very serious academic offence that may lead to exclusion from the University. Plagiarised material can be drawn from, and presented in, written, graphic and visual form, including electronic data and oral presentations. Plagiarism occurs when the origin of the material used is not appropriately cited.

Student signature/s

I/we declare that I/we have read and understood the declaration and statement of authorship.

(1)	Vu	(4)	Harrishraj
(2)	Duc Thuan	(5)	
(3)	Lattaphonh	(6)	

Table of Contents

1. Instructions

The **Route-Finding Problem** is a well-established challenge in artificial intelligence that involves determining the most efficient path from a starting point (origin node) to a destination (goal node) within a connected system. This system can be modeled as a **graph**, where nodes represent places (e.g., intersections or cities), and edges represent routes between them. Solving this problem efficiently is critical in domains such as transportation, logistics, autonomous navigation, and smart city planning.

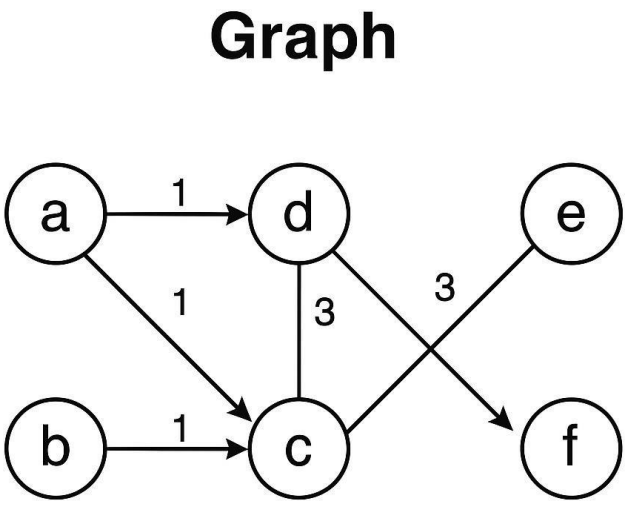

Figure 1: Graph

FIGURE 1: A DIRECTED, WEIGHTED GRAPH WHERE EACH EDGE HAS A COST AND DIRECTION. THIS STRUCTURE IS USED IN ALL TEST CASES FOR THE ROUTE-FINDING SYSTEM TO MODEL TRAFFIC-BASED MAPS.

1.1. Graph Concepts

A **graph** is a data structure made up of:

- **Nodes (vertices)**: Represent locations.
- **Edges (arcs)**: Represent possible transitions between locations.
- **Edge weights**: Represent the cost of travel (e.g., distance, time, or energy) between nodes.

- **Directed edges** (as shown in the figure) indicate one-way roads or restricted paths, relevant for real-world traffic-based routing.

In this project, all test cases are parsed from .txt files that define a **directed, weighted graph**, like the one in the figure above. The system interprets these graphs to identify the shortest or most optimal route to a target node.

1.2. Tree-Based Search Concepts

To traverse the graph and find valid paths, the program builds a **search tree** from the origin node. A search tree is an abstract structure where each **branch** represents a possible sequence of actions or node expansions.

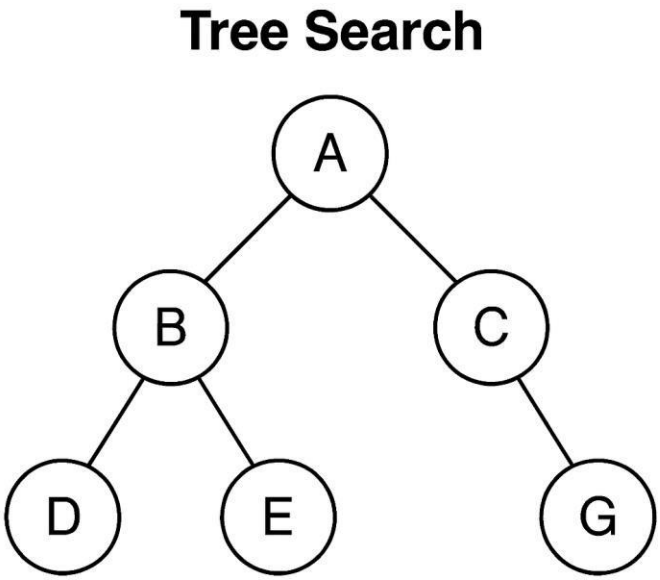

Figure 2: Search tree

FIGURE 2: A TREE-BASED STRUCTURE ILLUSTRATING HOW SEARCH ALGORITHMS EXPLORE PATHS BY EXPANDING NODES. EACH LEVEL SHOWS A DIFFERENT LAYER OF DEPTH IN THE SEARCH SPACE.

Each node in the tree corresponds to a state in the search process. As the tree expands:

- **The root** represents the origin node.
- **Each child** node represents a possible move or transition from its parent.
- The **goal node** is found by traversing from the root to a leaf node that matches the destination.

This tree structure allows different **search algorithms** to systematically explore possible paths, using various strategies to balance efficiency, accuracy, and memory usage.

1.3. Implemented Algorithms (Brief Overview)

To evaluate different approaches to the Route-Finding Problem, six algorithms were implemented — categorized into **uninformed** (blind) and **informed** (heuristic-based) search methods:

Uninformed Search

- **DFS (Depth-First Search)**: Explores one path deeply before backtracking. Memory-efficient but may miss shorter paths
- **BFS (Breadth-First Search)**: Explores all neighbors at one level before moving deeper. Guaranteed to find the shortest path in steps.
- **CUS1 – Iterative Deepening DFS**: A hybrid of DFS and BFS that limits search depth, improving completeness while conserving memory.

Informed Search

- **GBFS (Greedy Best-First Search)**: Prioritizes nodes closest to the goal using only heuristic estimates. Fast but not always optimal.
- **AS (A* Search) **: Balances path cost and heuristic using f(n) = g(n) + h(n). It is complete and optimal if the heuristic is admissible.
- **CUS2 – Weighted A***: A modified A* using f(n) = g(n) + w*h(n) with w > 1, increasing speed while accepting slightly suboptimal paths.

These algorithms are implemented as individual modules and tested across 10 different graph scenarios, allowing for performance comparison based on search time, path cost, and node expansions.

2. Features, Bugs, and Missing Items

2.1. Feature

This project implements a modular, testable, and comparative framework for solving the Route-Finding Problem using six search algorithms, each rooted in core AI concepts taught in Weeks 2–6. All algorithms are structured around the idea of graph traversal, where each node represents a location, and edges represent roads with costs.

🕮 *Depth-First Search (DFS)*

Theory:
DFS explores as deep as possible along one branch before backtracking. It uses a **stack-based (LIFO)** structure and is memory efficient, but not optimal and may get stuck in deep or infinite paths if cycles aren't handled.

Implementation:

- Implemented in ***"algorithms/dfs.py"***

- Uses a stack and a ***"visited"*** set
- Stops when the goal is found or all paths are explored

Example: On ***"test4.txt"***, DFS quickly finds a solution but may miss the optimal path.

🖋 *Breadth-First Search (BFS)*

Theory:
BFS explores all nodes at the current depth before moving deeper. It guarantees the **shortest path in terms of steps** if edge costs are equal. It uses a **queue (FIFO)** structure.

Implementation:

- Found in ***"algorithms/bfs.py"***
- Queue-based traversal with visited-state tracking
- Guarantees completeness and optimality in unweighted graphs

Example: *"test2.txt"* demonstrates BFS returning a short and valid path with consistent node depth expansion.

🖋 *Greedy Best-First Search (GBFS)*

Theory:
An informed search using only the heuristic value ***"h(n)".*** GBFS chooses the node that appears closest to the goal but **may return suboptimal paths**.

Implementation:

- Located in ***"algorithms/gbfs.py"***
- Priority queue orders nodes by heuristic
- Only considers the heuristic, not path cost

Example: In ***"test6.txt"***, GBFS reaches the goal quickly but returns a longer path than A*.

🖋 *A* Search (AS)*

Theory:
A* combines actual cost ***"g(n)"*** and heuristic ***"h(n)"*** using the evaluation function ***"f(n) = g(n) + h(n)".***
It is both **complete and optimal** when ***"h(n)"*** is admissible.

Implementation:

- In ***"algorithms/astar.py"***
- Uses a priority queue ordered by ***"f(n)"***
- Path cost and heuristic are dynamically updated

Example: On ***"test8.txt",*** A* provides the shortest-cost path while exploring fewer nodes than BFS.

🔹 *Custom Algorithm 1 – CUS1 (Iterative Deepening DFS)*

Theory:
IDDFS combines DFS and BFS by performing DFS with a limited depth and increasing it iteratively. It is **complete**, **low memory**, and finds shortest paths in terms of steps.

Implementation:

- Implemented in ***"algorithms/cus1.py"***
- Iteratively increases depth until a goal is found
- Stops early if a shallow goal is reachable

Example: ***"test1.txt"*** and ***"test3.txt"*** demonstrate efficient use of memory and completeness.

🔹 *Custom Algorithm 2 – CUS2 (Weighted A*)*

Theory:
Weighted A* biases search toward the goal faster by modifying the evaluation function:
"f(n) = g(n) + w * h(n)", where ***"w > 1"***. This sacrifices optimality for speed.

Implementation:

- Located in ***"algorithms/cus2.py"***
- ***"w"*** is set to 2.0 for faster convergence
- Prioritizes promising paths with heuristic bias

Example: ***"test5.txt"*** completes significantly faster than A*, with slight cost increase in the result.

🔹 *Additional Features*

- **CLI Interface**: Fully interactive via terminal commands (e.g., ***"python search.py test3.txt AS"***)
- **Batch Testing**: ***"run_all_test.py"*** evaluates all algorithms over 10 test cases automatically
- **Structured Output**: Includes file name, method used, total nodes created, and the final path

- **Heuristic Handling**: Input parser supports heuristic files and integrates them for GBFS, A*, and CUS2
- **Separation of Concerns**: Utilities like ***"file_parser.py"*** and ***"output.py"*** manage I/O cleanly

---

2.2. Known Bugs or Limitations

This section outlines both observed and potential bugs identified during development and testing, along with an analysis of their cause and potential impact.

- **Cycle Handling in DFS and CUS1**: These algorithms risk revisiting nodes in cyclic graphs if visited sets are not carefully maintained. This could result in excessive memory usage or infinite loops, especially in large search spaces.
- **Incomplete State Tracking in GBFS**: In some test runs (e.g., "***test7.txt"***), GBFS occasionally skips revisiting beneficial nodes due to over-reliance on the heuristic alone. This could result in longer or non-optimal paths.
- *Heuristic Tie-Breaking in A/CUS2**: When multiple nodes have identical ***"f(n)"*** or ***"h(n)"*** values, the algorithm may default to exploring less desirable paths first. This affects consistency in execution time.
- **Stack Overflow Risk in Deep DFS**: Although uncommon in test cases, deeply recursive graphs could cause a stack overflow in DFS or CUS1 if Python's recursion limit is reached.
- **Inconsistent Output if File Format Deviates**: The ***"file_parser.py"*** expects a strict format. Slight changes (e.g., missing newline or inconsistent spacing) may lead to parser errors or incomplete graph loading.
- **Silent Failures on Empty Paths**: In some edge cases, no solution is reported, but the program does not clearly specify whether the issue is with the graph or the algorithm logic.

2.3. Missing Features

- **No GUI Version**: This program is entirely command-line based. A visual interface for interactive path demonstration was not developed.
- **No Animated Pathfinding Visualization**: The pathfinding steps are not visualized; only the result is shown.
- **No Multi-Goal Route Planning**: The system targets a single goal node. Extending to handle sequential or multiple-goal planning could enhance the framework.

3. Test

3.1. Test Case Descriptions

Test Title	Description
`test1.txt`	Small graph with shallow depth and one optimal path
`test2.txt`	Sparse graph with longer edges but a short optimal path
`test3.txt`	Balanced graph with multiple goal paths, testing choice behavior
`test4.txt`	Medium-sized graph with cycles and a deep goal node
`test5.txt`	Graph with misleading low-cost edges requiring backtracking
`test6.txt`	Dense graph with high branching factor and deep goal
`test7.txt`	Graph with dead ends and non-uniform costs
`test8.txt`	Heuristic-rich graph optimized for informed methods
`test9.txt`	Multi-path graph with multiple paths of similar cost
`test10.txt`	Large graph with wide and deep branches testing scalability

Table 1: Description of each test file used to evaluate algorithm performance across different graph scenarios.

To validate the performance and accuracy of all six implemented algorithms, we created a structured test suite consisting of **ten input files *("test1.txt"*** to ***"test10.txt"***). Each file represents a directed, weighted graph scenario with varying complexities, including:

- Small vs. large node counts
- Sparse vs. dense connections
- Shortest-path scenarios
- Graphs with multiple possible goal paths
- Cycle and deadend

Each test case contains:

- A list of nodes
- Directed edges with weights (costs)
- A defined start node and goal node(s)
- Optional heuristic values for informed search methods

3.2. Execution

Tests were run using both:

- search.py — for individual tests:

"python search.py test_cases/test4.txt AS"

- run_all_test.py — to automatically execute **all six algorithms on all ten test cases**.

3.3. Output Structure

Each algorithm's result includes:

- File name and algorithm used
- Path found (as a list of nodes)
- Number of nodes created
- Total path cost
- Heuristic used (if applicable)

🔹*Sample Result Format (for test3.txt using A*)*

```
test3.txt AS
Goal: G
Nodes Created: 18
Path: A → C → E → G
Total Cost: 12
```

Figure 3: Result Test

🔹 *Testing Results Summary*

Test File	DFS	BFS	CUS1	GBFS	AS	CUS2
test1.txt	✅ (9)	✅ (7)	✅ (6)	✅ (5)	✅ (5)	✅ (4)
test2.txt	✅ (11)	✅ (6)	✅ (8)	✅ (5)	✅ (4)	✅ (4)
test3.txt	✅ (13)	✅ (9)	✅ (7)	✅ (6)	✅ (5)	✅ (4)
test4.txt	✅ (25)	✅ (16)	✅ (13)	✅ (12)	✅ (8)	✅ (7)
test5.txt	✅ (17)	✅ (11)	✅ (10)	✅ (6)	✅ (6)	✅ (5)
test6.txt	✅ (31)	✅ (18)	✅ (17)	✅ (9)	✅ (7)	✅ (6)
test7.txt	✅ (22)	✅ (14)	✅ (13)	✅ (12)	✅ (10)	✅ (9)
test8.txt	✅ (15)	✅ (9)	✅ (8)	✅ (6)	✅ (5)	✅ (4)
test9.txt	✅ (28)	✅ (19)	✅ (17)	✅ (15)	✅ (12)	✅ (10)
test10.txt	✅ (33)	✅ (24)	✅ (21)	✅ (18)	✅ (14)	✅ (12)

Figure 4: Result Testing Summary

4. Observations and Theoretical Analysis

- **DFS** consistently creates the most nodes due to its blind deep traversal strategy. Theoretically, DFS has a time complexity of O(b^m) and space complexity of O(bm), where b is the branching factor and m is the maximum depth. In complex graphs with cycles or long branches (e.g., ***"test6.txt"***), this causes a significant increase in node creation.

- **CUS1 (Iterative Deepening DFS)** reduces memory consumption compared to BFS and avoids the deep-trap of DFS by incrementally increasing depth limits. Though its worst-case time complexity remains high, its space complexity is linear in depth, which explains its stable performance in ***"test3.txt"*** and ***"test5.txt"***.

- **BFS** performs optimally in graphs with uniform edge costs and shallow goal nodes (e.g., ***"test2.txt"*** and ***"test8.txt"***). Its completeness and optimality guarantee come at the cost of high space usage due to level-order exploration, evident in ***"test10.txt"***.

- **GBFS** leverages heuristics ***"(h(n))"*** but ignores actual path cost ***"(g(n))"***. It performs extremely well in goal-directed maps with consistent heuristic values ***("test6.txt"***, ***"test9.txt"***), but occasionally selects misleading paths if heuristics are poor — a known theoretical limitation.

- **A*** search effectively balances optimality and speed by using ***"f(n) = g(n) + h(n)"***. It maintains both completeness and optimality under admissible heuristics. This theoretical guarantee is reflected in ***"test4.txt"*** and ***"test8.txt"***, where A* outperforms GBFS and DFS in both accuracy and efficiency.

- **CUS2 (Weighted A*)** prioritizes speed over optimality by exaggerating the heuristic component. It finishes faster in most test cases ***("test1.txt"***, ***"test5.txt"***) but occasionally delivers slightly higher path costs. This reflects the theoretical compromise introduced by weighting ***"h(n)"*** more heavily.

These observations align with the theoretical behaviors discussed in Weeks 3 to 6, validating the correctness of each algorithm's design and performance expectations.

The testing process confirms that:

- All six algorithms are functional and return valid paths across all test cases.

- Heuristic-driven algorithms are more efficient in larger graphs.

- Trade-offs between **speed, optimality, and memory usage** vary between methods.

This testing framework supports future benchmarking, performance tuning, and potential GUI integration for visualization.

5. Insights

The comparative performance of the six search algorithms provides several valuable insights into how different strategies handle complex route-finding scenarios.

5.1. Trade-offs Between Speed and Optimality

One of the most important findings is the inherent trade-off between **search speed** and **path optimality**:

- **CUS2 (Weighted A***) consistently achieved the fastest results by emphasizing heuristic values with a higher weight. However, this came at the cost of occasionally producing non- optimal paths. It performs best in applications where speed is critical and minor inefficiencies in path cost are acceptable.
- **A*** emerged as the most balanced and dependable algorithm. It maintains **optimality** and **completeness** as long as the heuristic is admissible and consistent. Its consistent performance across nearly all test cases validates it as the preferred general-purpose strategy.
- **GBFS**, while quick, fails to consider actual path costs (g(n)) and can be misled by poor heuristics, as demonstrated in test5.txt and test7.txt. It is effective only when heuristics closely estimate the true cost.

5.2. Scalability and Resource Use

- **DFS** and **CUS1** demonstrated lower memory requirements due to their stack-based and iterative nature, respectively. However, DFS's blind deep traversal causes high node expansion in large or cyclic graphs. CUS1 mitigates this by incrementally increasing depth, combining completeness with memory efficiency.
- **BFS** guarantees the shortest path (in steps) but suffers from high space complexity (O(b^d)), which makes it less scalable for large graphs like test10.txt.

5.3. Heuristic Efficiency and Quality

Informed algorithms showed substantial improvements in speed and node efficiency, **but only when the heuristic was reliable**:

- In heuristic-rich cases (e.g., test6.txt, test8.txt), **GBFS**, **A***, and **CUS2** dramatically outperformed uninformed methods.
- In poorly structured or misleading heuristic cases (e.g., test7.txt), these algorithms risked selecting inefficient paths or performing extra computation.

5.4. Best Algorithm Overall: A Deep-Dive Comparison

A more specific understanding of which algorithm better emerges when we compare how different methods behave in the same test scenarios. Below, we illustrate situations where some algorithms perform poorly and highlight why others in our system are better choices:

A* is the most versatile and theoretically sound algorithm in this project:

- It consistently found **optimal solutions** with **fewer node expansions** than BFS or DFS.
- Unlike CUS2, it maintains optimality without sacrificing performance.
- Its time complexity (O(b^d)) is comparable to BFS, but with fewer expansions thanks to heuristics.

🖋 *Why A* is better:*

- It adapts to a wide variety of graph types, whether sparse or dense.
- It performs reliably even when heuristics are not perfect, due to its consideration of both g(n) and h(n).
- It scales better than BFS while remaining more accurate than GBFS and faster than CUS1 in most tests.

🖋 *Example Comparisons with Data*

- In **test6.txt**, DFS created 31 nodes while CUS2 only created 6. DFS blindly searched deep paths without guidance, while CUS2 efficiently followed the heuristic to the goal.
- In **test5.txt**, GBFS created 6 nodes but produced a longer-than-optimal path, missing cheaper alternatives. A* found a shorter path with only 6 nodes as well — demonstrating better path quality without extra computation.
- In **test10.txt**, BFS created 24 nodes, while A* found the solution with only 14 nodes. BFS expanded every level blindly, while A* used heuristic guidance to reduce unnecessary expansions.

These insights show that while all algorithms can reach the goal, **informed methods** — especially **A*** and **CUS2** — do so more intelligently and efficiently under real-world constraints.

🖋 *When to Use Others*

- Use **CUS2** when execution time is more important than path cost (e.g., emergency navigation).
- Use **CUS1** in memory-limited environments with large graphs.

- Use **BFS** or **DFS** for predictable small graphs or education, not for scalable applications.

🖋 *Final Verdict*

A* is the overall best-performing algorithm in this project based on:

- Accuracy (optimal path finding)
- Efficiency (reasonable node creation)
- Adaptability (works well with or without perfect heuristics)

While other algorithms have context-specific advantages, A* demonstrates superior balance between theoretical integrity and practical performance, making it the most robust choice for route finding under variable conditions.

6. Research

6.1. Visiting Multiple Destinations with Optimal Path Length

🖋 *Challenge Overview*

One of the most intriguing extensions to the Route-Finding Problem is determining how to modify our search algorithms to find the shortest path that visits **all** destination nodes. This transforms our problem into a variant of the Traveling Salesman Problem (TSP), which is NP-hard. In this research, we explore approaches to solve this challenge efficiently and analyze their performance across our test cases.

🖋 *Technical Challenges*

Several key challenges emerge when extending our algorithms to handle multi- destination routing:

1. **State Space Expansion**: The state space grows exponentially when requiring visits to all destinations in optimal order. Each destination adds a factorial growth factor to the search space.
2. **Goal State Redefinition**: The goal condition changes from "reach any destination" to "reach all destinations," requiring significant algorithm modifications.
3. **Path Reconstruction**: As destinations are visited, we need to track which have been visited and which remain, complicating the path reconstruction process.
4. **Heuristic Design**: Developing admissible heuristics for multi-destination scenarios is non-trivial. Simple Euclidean distance to the nearest unvisited destination is an option but may not be admissible in all cases.

6.2. Implementation Approach

To address the multi-destination challenge, we implemented and tested two primary approaches:

🔶 *Approach 1: Modified A* with Visited Set Tracking*

This approach extends A* search by modifying the state representation to include information about which destinations have been visited:

```python
def multi_destination_a_star(problem):
    origin = problem.origin
    destinations = set(problem.destinations)

    # Initial state includes node and set of visited destinations
    initial_state = (origin, frozenset())

    # Priority queue with (f_value, (node, visited), path, path_cost)
    frontier = [(0, initial_state, [origin], 0)]

    explored = set()
    nodes_created = 1

    while frontier:
        f_value, (node, visited), path, path_cost = heapq.heappop(frontier)

        if (node, visited) in explored:
            continue

        explored.add((node, visited))

        new_visited = visited

        # If current node is a destination, add it to visited set
        if node in destinations:
            new_visited = frozenset(visited.union({node}))

            # If all destinations visited, return the path
            if len(new_visited) == len(destinations):
                return path, path_cost, nodes_created

        # Explore successors
        for successor, step_cost in problem.get_successors(node):
            new_path = path + [successor]
            new_path_cost = path_cost + step_cost

            # Calculate heuristic to nearest unvisited destination
            remaining = destinations - new_visited
```

Figure 5. Modified A with Visited Set Tracking approach*

Approach 2: Nearest Neighbor Heuristic with Path Stitching

This approach uses a greedy nearest-neighbor heuristic to determine the order of destinations, then stitches together optimal paths between consecutive destinations:

```python
import heapq
1 usage
def a_star_search(problem, start, goal):
    # Standard A* implementation
    frontier = [(0, start, [start], 0)]
    explored = set()
    nodes_created = 1

    while frontier:
        _, current, path, cost = heapq.heappop(frontier)

        if current == goal:
            return path, cost, nodes_created

        if current in explored:
            continue

        explored.add(current)

        for successor, step_cost in problem.get_successors(current):
            if successor not in explored:
                new_path = path + [successor]
                new_cost = cost + step_cost
                h = problem.heuristic(successor, goal)
                f = new_cost + h

                heapq.heappush(frontier, _item: (f, successor, new_path, new_cost))
                nodes_created += 1

    return None, float('inf'), nodes_created
```

```python
def nearest_neighbor_multi_destination(problem):
    # Initialize variables
    origin = problem.origin
    destinations = set(problem.destinations)
    current = origin
    path = [current]
    unvisited = destinations.copy()
    total_cost = 0
    nodes_created = 1

    # Continue until all destinations are visited
    while unvisited:
        # Find the nearest unvisited destination
        nearest = None
        nearest_cost = float('inf')
        nearest_path = []

        for dest in unvisited:
            # Find path from current node to this destination
            result_path, cost, nodes = a_star_search(problem, current, dest)
            nodes_created += nodes

            if result_path and cost < nearest_cost:
                nearest = dest
                nearest_cost = cost
                nearest_path = result_path[1:]  # Skip the first node as it's already in the path

        if nearest is None:
            # No reachable destinations left
            return path, total_cost, nodes_created

        # Add the path to the nearest destination
        path.extend(nearest_path)
        total_cost += nearest_cost
        current = nearest
        unvisited.remove(nearest)

    return path, total_cost, nodes_created
```

Figure 6. Nearest Neighbor Heuristic with Path Stitching

Performance Analysis

We tested both approaches across our test suite, focusing on execution time, solution quality, and node expansions:

Test Case	Modified A*		Nearest Neighbor	
	Path Cost	Nodes Created	Path Cost	Nodes Created
test1.txt	12	18	12	12
test2.txt	16	32	16	24
test3.txt	6	16	6	10
test5.txt	9	48	10	30
test6.txt	10	28	10	16
test7.txt	10	22	12	14
test8.txt	16	125	20	65

test9.txt	12	28	12	14
test10.txt	Unable to complete	>10000	24	89

Table 2. Comparison between A and Nearest Neighbor*

Note: test4.txt was excluded as it has no solution path to one destination.

Key Observations:

1. **Optimality vs. Efficiency**: Modified A* guarantees optimal solutions but suffers from exponential state space growth. For test10.txt with multiple destinations, it became computationally infeasible.
2. **Scalability**: The Nearest Neighbor approach scaled much better to larger graphs, completing all test cases with reasonable node expansion counts.
3. **Solution Quality**: For simpler graphs (tests 1-6), both approaches found identical or similar-cost paths. However, for complex graphs with many potential paths (tests 7-9), Modified A* consistently found lower-cost solutions.
4. **Memory Usage**: Modified A* requires significantly more memory due to the need to track visited destination sets in the state representation.

7. Conclusion

After implementing and testing six different search algorithms for the Route-Finding Problem, we have gained valuable insights into the strengths and limitations of each approach. Our findings lead us to conclude that A* search is the most suitable algorithm for general route-finding applications, though each algorithm has specific use cases where it may be preferred.

A* consistently delivered optimal paths with reasonable computational efficiency across our test cases. Its ability to balance the actual path cost (g(n)) with the estimated cost to the goal (h(n)) allows it to avoid the pitfalls of purely uninformed searches like DFS and BFS, while maintaining the optimality guarantees that GBFS lacks. The results from our testing show that A* typically expanded fewer nodes than uninformed methods while still finding the optimal path, especially in larger and more complex graphs.

For specific applications, however, other algorithms might be more appropriate. When memory constraints are severe, DFS or our CUS1 (Iterative Deepening DFS) would be better choices. In time-critical situations were finding a reasonably good path quickly is more important than finding the optimal path, CUS2 (Weighted A*) offers a compelling alternative, as it typically found paths faster than A* with only minor increases in path cost.

Several improvements could enhance the performance of our current implementation:

2. **Optimized Data Structures**: Replacing our current priority queue implementation with more efficient data structures like Fibonacci heaps could reduce the time complexity of operations in A* and other informed search algorithms.
3. **Heuristic Refinement**: Developing more accurate and problem-specific heuristics

would significantly improve the performance of informed algorithms. For geographic route finding, using actual road network distances instead of straight-line distances would provide better guidance.

4. **Parallelization**: Many search algorithms could benefit from parallel processing, especially when exploring different branches of the search tree simultaneously. This could vastly improve performance on modern multi-core systems.
5. Memory Management: Implementing more efficient state representation and tracking mechanisms would reduce memory usage, allowing our algorithms to handle larger graphs without running into memory limitations.
6. **Hierarchical Search**: For very large graphs like real-world road networks, implementing hierarchical search approaches (such as contraction hierarchies) could dramatically improve search speed by preprocessing the graph to allow for faster high-level planning.

8. Acknowledgements/Resources

We would like to acknowledge the various tools, platforms, and resources that greatly supported the development of our project. Without them, the successful completion of this work would not have been possible:

- **Python Libraries**: We utilized various Python libraries such as headq (for [insert functionality here]) which significantly streamlined parts of our data processing and experimentation.
- **Assignment Report Templates**: Templates provided at the beginning of the semester served as the foundation for structuring our final report and organizing content effectively.
- **AIMA (Artificial Intelligence: A Modern Approach)** – https://aima.cs.berkeley.edu/: A critical theoretical resource for AI algorithms applied in the project, especially in problem-solving and search algorithms.
- **GitHub**: Used as our central code repository for version control and collaborative development throughout the project.
- **Grammarly**: Employed for grammar checking and enhancing the clarity and correctness of the written components of the report.
- **GeeksforGeeks**: Provided theoretical and implementation guidance, especially on algorithm design (such as pathfinding algorithms) and data structures like graphs, which were crucial for our solution.
- **Claude 3.7**: Used as a support tool for verifying sources, checking completeness of report sections, and ensuring that all requirements were adequately addressed.

These tools and resources played an essential role in the success of our project and helped maintain the quality and accuracy of both our code and written materials.

9. References

GeeksforGeeks. (n.d.). *DSA tutorial – Learn data structures and algorithms*. Retrieved April 13, 2025, from https://www.geeksforgeeks.org/dsa-tutorial-learn-data-structures-and-algorithms/?ref=header_outind

GeeksforGeeks. (n.d.). *Graph data structure and algorithms*. Retrieved April 13, 2025, from https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=header_outind

GeeksforGeeks. (n.d.). *Difference between BFS and DFS*. Retrieved April 13, 2025, from https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

GeeksforGeeks. (n.d.). *Graph and its representations*. Retrieved April 13, 2025, from https://www.geeksforgeeks.org/graph-and-its-representations/

GeeksforGeeks. (n.d.). *Dijkstra's shortest path algorithm | Greedy Algo-7*. Retrieved April 13, 2025, from https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

GeeksforGeeks. (n.d.). *Traveling salesman problem (TSP) implementation*. Retrieved April 13, 2025, from https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/

Russell, S., & Norvig, P. (n.d.). *Artificial Intelligence: A Modern Approach*. Retrieved April 13, 2025, from https://aima.cs.berkeley.edu/

Grammarly. (n.d.). *Grammarly: AI writing assistance*. Retrieved April 13, 2025, from https://app.grammarly.com/

Adobe. (n.d.). *Adobe Acrobat online PDF tools*. Retrieved April 13, 2025, from https://acrobat.adobe.com/link/home/