Traveling Salesman Problem

COMPSCI 271A - Project Team 12

Shobhit Mendiratta
Department of EECS
University of California, Irvine
ID: 19046223
Email: smendir1@uci.edu

Sangeetha Chandramouli
Department of EECS
University of California, Irvine
ID: 46595620
Email: sangeec1@uci.edu

Maganth Seetharaman
Department of EECS
University of California, Irvine
ID: 64409266
Email: seetharm@uci.edu

Contents

***Abstract*—In our course project, we embark on an investigation of two prominent algorithms to address the Traveling Salesman Problem (TSP): the A* algorithm, with heuristics such as the minimum spanning tree and the reduced cost matrix, and Adaptive Simulated Annealing (ASA). Our choice of Python as the development language reflects its widespread use and support for complex algorithmic implementations. Through this project, we aim to not only deepen our understanding of algorithmic solutions to TSP but also to demonstrate the practical application of these algorithms.**

1. Introduction

The Traveling Salesman Problem (TSP) is one of the most studied problems in operational research and theoretical computer science due to its straightforward premise and complex, NP-hard nature. The problem can be succinctly described as follows: given a list of cities and the distances between each pair of them, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city. Despite its seemingly simple formulation, the TSP encapsulates a wide array of optimization and decision-making challenges that are fundamental to various scientific and practical domains.

In this project, we aim to implement two established algorithms to solve the travelling salesman problem: the A* algorithm, which employs a heuristic function to identify the most efficient path, and the Stochastic Local Search (SLS) using Adaptive Simulated Annealing (ASA), which explores the search space probabilistically to transition between local optima in pursuit of the global optimum. This document details these algorithms, discussing their key components, implementation strategies, and concludes with an overview of our methodology and the tools to be used for the implementation.

2. A* Algorithm

The A* algorithm stands as a hallmark of pathfinding and graph traversal techniques, renowned for its efficiency and accuracy in finding the shortest path between nodes. Rooted in the principles of heuristics, A* combines the strengths of Dijkstra's algorithm and the Greedy Best-First Search to expedite the search process through informed decision-making. At its core, A* utilizes a heuristic function to estimate the cost from a given node to the goal, thereby guiding the search towards the most promising paths.

2.1. Core Concepts of the A* Algorithm

The A* algorithm is underpinned by several foundational concepts that enable its efficient search mechanism. These concepts include nodes, the evaluation function, and the heuristic function, each playing a critical role in the algorithm's performance.

1) **Nodes:** Nodes represent the discrete points or states in the search space through which the algorithm navigates. Each node corresponds to a possible state or configuration in the problem domain.

2) **Evaluation Function**: The evaluation function, denoted as $f(n)$, is central to determining the path that the A* algorithm will follow. It is defined as:

$$f(n) = g(n) + h(n) \tag{1}$$

where:

- $g(n)$ is the cost of the path from the start node to node n, representing the known cost of reaching n.
- $h(n)$ is the heuristic estimate of the cost to reach the goal from node n, providing an approximation of the remaining distance to the goal.

3) **Heuristic Function:** The heuristic function $h(n)$ estimates the cost to reach the goal from node n. A well-designed heuristic is key to the algorithm's efficiency, guiding the search towards the goal in a manner that minimizes unnecessary exploration. For A* to be both efficient and accurate, $h(n)$ should be admissible (never overestimating the true cost to reach the goal) and consistent (the estimated cost to reach the goal from n is always less than or equal to the cost from any adjacent node m plus the cost of moving from n to m).

2.2. Implementation of A* Algorithm

The A* algorithm finds the shortest path from a start node to a goal node by dynamically selecting the most promising node based on cost estimates. The pseudocode provided (algorithm 1) outlines this process in a stepwise manner, focusing on the algorithm's logical flow.

1) **Initialization:** The algorithm begins with the initialization of the start node. The cost from the start node to itself ($gScore$) is set to zero, and the estimated total cost from the start to the goal ($fScore$) is calculated using the heuristic function h.

2) **Node Selection:** At each iteration, the algorithm selects the node with the lowest estimated total cost ($fScore$) for expansion. This strategy ensures that the search is directed towards the goal in an efficient manner.

3) **Goal Check:** Upon selecting a node, the algorithm checks if it is the goal node. If so, the algorithm reconstructs and returns the path from the start node to the goal, utilizing the recorded parent nodes.

4) **Exploring Neighbors:** For the current node, the algorithm explores all unvisited neighbors. It calculates a tentative $gScore$ for each neighbor, reflecting the total cost of reaching that neighbor from the start node via the current path.

5) **Path Update:** If the tentative $gScore$ of a neighbor is lower than any previously recorded score, this indicates a more efficient path to this neighbor has been found. The algorithm updates the neighbor's $gScore$, $fScore$, and parent node accordingly.

Algorithm 1 A* Algorithm

Time Complexity $\mathcal{O}(b^d)$ where b = branching factor; d = depth of the shortest path
Space Complexity $\mathcal{O}(b^d)$ where b = branching factor; d = depth of the shortest path

```
1: procedure ASTAR(start, goal)
2:     openSet ← PRIORITYQUEUE
3:     openSet.insert(start, h(start))                          ▷ Priority is fScore
4:     cameFrom ← an empty map
5:     gScore ← map with default value of ∞
6:     gScore[start] ← 0
7:     fScore ← map with default value of ∞
8:     fScore[start] ← h(start)
9:     while not openSet.isEmpty() do
10:        current ← openSet.popLowest()                        ▷ Node with lowest fScore
11:        if current = goal then return RECONSTRUCTPATH(cameFrom, current)
12:        end if
13:        for each neighbor of current do
14:            tentativeGScore ← gScore[current]+ DISTBETWEEN(current, neighbor)
15:            if tentativeGScore < gScore[neighbor] then
16:                cameFrom[neighbor] ← current
17:                gScore[neighbor] ← tentativeGScore
18:                fScore[neighbor] ← gScore[neighbor] + h(neighbor)
19:                if neighbor is not in openSet or tentativeGScore < gScore[neighbor] then
20:                    openSet.insertOrUpdate(neighbor, fScore[neighbor])
21:                end if
22:            end if
23:        end for
24:    end while
25:    return failure
26: end procedure
```

6) **Iteration:** The search continues, iterating over the open set until it either finds the goal or exhausts all possible paths. If the open set is empty and the goal has not been reached, the algorithm concludes that there is no viable path.

2.3. Heuristic Function in A* Algorithm

The efficacy of the A* algorithm in solving complex pathfinding and optimization problems, such as the Traveling Salesman Problem (TSP), is significantly enhanced by the choice of heuristic function. We plan to explore two heuristic functions: the Minimum Spanning Tree (MST) and the Reduced Cost Matrix.

2.4. Minimum Spanning Tree (MST) Heuristic

The utilization of the Minimum Spanning Tree (MST) as a heuristic in the A* algorithm provides a compelling approach to estimating the cost to reach the goal in graph-based optimization problems. This heuristic is particularly advantageous in scenarios where path costs are determined by the underlying graph structure rather than geometric distances.

The MST heuristic is grounded in the principle that the cost to connect a set of nodes in a graph cannot be less than the cost of the MST spanning those nodes. In the context of A*, the MST heuristic estimates the minimum remaining cost to reach the goal from a given node by constructing an MST that includes the node itself, any remaining unvisited nodes, and the goal node. This cost represents a lower bound on the actual cost to complete the path, ensuring admissibility.

2.4.1. Implementation. We plan to use Prim's algorithm to construct the Minimum Spanning Tree. The algorithm constructs the MST by gradually growing a tree, starting from the given start node.

The pseudocode (algorithm 2) outlines the steps to calculate the MST's total weight, which is used as the lower bound of the A* algorithm for the given graph.

1) Initialize a priority queue Q with all vertices in the graph G, setting the weight of each vertex to infinity, except for the start node, which is set to 0.

2) A $Parent$ array is used to keep track of the MST structure by storing the parent of each vertex in the MST.

3) The algorithm repeatedly selects the vertex u with the minimum weight from Q and removes it, considering u as the next vertex to be included in the MST.

4) For each vertex v adjacent to u, if v is still in Q and the weight of the edge (u, v) is less than the current weight of v in Q, the algorithm updates v's parent to u and its weight to the weight of (u, v).

5) After all vertices have been processed and included in the MST, the algorithm sums the weights of the edges

Algorithm 2 Calculate MST Heuristic Using Prim's Algorithm

Time Complexity $\mathcal{O}((V + E) \log V)$ where V = number of vertices and E = number of edges
Space Complexity $\mathcal{O}(V + E)$ where V = number of vertices and E = number of edges

```
procedure MSTHEURISTIC(G, start)
    V ← vertices in G
    Q ← V                                   ▷ Priority queue of vertices initialized with infinity
    Q[start] ← 0                            ▷ Start node weight set to 0
    Parent ← array of size |V|, initialized with NULL
    while Q is not empty do
        u ← vertex in Q with min weight and remove it from Q
        for each v adjacent to u in G do
            if v in Q and weight(u, v) < Q[v] then
                Parent[v] ← u
                Q[v] ← weight(u, v)         ▷ Update weight to smaller value
            end if
        end for
    end while
    mstWeight ← 0
    for each v in V do
        if Parent[v] ≠ NULL then
            mstWeight ← mstWeight + weight(Parent[v], v)
        end if
    end for
    return mstWeight
end procedure
```

in the MST to calculate the total weight $mstWeight$, which is returned as the heuristic value.

This MST heuristic calculation provides a systematic approach to estimating the lower bound cost from a node to complete a path.

2.5. Reduced Cost Matrix Heuristic

The Reduced Cost Matrix heuristic is an effective tool for estimating the lower bound of the cost to complete a path for the Traveling Salesman Problem. This heuristic involves reducing the cost matrix of the graph so that each row and column has at least one entry with a value of zero, indicating the minimum cost to move from one node to another. The pseudocode (algorithm 3) outlines the process of calculating the Reduced Cost Matrix heuristic.

2.5.1. Implementation. This algorithm computes the Reduced Cost Matrix heuristic by performing row and column reductions on the cost matrix C of a graph. The key steps involved are:

1) **Row Reduction:** For each row in the cost matrix C, find the minimum value ($rowMin$) and subtract it from every element in that row. This step ensures that each row has at least one zero, representing the minimum cost to leave a node.

2) **Column Reduction:** After row reduction, perform a similar process for each column. Find the minimum value ($colMin$) in each column of the already row-reduced matrix and subtract it from every element in that column. This ensures that each column has at least one zero, reflecting the minimum cost to enter a node.

3) **Heuristic Value Calculation:** The heuristic value (h) is the sum of all $rowMin$ and $colMin$ values found during the reduction process. This value represents a lower bound on the cost to complete the tour since it is the minimum additional cost required to ensure that each node can be entered and left at least once.

3. Stochastic Local Search (SLS)

Stochastic Local Search (SLS) represents a broad category of metaheuristic algorithms that employ randomness as a fundamental component of the search process. Unlike deterministic algorithms, which follow a predefined sequence of steps, SLS methods introduce stochastic variations in the decision-making process, enabling a more diverse exploration of the solution space.

3.1. Adaptive Simulated Annealing (SA)

Simulated Annealing (SA) is a prominent example of an SLS technique inspired by the physical process of annealing in metallurgy. SA seeks global optimization by allowing controlled excursions to suboptimal solutions, thereby avoiding premature convergence to local optima. The algorithm's core mechanism involves a temperature parameter that gradually decreases according to a cooling schedule. High temperatures allow the algorithm to accept worse solutions with higher probability, facilitating exploration. As the temperature lowers, the algorithm becomes more selective, increasingly favoring improvements and stabilizing around an optimum solution.

Algorithm 3 Calculate Reduced Cost Matrix Heuristic

Time Complexity $\mathcal{O}(n^2)$ where n = number of cities
Space Complexity $\mathcal{O}(n^2)$ where n = number of cities

```
procedure REDUCEDCOSTMATRIXHEURISTIC(C)
    n ← number of nodes in C
    R ← copy of cost matrix C
    for i ← 1 to n do
        rowMin ← minimum value in row i of R
        for j ← 1 to n do
            R[i, j] ← R[i, j] − rowMin
        end for
    end for
    for j ← 1 to n do
        colMin ← minimum value in column j of R
        for i ← 1 to n do
            R[i, j] ← R[i, j] − colMin
        end for
    end for
    h ← sum of all rowMin and colMin values found
    return h
end procedure
```

3.1.1. Core Concepts. This optimization technique is designed to find a good approximation to the global optimum of a given function over a large search space. The key concepts underlying SA include:

1) **Temperature:** Central to SA is the concept of temperature, a control parameter that dictates the probability of accepting solutions as the algorithm progresses. Initially set high, the temperature allows the algorithm to explore the solution space freely, accepting both improvements and certain degradations in solution quality to avoid being trapped in local optima.

2) **Cooling Rate:** The cooling rate, denoted by α, is a fundamental parameter that governs the rate at which the temperature decreases during the search process. At each iteration, the cooling rate is applied to the current temperature as follows:

$$T_{new} = \alpha \times T_{current} \tag{2}$$

where $0 < \alpha < 1$. The exact value of α is chosen based on the problem characteristics and desired search properties. A value of α closer to 1 means a slower cooling and thus a longer search process, while a lower α value accelerates cooling, leading to a faster convergence.

3) **Acceptance Probability:** SA introduces a probabilistic rule for accepting new solutions. Even if a new solution is worse than the current one, it may still be accepted with a probability that decreases with the solution's relative quality and the current temperature. This is typically modeled by the Boltzmann distribution:

$$P(e) = \exp(-\Delta E/T) \tag{3}$$

where ΔE is the change in energy (or cost), and T is the current temperature.

4) **Cost Function:** The algorithm evaluates solutions using a cost function specific to the problem being solved. The goal of SA is to minimize this function, with each solution's cost indicating its quality.

5) **Neighborhood Search:** At each step, SA generates a "neighbor" of the current solution by making a small change. This local search mechanism is crucial for exploring the solution space iteratively, guided by the acceptance probability and temperature.

6) **Global and Local Optima:** One of the strengths of SA is its ability to escape local optima—solutions that are optimal within a neighboring set of solutions but not overall. By probabilistically accepting worse solutions, SA can explore beyond local optima in search of the global optimum.

3.1.2. Adaptive Simulated Annealing (ASA). Adaptive Simulated Annealing (ASA) introduces dynamic adjustments to the cooling rate and other parameters based on the algorithm's current state and performance. In standard SA, the cooling rate is typically fixed or follows a predetermined schedule, which may not be optimal for all stages of the search or for different problem landscapes. In contrast, adaptive mechanisms adjust the cooling rate in response to the search's progress.

Since ASA closely resembles SA, and terms will be used interchangeably throughout this section unless otherwise noted.

3.1.3. Implementation. The pseudocode (algorithm 4) provided outlines an implementation of ASA. This section provides the description of the algorithm

1) **Generate Initial Solution:** The algorithm starts with

Algorithm 4 Adaptive Simulated Annealing

Time Complexity $\mathcal{O}(T \cdot \alpha \cdot n^2)$ where n = number of cities; T = Initial Temperature; α = Cooling Rate
Space Complexity $\mathcal{O}(n)$ where n = number of cities

procedure SIMULATEDANNEALINGTSP($cities$, T_{init}, T_{final}, α)
    $currentSolution \leftarrow$ GENERATEINITIALSOLUTION($cities$)
    $currentCost \leftarrow$ CALCULATETOTALDISTANCE($currentSolution$)
    $T \leftarrow T_{\text{init}}$
    while $T > T_{\text{final}}$ **do**
        $newSolution \leftarrow$ GENERATENEIGHBORSOLUTION($currentSolution$)
        $newCost \leftarrow$ CALCULATETOTALTOURDISTANCE($newSolution$)
        $\Delta C \leftarrow newCost - currentCost$
        if $\Delta C < 0$ **or** RANDOM $< e^{-\Delta C/T}$ **then**
            $currentSolution \leftarrow newSolution$
            $currentCost \leftarrow newCost$
        end if
        $\alpha \leftarrow$ COMPUTECOOLINGRATE($currentSolution$, α) ▷ Adaptive cooling rate
        $T \leftarrow T \times \alpha$
    end while
    return $currentSolution$
end procedure

Algorithm 5 Adaptive Simulated Annealing Utility Functions

Time Complexity $\mathcal{O}(n)$ where n = number of cities in tour
Space Complexity $\mathcal{O}(1)$

function COMPUTECOOLINGRATE($solution$, α)
    $acceptanceRate \leftarrow$ `Acceptance threshold` ▷ Fixed value, but can be made dynamic
    $\alpha_{modifier} \leftarrow$ `Modifier value to increase/decrease` α
    $newAcceptanceRate \leftarrow$ EVALUATESOLUTION(solution)
    if $newAcceptanceRate > acceptanceRate$ **then**
        $\alpha_{new} \leftarrow \alpha \times (1 + \alpha_{modifier})$ ▷ Increase α
    else
        $\alpha_{new} \leftarrow \alpha \times (1 - \alpha_{modifier})$ ▷ Decrease α
    end if
    return α_{new}
end function

a randomly generated tour of the cities. This initial solution serves as the starting point for exploration. Further discussion is provided in section 3.2.

2) **Cost Calculation:** For each solution, the total distance ($currentCost$) of the tour is calculated. This serves as the objective function that SA attempts to minimize.
Given a tour represented as an ordered list of cities and a distance matrix where the entry `Distances[i][j]` denotes the distance from city `i` to city `j`, the total distance can be calculated as follows:

    a) Initialize a variable, `totalDistance`, to zero. This will accumulate the total distance of the tour.

    b) For each city in the tour, add the distance from that city to the next city in the tour to `totalDistance`. When considering the last city in the tour, add the distance from the last city back to the first city to complete the circuit.

    c) The value of `totalDistance` after processing all cities in the tour represents the total distance of the tour.

3) **Neighbor Solution Generation:** At each iteration, a new solution ($newSolution$) is generated by slightly altering the current solution, typically by swapping two cities in the tour. Further discussion of this step is provided in section 3.5.

4) **Acceptance Criterion:** The decision to move to the $newSolution$ is based on the change in cost (ΔC) and the current temperature (T). Improving solutions ($\Delta C < 0$) are always accepted, while worsening solutions may be accepted with a probability of $e^{-\Delta C/T}$, allowing the algorithm to escape local optima.

5) **Adaptively Adjust the Cooling Rate:** The function `ComputeCoolingRate` evaluates the current solution's acceptance rate and adjusts α based on whether this rate is higher or lower than a predetermined threshold.

If the new acceptance rate is above the threshold, indicating a higher than desired acceptance of new solutions, α is increased to accelerate the cooling process and focus more on exploitation.
Conversely, if the new acceptance rate is below the threshold, suggesting the algorithm is too quickly converging or getting stuck, α is decreased to slow down the cooling process, thereby encouraging exploration.

6) **Temperature Update:** The temperature is reduced according to the cooling rate ($T \leftarrow T \times \alpha$), gradually shifting the focus from exploration to exploitation as the temperature decreases.

7) **Termination:** The algorithm concludes when the temperature reaches T_{final}, returning the best solution found. This approach ensures that the search is comprehensive in the early stages but becomes increasingly focused as the temperature drops, aiming to find a near-optimal solution to the TSP.

3.2. Generating the Initial Solution

Generating a good initial solution can significantly impact the performance of Simulated Annealing (SA) and other optimization algorithms. While a purely random initial solution is common, employing heuristics to generate the initial solution can provide a better starting point, potentially leading to faster convergence and improved solution quality.

3.3. Insertion Heuristic for Initial Solution

This heuristic method systematically constructs a tour by incrementally inserting cities into an existing path, aiming to minimize the total distance or cost with each addition.

The heuristic's fundamental operation involves selecting a city not yet in the tour and inserting it at the position that results in the smallest possible increase in the total tour length. This process is repeated until all cities are included in the tour for initial solution generation, or applied selectively to modify the current solution, thereby navigating the solution space.

3.3.1. Implementation. The Insertion Heuristic for the Traveling Salesman Problem (TSP) constructs a tour by starting with a single randomly selected city and incrementally adding each remaining city to the tour. The following pseudocode outlines the general procedure of this heuristic.

1) **Initialize the Tour:** Begin with a tour consisting of a single city selected at random from the set of all cities.

2) **Mark Remaining Cities as Unvisited:** Create a set of unvisited cities, initially containing all cities except the one in the tour.

3) **Iterative Insertion:** While there are still unvisited cities, perform the following steps:

   a) For each unvisited city, determine the position in the current tour where inserting the city would result in the smallest increase in the total tour distance. This requires calculating the additional distance for inserting the city between each pair of consecutive cities in the tour.

   b) Select the city and insertion position that leads to the minimum increase in distance.

   c) Insert the selected city into the tour at the identified position and remove it from the set of unvisited cities.

4) **Complete the Tour:** Repeat the insertion process until no unvisited cities remain. The result is a complete tour that visits each city exactly once.

3.3.2. Finding Best Insertion. To determining the optimal insertion point for each city, there are three popular variants of the insertion heuristic, each offering a unique strategy for tour construction:

- **Cheapest Insertion:** Identifies the least costly insertion point for a city across the entire tour, optimizing for minimal increase in tour cost.

- **Nearest Insertion:** Prioritizes the addition of cities closest to any part of the existing tour, promoting local cohesion but not necessarily global optimality.

- **Farthest Insertion:** Starts by inserting cities that are farthest from the current tour, potentially improving global tour structure by initially focusing on distant outliers.

3.4. Greedy Heuristic for Initial Solution

The Greedy Heuristic constructs a tour by repeatedly selecting the shortest available edge from the current city to an unvisited city, thereby incrementally building a path until all cities are visited.

Implementation. The Greedy Heuristic for TSP begins by selecting a random city as the starting point of the tour. It then iteratively expands the tour by choosing the closest unvisited city to the current city, based on the precomputed distances between cities. This process continues until all cities are included in the tour.

- **Starting City:** The heuristic can start from any city in the set of cities. A random selection ensures variability across multiple runs.

- **Selection of Next City:** At each step, the heuristic selects the unvisited city closest to the current city, aiming to minimize the immediate distance added to the tour.

- **Tour Construction:** The tour is incrementally constructed by appending each newly selected city to it, with the process repeating until no unvisited cities remain.

Algorithm 6 Insertion Heuristic For Initial Solution

Time Complexity $\mathcal{O}(n^2)$ where n = number of cities
Space Complexity $\mathcal{O}(n)$ where n = number of cities

```
procedure CompactInsertionHeuristic(Cities)
    Tour ← [SelectRandomCity(Cities)]
    Unvisited ← Cities \ Tour
    while Unvisited ≠ ∅ do
        Insertion ← FindBestInsertion(Tour, Unvisited)
        InsertCityIntoTour(Tour, Insertion.city, Insertion.position)
        Unvisited ← Unvisited \ {Insertion.city}
    end while
    return Tour
end procedure
```

Algorithm 7 Greedy Heuristic for Initial Solution

Time Complexity $\mathcal{O}(n^2)$ where n = number of cities
Space Complexity $\mathcal{O}(n)$ where n = number of cities

```
procedure CompactGreedyTSP(Cities, Distances)
    startCity ← Choose a starting city from Cities
    Tour ← [startCity]
    Unvisited ← Cities \ {startCity}
    while Unvisited ≠ ∅ do
        nextCity ← Find the closest city in Unvisited to the last city in Tour
        Append nextCity to Tour
        Remove nextCity from Unvisited
    end while
    return Tour
end procedure
```

3.5. Generating Neighbor Solutions

Generating neighbor solutions is a fundamental step in iterative optimization algorithms like Simulated Annealing (SA), which rely on exploring the solution space by making small, random modifications to the current solution. In the context of the Traveling Salesman Problem (TSP), a neighbor solution typically involves a slight alteration of the current tour, aiming to discover new routes that may lead to a shorter total distance.

Strategies for Neighbor Generation. Three primary strategies for generating neighbor solutions in the TSP context include Swap, Two-Opt, and Insertion. Each strategy offers a different mechanism for altering the current tour, thereby exploring new potential solutions.

- **Swap:** The Swap strategy involves selecting two cities within the tour and swapping their positions. This method is simple yet effective for local adjustments to the tour, potentially resolving minor inefficiencies.

- **Two-Opt:** The Two-Opt strategy is more involved, aiming to eliminate route crossings that increase the total distance. It selects two non-adjacent edges and swaps their endpoints to create a new connection, effectively reversing the segment of the tour between these edges. This strategy can significantly alter the tour structure and often leads to substantial improvements.

Algorithm 8 Swap Strategy for Neighbor Generation

Time Complexity $\mathcal{O}(1)$
Space Complexity $\mathcal{O}(1)$

```
function SwapNeighbor(Tour)
    i, j ← Two random distinct nodes in Tour
    Swap Tour[i] with Tour[j]
    return Tour
end function
```

Algorithm 9 Two-Opt Strategy for Neighbor Generation

Time Complexity $\mathcal{O}(1)$
Space Complexity $\mathcal{O}(1)$

```
function TwoOptNeighbor(Tour)
    i, j ← Two random distinct nodes in Tour
    Reverse the segment of Tour from i to j
    return Tour
end function
```

- **Insertion:** The Insertion strategy picks a city and inserts it into a different position within the tour. This method is particularly useful for fine-tuning the tour's order and can be guided by various heuristics to choose the most promising insertion points.

Algorithm 10 Insertion Strategy for Neighbor Generation

Time Complexity $\mathcal{O}(1)$
Space Complexity $\mathcal{O}(1)$

```
function InsertionNeighbor(Tour)
    i ← Select a random index in Tour
    city ← Tour[i]
    Remove city from Tour
    j ← Select a new random index in Tour
    Insert city into Tour at position j
    return Tour
end function
```

Implications of Neighbor Generation Strategies. Each neighbor generation strategy has distinct implications for how the solution space is explored:

- **Swap:** Offers localized exploration, making it suitable for fine-tuning near-optimal solutions.
- **Two-Opt:** Provides a more global exploration capability, capable of escaping local optima by addressing inefficient route crossings.
- **Insertion:** Balances between local and global exploration, allowing for adjustments in the sequence of city visits with potentially significant impacts on the tour distance.

4. Implementation

In this section, we define few classes that were crucial in our implementation

4.1. Node

Nodes constitute the primary elements within a graph structure and serve as abstractions for cities in the context of the Traveling Salesman Problem (TSP). Each node is interconnected with another through an entity known as an edge, which will be delineated in the subsequent subsection. In the implementation framework of nodes, two principal parameters are introduced: `id` and `neighbors`.

The `id` parameter assigns a unique identifier to each node. For the sake of simplicity, this identifier is a sequentially incremented value commencing from 0. Alternatively, a more sophisticated approach could involve the utilization of unique identifiers generated through mechanisms such as UUID or GUID, ensuring randomness and uniqueness.

Within the scope of the Minimum Spanning Tree (MST) algorithm's execution, the edge's weight is employed by the priority queue for operational purposes. However, in instances where a tie arises between two edges of identical weight, the resolution is facilitated through the comparison of the destination nodes' IDs. This method of tie breaking is selected arbitrarily and does not influence the ultimate outcome of the algorithm.

The `neighbors` parameter is delineated as a list comprising path edges that emanate from the current node to alternate nodes. This list is structured in a manner that aligns the list index with the node ID, thereby rendering the weight of the edge corresponding to the current node ID as 0. This design decision is predicated on enhancing the accessibility and identification efficiency of destination nodes, rendering the process more intuitive.

4.2. Edge

Edges signify the connections between nodes within a graph. An examination of the TSP graph generator reveals that the generated graph is both complete and undirected, thereby simplifying the implementation and application of edges. Edges are defined by three parameters: `weight`, `source_node`, and `destination_node`.

The `weight` parameter quantifies the cost associated with traversing from the source node to the destination node and is instrumental in the comparative analysis of edges.

Furthermore, `source_node` and `destination_node` encapsulate the node objects themselves. An alternative implementation might involve storing node objects within a dictionary, keyed by their IDs. Consequently, edges could retain the node IDs, allowing for the corresponding node object to be retrieved from the dictionary. This methodology could significantly reduce the memory footprint of the application.

4.3. State

A state encapsulates a specific configuration or instance that potentially contributes towards a solution. The state space, consequently, represents the exhaustive aggregation of all conceivable states. Within the framework of our implementation, a state is characterized by a set of parameters: `cost`, `tour_cost`, `current_city_id`, `path`, and `unvisited`.

The parameters `cost` and `tour_cost` signify the cumulative costs associated with a state. Specifically, `tour_cost` records the accumulated cost of the tour up to the present state, effectively capturing $g(n)$. In contrast, `cost` extends this notion by incorporating the heuristic cost, $h(n)$, thus representing the sum $g(n) + h(n)$.

The `path` parameter chronicles the sequence of nodes traversed by the state, encapsulating the tour navigated thus far. The `current_city_id` parameter facilitates immediate access to the most recently explored node, thereby streamlining the process of extending the path with new nodes.

Furthermore, the `unvisited` parameter maintains a collection of nodes yet to be explored by the current state. This set plays a role in the application of Prim's algorithm, enabling the computation of the Minimum Spanning Tree (MST) for the subset of unvisited nodes in conjunction with the last node explored.

5. Sample Results

5.1. Heuristic Algorithm Experiment

This section details the experiment conducted using a heuristic algorithm. The experiment's primary aim was to evaluate the algorithm's effectiveness in generating an efficient tour

Input Parameter	Value
Number of Cities (n)	10
Number of Distinct Values (k)	30
Mean Value of Distances (μ)	75
Variance of Distances (σ^2)	25
Heuristic Method	Prim's Algorithm for MST
Initial Tour Generation	Greedy Initial Tour

The experiment yielded the following outcomes

Metric	Outcome
Final Tour Sequence	0, 7, 4, 6, 1, 5, 2, 3, 9, 8
Total Cost of the Tour	850.162
Elapsed Time	5522.63 ms

5.2. Stochastic Local Search Algorithm Experiment

This experiment employs a Stochastic Local Search (SLS) algorithm, with a focus on Adaptive Simulated Annealing (ASA). By adjusting parameters such as the cooling rate and thresholds, the experiment aims to uncover efficient tour sequences

Input Parameter	Value
Number of Cities (n)	10
Number of Distinct Values (k)	30
Mean Value of Distances (μ)	75
Variance of Distances (σ^2)	25
Initial Tour Generation	Greedy Initial Tour
Neighbor Tour Generation	Swap neighbor tour
Cooling Rate	1
Cooling Rate Factor	0.0001
Initial Threshold	10000
Final Threshold	1

The SLS algorithm provided the following results, demonstrating its capacity for rapidly identifying efficient tours:

Metric	Outcome
Final Tour Sequence	3, 2, 7, 6, 0, 5, 9, 1, 4, 8
Total Cost of the Tour	508.15
Elapsed Time	7.44421 ms

6. Analysis & Future Work

6.1. A Star

In our investigation of the A* algorithm's performance on the TSP, we observed challenges when dealing with a large number of nodes.

6.1.1. Exponential Growth of State Space. The state space associated with the problem exhibits exponential growth as the node count increases. This complexity became evident when the algorithm was required to explore several million potential states for a problem size of merely 17 nodes.

As a mitigative strategy, we proposed the computation of an approximate initial tour cost to serve as a global heuristic. This approximation was intended to limit the exploration to states where the cumulative cost did not surpass the estimated global tour cost. Any discovery of a tour exhibiting a lower cost would result in an update to this global threshold. Despite this adjustment proving beneficial in certain scenarios, the state space's exponential increase remained a considerable obstacle.

Future work will aim to enhance our approach by developing methods for early identification and elimination of irrelevant states from the exploration space. This refinement seeks to prevent the unnecessary examination of states that are unlikely to contribute to an optimal solution, thereby improving the efficiency of the algorithm.

6.1.2. Heuristic Performing Worse. The application of Prim's algorithm as a heuristic for the MST within the A* framework exhibited suboptimal performance in determining the lowest possible tour cost. Notably, this approach was outperformed by a simplistic heuristic that assigns a zero value to all states (effectively operating without a heuristic).

This observation suggests an imperative for future studies to explore alternative heuristics more closely aligned with the TSP's characteristics. Our analysis indicates that the MST heuristic, as implemented, tends to overestimate the costs associated with state transitions. This overestimation adversely impacts the algorithm's ability to identify the global minimum cost from the initial tour assessments.

6.1.3. Optimizing MST Computation by Caching. During the initial implementation phase of Prim's algorithm, we observed significant computational delays, especially with graphs of merely 10 nodes taking up to a minute for computation. This inefficiency was traced back to the algorithm's handling of similar states, which led to redundant recalculations of the MST cost for these states.

To address this issue, we implemented a caching mechanism. This optimization identifies states with similar unvisited nodes and utilizes cached MST costs to avoid unnecessary recomputation. The introduction of this caching strategy significantly reduced the computation time to under five seconds, demonstrating a substantial improvement in the algorithm's efficiency.

6.2. Adaptive Simulated Annealing

In our investigation of the ASA algorithm's performance on the TSP, we observed challenges when dealing with a large number of nodes.

6.2.1. Significance of Initial Tour. Our approach to constructing the initial tour involved leveraging the simplicity of greedy and insertion heuristics. These methods provided a straightforward means to kick-start the ASA algorithm's search process.

The journey toward the global minimum revealed the critical impact of the initial tour's quality. A tour that closely resembles the global optimum can significantly streamline the ASA algorithm's path to convergence. Despite this, our trials with greedy and insertion heuristics occasionally led to entrapment in local optima. The algorithm's design to escape such traps by selecting lesser solutions didn't consistently facilitate progress toward the global optimum.

Our search for a more effective heuristic led us to the Christofides algorithm. Its appeal lies in the promise to deliver a solution no worse than 1.5 times the optimal

solution. This characteristic positions the Christofides algorithm as a potentially valuable tool for setting a strong starting point for the ASA algorithm.

Time constraints and the technical challenge of integrating the Christofides algorithm prevented its implementation within our project's time-frame. However, its potential for significantly improving our approach makes it a prime candidate for future development.

6.2.2. Identifying Optimal Parameters. Our program accepts the following four inputs: cooling rate, cooling rate factor, initial temperature, and final temperature. Our experimentation revealed that fine-tuning these parameters is vital for effectively broadening the search space.

The rationale behind parameter adjustment stems from the notion that an ideally large search space harbors the potential to uncover the optimal solution. Yet, this becomes a double-edged sword when dealing with extensive graphs, as traversing an overly expansive search space could ostensibly extend into perpetuity.

These parameters are somewhat contingent upon the initial tour's effectiveness. The underlying assumption is that the optimal solution is merely n iterations from the initial tour, thus implying that the search space need not be larger than n. However, since the insertion and greedy algorithms fall short of providing definitive guarantees, estimating these parameters becomes a challenge.

Looking ahead, a promising area for future research involves developing methodologies for the intelligent search of these parameters. Envisioning an algorithm capable of dynamically exploring various parameter configurations could significantly enhance our approach's proximity to the optimal solution.

6.2.3. Alternate Neighbor Solution Heuristics. Our current neighbor solution strategy employs three distinct heuristics: swap, two-opt, and insertion. These methods primarily operate by randomly selecting pairs of nodes to apply their respective transformations. This random selection strategy, while foundational and straightforward, presents an opportunity for enhancement.

Looking ahead to future research directions, we see significant potential in developing algorithms that adopt a more strategic approach to state modification. The core idea revolves around intelligently choosing nodes for alteration based on their potential impact on reducing the overall tour cost. This approach departs from the randomness of current methods, aiming instead to leverage insights gained from the search space's evolving landscape.

Implementing such targeted selection methods could lead to more efficient search strategies, enabling the ASA algorithm to navigate towards optimal solutions with improved accuracy and speed. By adopting a more nuanced understanding of the search space dynamics, these refined heuristics could offer a more sophisticated toolkit for addressing the complexities of the TSP.

7. Conclusion

In this project, we investigated the A* algorithm and Adaptive Simulated Annealing (ASA) to tackle the Traveling Salesman Problem (TSP), a cornerstone challenge within computational optimization. Our exploration revealed the nuanced strengths and limitations of both approaches. The A* algorithm's heuristic-driven pathfinding demonstrated precision and efficiency, yet faced scalability challenges in complex scenarios. ASA, with its stochastic exploration, showcased flexibility in navigating towards optimal solutions, underscoring the importance of heuristic selection and parameter optimization. Future work will focus on enhancing algorithmic efficiency, exploring alternative heuristics, and refining parameter tuning, aiming to bridge the gap between theoretical potential and practical application. This endeavor not only deepened our understanding of algorithmic solutions to TSP but also illuminated pathways for future research in computational optimization.

References

[1] Wikipedia contributors, "Travelling salesman problem — Wikipedia, the free encyclopedia," 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1205913125

[2] ——, "A* search algorithm — Wikipedia, the free encyclopedia," 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1206825447

[3] A. Patel, "Introduction to a*." [Online]. Available: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[4] T. A. et.al, "A* search. brilliant.org." [Online]. Available: https://brilliant.org/wiki/a-star-search/

[5] Wikipedia contributors, "Minimum spanning tree — Wikipedia, the free encyclopedia," 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Minimum_spanning_tree&oldid=1208384769

[6] omar khaled abdelaziz abdelnabi, "Minimum spanning tree." [Online]. Available: https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/

[7] Ole Kröger, "Tsp: Greedy approach and using a 1-tree," 2021. [Online]. Available: https://opensourc.es/blog/tsp-1tree-and-greedy/

[8] Wikipedia contributors, "Adaptive simulated annealing — Wikipedia, the free encyclopedia," 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Adaptive_simulated_annealing&oldid=1191752616

[9] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning," *Operations Research*, vol. 37, no. 6, pp. 865–892, 1989. [Online]. Available: http://www.jstor.org/stable/171470

[10] Wikipedia contributors, "Simulated annealing — Wikipedia, the free encyclopedia," 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Simulated_annealing&oldid=1187355062

[11] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II, "An analysis of several heuristics for the traveling salesman problem," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 563–581, 1977. [Online]. Available: https://doi.org/10.1137/0206041

[12] M. Fischetti and M. Stringher, "Embedded hyper-parameter tuning by simulated annealing," *CoRR*, vol. abs/1906.01504, 2019. [Online]. Available: http://arxiv.org/abs/1906.01504

[13] D. Weyland, "Simulated annealing, its parameter settings and the longest common subsequence problem," in *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 803–810. [Online]. Available: https://doi.org/10.1145/1389095.1389253