NATIONAL ECONOMICS UNIVERSITY, HANOI
COLLEGE OF TECHNOLOGY
FACULTY OF DATA SCIENCE AND ARTIFICIAL INTELLIGENCE

INTRODUCTION TO AI (TOKT11121.AI66A)

REPORT

The Optimal Route Planning Problem for Waste Collection: A Comparative Analysis of Search Algorithms Applied in Reality

Instructor: PhD. Nguyen Trong Nghia

Student:

Pham Huu Gia An	11247254
Nguyen Ngan An	11247253
Nguyen Trong Dai	11247268
Le Ba Phong	11247339

Hanoi, October 2025

Contents

1 Introduction

In the domain of urban management, the efficient collection of municipal solid waste (MSW) is a critical logistical operation essential for maintaining public health, environmental sustainability, and fiscal responsibility. Suboptimal routing strategies for waste collection vehicles lead to excessive fuel consumption, increased operational costs, higher carbon emissions, and undue strain on municipal resources. This complex logistical challenge can be modeled as the Traveling Salesman Problem (TSP) or the Hamiltonian Path Problem (HPP), and more broadly as a Vehicle Routing Problem (VRP), which is a well-known NP-hard problem in combinatorial optimization.

The VRP, in its essence, can be viewed as an extension of the TSP. In the context of waste collection, the depot represents the starting and ending point, the collection bins are the nodes (or "cities") that must be visited, and the road network forms the weighted edges of the graph. The objective is to determine a sequence of collection points that forms the shortest possible tour, beginning and ending at the depot while visiting each required location exactly once to save fuel, reduce operational costs, and increase collection efficiency.

The relevance of solving this optimization problem extends far beyond theoretical computer science and waste management. The core principles of route optimization are fundamental to a wide array of modern logistical challenges, including package delivery for e-commerce, tourist itinerary planning, and the navigation of autonomous robots in warehouses and manufacturing facilities. As urban systems and supply chains grow in complexity, the need for intelligent, automated routing systems becomes paramount. Therefore, understanding the computational trade-offs of different search strategies provides crucial insights for developing robust platforms for a multitude of real-world applications.

In this study, we apply and evaluate three foundational search algorithms for the waste collection routing problem: Breadth-First Search (BFS) as a baseline uninformed search, Greedy Best-First Search (GBFS) with a straight-line distance heuristic, and the A* search algorithm using the sum of the path cost and the Manhattan distance as its heuristic function. We will compare their performance based on the total length of the generated route (solution quality), the number of nodes expanded (search space efficiency), and overall computational time, thereby assessing the applicability of these AI search methodologies for a tangible, real-world optimization task.

2 Problem Formulation

2.1 State Space Representation

The optimal route planning problem for waste collection, modeled as a Traveling Salesman Problem (TSP), can be formally defined as a search problem with the following components:

State Space (S): Each state is represented by a tuple `(current_city, visited_cities)`, where `current_city` is the node the collection vehicle is currently at, and `visited_cities` is the set of all nodes (waste bin locations) that have already been visited. The environment is a weighted graph where nodes represent locations and edges represent roads with associated travel distances.

Initial State (s_0): The starting configuration, `(Depot, {Depot})`, where the vehicle begins its route at the main depot.

Goal State (s_G): A state where all required waste bins have been visited and the vehicle has returned to the depot. This is represented as `(Depot, {All_Bins` $\cup$ `Depot})`.

Actions (A): The set of possible actions from any state is to move from the `current_city` to an adjacent, unvisited city `next_city` along a valid edge in the graph.

Transition Model: Given a state $s = (u, V)$ and an action of moving to an adjacent city v, the resulting state is $s' = (v, V \cup \{v\})$.

Path Cost: The cost of a path is the sum of the distances (weights) of all edges traversed. The objective is to find a path from the initial state to the goal state with the minimum total path cost.

2.2 Heuristic Functions

Two heuristic functions are employed in this study for the informed search algorithms:

Dijkstra-based Nearest Neighbor Heuristic (h_1): Used for Greedy Best-First Search. This heuristic estimates the cost to the goal by finding the shortest path from the current node to the *nearest* unvisited waste bin. The algorithm repeatedly calculates the shortest path to the next closest goal, making it a greedy approach. It does not consider the overall path structure.

Minimum Spanning Tree Heuristic (h_2): Used for A* Search. This is a more complex and admissible heuristic that estimates the cost to complete the tour. It is calculated as:

$$h(n) = \text{Cost}(\text{MST}_{\text{remaining}}) + \text{Cost}(\min_{\text{edge\_out}}) + \text{Cost}(\min_{\text{edge\_in}})$$

Where:

- $\text{Cost}(\text{MST}_{\text{remaining}})$ is the cost of the Minimum Spanning Tree connecting all unvisited waste bins.
- $\text{Cost}(\min_{\text{edge\_out}})$ is the cost of the shortest edge from the current node n to one of the unvisited bins.
- $\text{Cost}(\min_{\text{edge\_in}})$ is the cost of the shortest edge from any unvisited bin back to the starting depot.

This heuristic is admissible because the cost of an optimal tour is always greater than or equal to the cost of any spanning tree on the same set of nodes.

3 Algorithms

This section describes the three search algorithms applied to solve the waste collection routing problem.

3.1 Uniform Cost Search (UCS)

UCS is an uninformed search algorithm that explores the state space by expanding the node with the lowest path cost ($g(n)$) from the start node. It systematically explores paths in increasing order of cost.

Characteristics: UCS does not use a heuristic function. It is equivalent to Dijkstra's algorithm and guarantees finding the optimal (least-cost) route if all edge costs are non-negative. However, its efficiency degrades significantly in large state spaces as it explores in all directions without a goal-oriented guide.

Complexity: The time complexity for a TSP-like problem is factorial, approximately $O(n! \log(n!))$. Its space complexity is proportional to the number of expanded nodes, which can become very high for larger problems.

3.2 Greedy Best-First Search (GBFS)

GBFS is an informed search algorithm that expands the node that appears to be closest to the goal, using only the heuristic function $h(n)$ to make its decision. In this study, it uses the Dijkstra-based nearest neighbor heuristic (h_1).

Characteristics: By prioritizing the heuristic estimate, GBFS is often very fast and requires less memory. It is suitable for problems where a quick, reasonably good solution is sufficient. However, because it ignores the path cost already incurred ($g(n)$), it does not guarantee an optimal solution.

Complexity: The time complexity depends heavily on the quality of the heuristic but is generally better than uninformed search, roughly $O(n^2 \log n)$ in this implementation. The space complexity is low.

3.3 A* Search

A* is an informed search algorithm that combines the advantages of UCS and GBFS. It aims to find the shortest path by using an evaluation function $f(n) = g(n) + h(n)$.

Characteristics: A* balances the cost to reach the current node ($g(n)$) with an estimated cost to the goal ($h(n)$). When used with an admissible heuristic, such as the Minimum Spanning Tree heuristic (h_2), A* is both complete and optimal. It is significantly more efficient than UCS because the heuristic guides the search toward the goal.

Complexity: The time complexity is highly dependent on the accuracy of the heuristic but can be factorial in the worst case ($O(n! \log(n!))$). The space complexity is also a concern as it stores all generated nodes in memory.

4 Experiment, Results and Evaluation

4.1 Experiment Setup

The experiments were conducted on a simulated city map represented as a weighted graph. The graph structure, defined in Python, consists of 18 nodes and various types of connections to mimic real-world conditions.

Listing 1: Graph structure used for simulation

```
graph = {
    "Le␣Duan": {"Tran␣Hung␣Dao": {"distance": 15}},
    "Ly␣Thuong␣Kiet": {
        "Hai␣Ba␣Trung": {"distance": 20},
        "Nguyen␣Du": {"distance": 25},
        "Tran␣Hung␣Dao": {"distance": 30}
    },
    "Hai␣Ba␣Trung": {
        "Ly␣Thuong␣Kiet": {"distance": 20},
        "Ba␣Trieu": {"distance": 18},
        "Hang␣Bai": {"distance": 22}
    },
    "Phan␣Dinh␣Phung": {},
    "Tran␣Hung␣Dao": {
        "Le␣Duan": {"distance": 15},
        "Nguyen␣Du": {"distance": 12},
        "Kim␣Ma": {"distance": 40},
        "Ly␣Thuong␣Kiet": {"distance": 30}
    },
    "Nguyen␣Du": {
        "Ly␣Thuong␣Kiet": {"distance": 25},
        "Tran␣Hung␣Dao": {"distance": 12},
        "Ba␣Trieu": {"distance": 20},
        "Giang␣Vo": {"distance": 28},
        "Lang␣Ha": {"distance": 26}
    },
    "Ba␣Trieu": {
        "Hai␣Ba␣Trung": {"distance": 18},
        "Nguyen␣Du": {"distance": 20},
        "Hang␣Bai": {"distance": 14},
        "Lang␣Ha": {"distance": 25},
        "Giang␣Vo": {"distance": 27}
    },
    "Hang␣Bai": {
        "Phan␣Dinh␣Phung": {"distance": 16},
        "Ba␣Trieu": {"distance": 14},
        "Hoang␣Hoa␣Tham": {"distance": 30},
        "Hai␣Ba␣Trung": {"distance": 22}
    },
    "Kim␣Ma": {
        "Giang␣Vo": {"distance": 20},
        "Lang␣Ha": {"distance": 25},
        "Ngoc␣Khanh": {"distance": 28}
    },
    "Giang␣Vo": {
        "Nguyen␣Du": {"distance": 28},
        "Kim␣Ma": {"distance": 20},
        "Lang␣Ha": {"distance": 12},
        "Ngoc␣Khanh": {"distance": 21},
        "Ba␣Trieu": {"distance": 27}
    },
```

```
    "Lang␣Ha": {
        "Ba␣Trieu": {"distance": 25},
        "Giang␣Vo": {"distance": 12},
        "Hoang␣Hoa␣Tham": {"distance": 18},
        "Lac␣Long␣Quan": {"distance": 26},
        "Nguyen␣Du": {"distance": 26}
    },
    "Hoang␣Hoa␣Tham": {
        "Hang␣Bai": {"distance": 30},
        "Lang␣Ha": {"distance": 18},
        "Tran␣Duy␣Hung": {"distance": 32},
        "Lac␣Long␣Quan": {"distance": 24}
    },
    "Lang": {"Kim␣Ma": {"distance": 25}},
    "Ngoc␣Khanh": {
        "Giang␣Vo": {"distance": 21},
        "Lac␣Long␣Quan": {"distance": 22},
        "Kim␣Ma": {"distance": 28}
    },
    "Lac␣Long␣Quan": {
        "Ngoc␣Khanh": {"distance": 22},
        "Lang␣Ha": {"distance": 26},
        "Hoang␣Hoa␣Tham": {"distance": 24}
    },
    "Tran␣Duy␣Hung": {"Hoang␣Hoa␣Tham": {"distance": 32}}
}
```

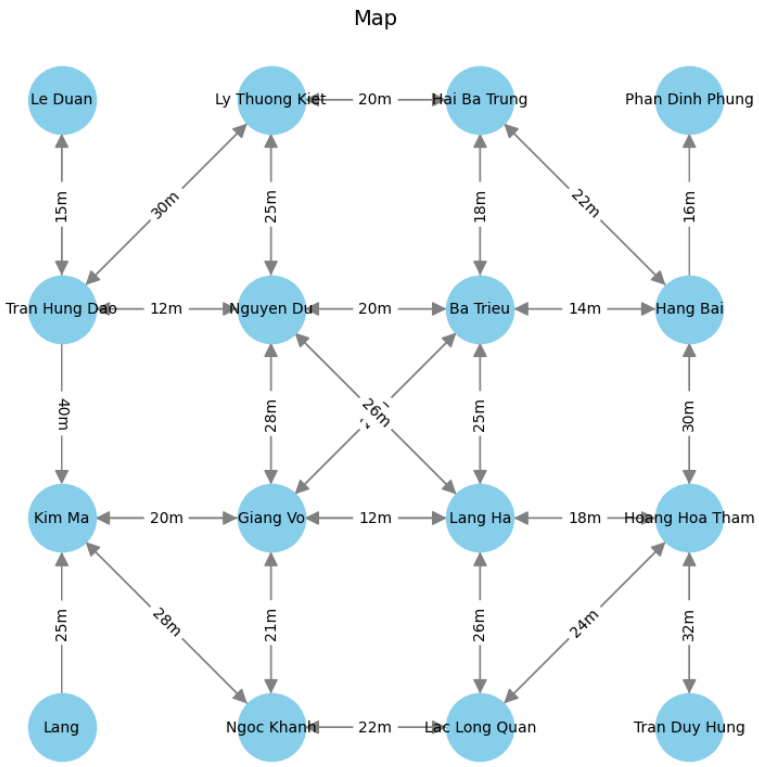

Figure 1: Simulated Map

The task was formulated as a Traveling Salesman Problem (TSP), where the starting and ending point was node `"Le Duan"`. The number of intermediate waste bins to visit (n) increased incrementally from 2 to 12. The performance of UCS, GBFS, and A* was compared based on the path cost, execution time, and number of expanded nodes.

The following code snippets represent the core logic of each algorithm tested.

Listing 2: Uniform Cost Search Implementation

```
def uniform_cost_search(graph, start, bins, TSP=False):
    bins = set(bins)
    if TSP:
        bins = bins | {start}

    pq = [(0, start, frozenset(bins - {start}), [start])]  # (cost, node,
    ↪ remaining_bins, path)
    visited = {}
    expanded_nodes = 0

    while pq:
        g, curr, remaining, path = heapq.heappop(pq)
        expanded_nodes += 1

        if not remaining:
            if TSP:
                if curr == start:
```

```
                    return path, g, expanded_nodes
                else:
                    if start in graph[curr]:
                        total_cost = g + graph[curr][start]["distance"]
                        return path + [start], total_cost, expanded_nodes
            else:
                return path, g, expanded_nodes

        state_key = (curr, remaining)
        if state_key in visited and visited[state_key] <= g:
            continue
        visited[state_key] = g

        for neighbor, info in graph[curr].items():
            cost = info["distance"]
            new_g = g + cost
            new_remaining = set(remaining)
            if neighbor in new_remaining:
                new_remaining.remove(neighbor)

            heapq.heappush(pq, (new_g, neighbor, frozenset(new_remaining), path +
                ↪ [neighbor]))

    return None, float("inf"), expanded_nodes
```

Listing 3: Greedy Best-First Search Implementation

```
def greedy_collect_bins(graph, start, bins, TSP=False):
    current = start
    full_path = [current]
    total_cost = 0
    remaining = set(bins)
    expanded_total = 0

    while remaining:
        dist, prev, expanded = dijkstra_greedy(graph, current)
        expanded_total += expanded

        next_bin = min(remaining, key=lambda b: dist[b])
        cost_to_next = dist[next_bin]
        segment = reconstruct_path(prev, current, next_bin)

        full_path.extend(segment[1:])
        total_cost += cost_to_next
        current = next_bin
        remaining.remove(next_bin)

    if TSP:
        dist, prev, expanded = dijkstra_greedy(graph, current)
        expanded_total += expanded
        cost_back = dist[start]
        segment = reconstruct_path(prev, current, start)
        full_path.extend(segment[1:])
        total_cost += cost_back

    return full_path, total_cost, expanded_total
```

Listing 4: A* Search Implementation

```
def a_star_collect(graph, start, bins, TSP = False):
```

```
    bins = set(bins)
    if TSP:
        bins = bins | {start}

    initial_h = heuristic(graph, start, bins - {start}, start, TSP)
    pq = [(initial_h, 0, start, frozenset(bins - {start}), [start])]
    visited = {}
    expanded_nodes = 0

    while pq:
        f, g, curr, remaining, path = heapq.heappop(pq)
        expanded_nodes += 1

        if not remaining:
            if TSP:
                if curr == start:
                    return path, g, expanded_nodes
            else:
                return path, g, expanded_nodes

        state_key = (curr, remaining)
        if state_key in visited and visited[state_key] <= g:
            continue
        visited[state_key] = g

        for neighbor, info in graph[curr].items():
            cost = info["distance"]
            new_g = g + cost
            new_remaining = set(remaining)
            if neighbor in new_remaining:
                new_remaining.remove(neighbor)

            h = heuristic(graph, neighbor, new_remaining, start, TSP)
            new_f = new_g + h
            heapq.heappush(
                pq,
                (new_f, new_g, neighbor, frozenset(new_remaining), path +
                    ↪ [neighbor])
            )

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

4.2 Results

The quantitative results from running the three algorithms with increasing numbers of target nodes are summarized in Table 1.
The data from Table 1 illustrates exponential growth in computation for both uninformed and informed optimal searches.

[H]

Table 1: Comparison of Algorithm Performance

n	Algorithm	Cost	Time (s)	Expanded Nodes
2	UCS	152	0.000120	116
	Greedy	152	0.000050	48
	A*	152	0.001218	12
4	UCS	205	0.000679	510
	Greedy	205	0.000081	80
	A*	205	0.005665	41
6	UCS	217	0.001317	960
	Greedy	217	0.000088	112
	A*	217	0.005960	43
8	UCS	232	0.002711	2421
	Greedy	242	0.000120	144
	A*	232	0.007891	35
10	UCS	323	0.011796	9982
	Greedy	350	0.000142	176
	A*	323	0.040141	134
12	UCS	400	0.054237	32140
	Greedy	472	0.000167	208
	A*	400	0.271525	924

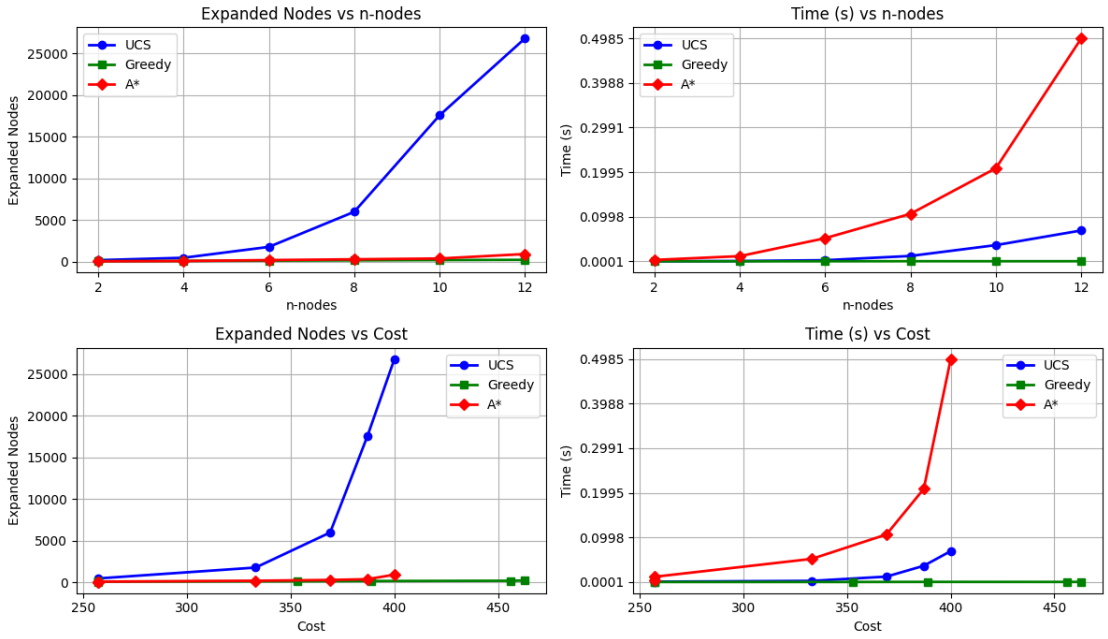

Figure 2: Comparison Between 3 Algorithms

4.3 Evaluation

Based on the experimental data, the performance of the algorithms varies significantly with the scale of the problem.
Greedy Best-First Search (GBFS) consistently delivered the fastest results with the lowest number of expanded nodes. This is expected, as its heuristic-only approach prunes the search space aggressively. However, its trade-off is optimality — for problems with $n \geq 8$, it failed to find the least-cost path.

Uniform Cost Search (UCS) guaranteed an optimal solution but at a significant computational price. The number of expanded nodes and the execution time grew exponentially with n, making it impractical for larger-scale problems. Its performance is acceptable only for very small instances.

A* Search also provided optimal solutions in all cases. While it was slower for small values of n (due to the overhead of computing the heuristic), its superiority became evident as the problem scaled. For $n = 12$, A* expanded approximately 35 times fewer nodes than UCS, demonstrating its effectiveness in managing combinatorial explosion and providing a better balance between optimality and performance.

In conclusion, while GBFS is suitable for scenarios where speed is critical and a near-optimal solution is acceptable, A* remains the most robust choice for finding guaranteed optimal routes in problems of moderate to large complexity.

5 Real-world Application Insight

The choice of algorithm for a real-world routing problem depends on the specific constraints and objectives of the application, such as the scale of the operation, the need for optimality, and the tolerance for computational delay.

5.1 Uniform Cost Search Applications

Given its guarantee of optimality but high computational cost, UCS is best suited for small-scale, static environments where the optimal route can be pre-calculated and followed repeatedly. The overhead is justified because the path is guaranteed to be the most efficient. Practical examples include route planning for autonomous robots in a small factory or warehouse, or a fixed daily garbage collection route in a small, well-defined residential block.

5.2 Greedy Best-First Search Applications

The primary advantage of GBFS is its speed, making it ideal for large-scale applications where real-time responsiveness is more critical than guaranteed optimality. A "good enough" solution delivered quickly is often preferable to a perfect solution that takes too long to compute. This makes it suitable for consumer-facing map applications that provide driving directions, route planning for city-wide delivery drones where conditions may change dynamically, or programming robotic vacuum cleaners in large office buildings.

5.3 A* Search Applications

A* offers a powerful balance between optimality and performance, making it the most versatile algorithm for a wide range of logistics and planning problems. It is the preferred method for medium to large-scale operations where efficiency directly translates to significant cost savings and thus, optimality is a requirement. Key applications include route optimization for inner-city delivery fleets, scheduling for transport robots in large automated fulfillment centers, and planning paths for drones delivering critical medical supplies, where both speed and efficiency are paramount.

6 Other Heuristic Functions

Beyond the traditional Greedy and A* methods, several advanced heuristics have been developed to improve the efficiency and quality of solutions for the Traveling Salesman Problem (TSP). Among them, two of the most powerful and widely used are the **1-Tree (Held–Karp Bound)** and the **Assignment (Hungarian Lower Bound)**. Both provide very tight lower bounds that significantly enhance the performance of A* and other search algorithms.

6.1 1-Tree Heuristic (Held–Karp Bound)

The 1-Tree heuristic is one of the most classical and effective lower bounds for TSP. It constructs a **Minimum Spanning Tree (MST)** on $n-1$ nodes, excluding one selected node (often the starting city). Then, it connects this excluded node back to the tree using its two cheapest edges. The resulting structure, known as a *1-Tree*, provides a lower bound for the optimal TSP tour cost because it relaxes the requirement that each node must have exactly two incident edges.

Advantages:

- Produces a very tight and admissible lower bound for the TSP.
- Efficient to compute using MST algorithms such as Prim or Kruskal.
- Performs extremely well in practice and is widely used in branch-and-bound solvers.

Limitations:

- Requires recomputation or adjustment when applied dynamically during A* search.
- Bound quality may slightly depend on which node is excluded.

Complexity: $O(n^2 \log n)$

Comparison with A*: When used within A*, the 1-Tree heuristic allows the algorithm to prune a large portion of the search space by providing a close approximation of the optimal path cost. Although it introduces additional computational overhead compared to simpler heuristics (like MST), the trade-off is beneficial for medium-sized instances where search reduction greatly outweighs per-state computation.

6.2 Assignment Heuristic (Hungarian Lower Bound)

The Assignment heuristic models the remaining unvisited nodes as a **cost assignment problem**, where each city must be "assigned" to another with minimal transition cost. By applying the **Hungarian Algorithm**, the heuristic finds the minimum total assignment cost, which serves as a strong lower bound on the cost to complete the tour.

Advantages:

- Provides one of the tightest admissible lower bounds for TSP.
- The heuristic is theoretically grounded in combinatorial optimization, ensuring consistency.
- Highly effective for small to medium-sized problems and near-optimal in practice.

Limitations:

- Computationally expensive with time complexity $O(n^3)$.
- Difficult to recompute for every node during large-scale A* search.

Complexity: $O(n^3)$

Comparison with A*: Incorporating the Hungarian Lower Bound into A* greatly improves pruning capability. Because the bound is very tight—often close to the true optimal cost—A* expands far fewer nodes than when using MST or simple distance-sum heuristics. However, the high computation cost per evaluation limits its practicality for large graphs, making it more suitable for smaller or highly constrained TSP instances.

6.3 Comparison

Both 1-Tree and Assignment heuristics offer significantly better performance than classical admissible heuristics such as MST or Minimum Matching. The 1-Tree heuristic achieves an excellent balance between accuracy and computational cost, making it ideal for dynamic A* implementations. Meanwhile, the Assignment heuristic provides the tightest theoretical bound, achieving near-optimal performance at the expense of higher time complexity.
In short:

- **1-Tree (Held–Karp Bound):** Best trade-off between efficiency and tightness.
- **Assignment (Hungarian Lower Bound):** Most accurate, but computationally heavier.

When combined with A*, both heuristics substantially reduce search space and improve convergence toward the optimal TSP tour.

7 References

1. Yumeng Yan: Research on the A Star Algorithm for Finding Shortest Path (2203)
2. Teemu Nuortioa, Jari Kytojokib, Harri Niskaa, Olli Braysy: Improved route planning and scheduling of waste collection and transport (2005)