ⵜⴰⵙⴷⴰⵡⵉⵜ ⵙⵉⴷⵉ ⵎⵓⵃⵎⵎⴰⴷ ⴱⵏ ⵄⴱⴷⵍⵍⴰⵀ
Université Sidi Mohamed Ben Abdellah

La Faculté des Sciences Dhar El Mahraz – FSDM –

Traveling Salesman Problem Report

A Weighted Approach

Prepared by:

Abdelilah Bajjou and Oussama Ghaouti

December 2024

Contents

Chapter 1

Introduction to the Traveling Salesman Problem (TSP)

1.1 Definition and Problem Statement

The Traveling Salesman Problem (TSP) is a fundamental optimization problem in graph theory and combinatorial optimization. The goal is to find the shortest possible route that visits each city exactly once and returns to the starting city. It is commonly represented using a weighted graph where nodes represent cities and edges represent the distances or costs between them.

1.2 Historical Background

The origins of the TSP can be traced back to the 18th century when it was first conceptualized in mathematical studies. The problem gained prominence in the 20th century with the development of computational methods and algorithms. It remains a central problem in optimization, inspiring research in both theoretical and practical domains.

1.3 Importance in Graph Theory and Optimization

TSP is pivotal in advancing the understanding of graph theory and combinatorial optimization. Solving TSP has significant implications for various applications, including logistics, transportation, and network design. It serves as a benchmark problem for testing optimization algorithms and exploring computational complexity.

Chapter 2

Mathematical Formulation

2.1 Formal Mathematical Definition

The Traveling Salesman Problem (TSP) can be defined as follows: Given a complete weighted graph $G = (V, E)$, where V is the set of vertices (cities) and E is the set of edges (connections between cities), the objective is to find a Hamiltonian cycle (a cycle visiting each vertex exactly once) with the minimum total weight.

The total cost of a tour T is given by:

$$C(T) = \sum_{(i,j)\in T} c_{ij}$$

where:

- T is the set of edges forming the tour.
- c_{ij} is the weight (distance or cost) associated with edge (i, j).

The goal is to minimize $C(T)$, which represents the total travel cost of the salesman.

2.2 Complexity Classification (NP-hard Problem)

The TSP is classified as an NP-hard problem, which implies that there is no known polynomial-time algorithm to solve it exactly for all instances. The complexity arises from the factorial growth in the number of possible tours as the number of vertices increases ($n!$). This makes exact solutions computationally infeasible for large-scale problems.

2.3 Mathematical Models and Representations

TSP can be modeled using various mathematical representations:

1. **Adjacency Matrix:** A matrix A where $A[i][j]$ represents the weight of the edge between city i and city j.

2. **Linear Programming Formulation:**

$$\text{Minimize } \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

$$\text{Subject to:}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in V$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in V$$

$$x_{ij} \in \{0,1\} \quad \forall i,j \in V$$

   Here, c_{ij} is the cost (or distance) between city i and city j, and x_{ij} is a binary variable indicating whether edge (i, j) is included in the tour (1 if included, 0 if not). The goal is to minimize the total cost by selecting the optimal edges that form the tour.

3. **Graph Representation:** Representing cities and routes visually as a graph with nodes and weighted edges.

Chapter 3

Algorithmic Approaches

3.1 Exact Methods

Exact methods are algorithms that guarantee finding the optimal solution to the Traveling Salesman Problem (TSP). These methods explore all possible solutions and select the one with the minimum cost. While they ensure an optimal solution, their time complexity grows exponentially with the number of cities, making them impractical for large instances of the problem.

3.1.1 Brute Force Algorithm (Exhaustive Search)

The Brute Force algorithm generates all possible permutations of cities, calculates the total distance for each permutation, and selects the one with the shortest distance. This method guarantees the optimal solution but is inefficient for large datasets due to its high time complexity.

Worst-case Time Complexity: $O(n!)$ (As all permutations must be examined.)

Space Complexity: $O(n)$ (Space required to store the cities and their distances.)

Advantages: Guarantees the optimal solution.

Consequences: Computationally infeasible for large numbers of cities, as the number of permutations grows factorially.

The following is a Python implementation of the Tabu Search algorithm, which demonstrates how the algorithm works step-by-step to solve the TSP.

```
    import itertools

    def brute_force_tsp(matrix):
    n = len(matrix)
    min_distance = float('inf')
    best_tour = None
    for perm in itertools.permutations(range(n)):
    tour_distance = sum(matrix[perm[i]][perm[i + 1]] for i
       in range(n - 1))
    tour_distance += matrix[perm[-1]][perm[0]]  # Return to
        start
    if tour_distance < min_distance:
    min_distance = tour_distance
    best_tour = perm
    return best_tour, min_distance
```

Listing 3.1: Brute Force Algorithm

3.1.2 Branch and Bound

The Branch and Bound algorithm systematically explores the solution space by dividing it into smaller subproblems (branches) and bounding the possible solutions within each branch. It prunes suboptimal branches, reducing the search space. This method also guarantees an optimal solution but is still computationally expensive for large instances.

Worst-case Time Complexity: $O(n!)$ (In the worst case, no pruning is done.)

Space Complexity: $O(n)$ (Space used to store the state of the search tree.)

Advantages: Pruning reduces the number of branches to explore, often improving performance.

Consequences: Still computationally expensive for large datasets.

The following is a Python implementation of the Branch and Bound algorithm, which demonstrates how the algorithm works step-by-step to solve the TSP.

```
    import heapq

    def branch_and_bound(matrix):
    n = len(matrix)
    pq = []
    heapq.heappush(pq, (0, [0]))  # Cost, Path
    best_cost = float('inf')
    best_path = None

    while pq:
    cost, path = heapq.heappop(pq)
```

```
if len(path) == n:
total_cost = cost + matrix[path[-1]][path[0]]
if total_cost < best_cost:
best_cost = total_cost
best_path = path[:]
continue
for city in range(n):
if city not in path:
new_cost = cost + matrix[path[-1]][city]
if new_cost < best_cost:
heapq.heappush(pq, (new_cost, path + [city]))
return best_path + [best_path[0]], best_cost
```

Listing 3.2: Branch and Bound Algorithm

3.2 Approximation Algorithm

Approximation algorithms provide near-optimal solutions for the Traveling Salesman Problem (TSP) in polynomial time. These algorithms do not guarantee an optimal solution but can give solutions that are close to optimal, often with a known approximation factor. They are typically used when finding the exact solution is computationally infeasible for large problems.

3.2.1 Nearest Neighbor Algorithm (Greedy)

The Nearest Neighbor algorithm is a greedy approach where the salesman starts at a city and always visits the closest unvisited city. This approach provides a quick solution, but it does not guarantee an optimal tour.

Worst-case Time Complexity: $O(n^2)$ (Each step involves finding the nearest neighbor, requiring a scan of all remaining cities.)

Average-case Time Complexity: $O(n^2)$ (Average performance remains similar due to the greedy nature of the algorithm.)

Space Complexity: $O(n)$ (Space used to store the cities and visited states.)

Advantages: Simple and fast; works well for small instances.

Consequences: Can produce suboptimal solutions, especially for large or complex instances.

The following is a Python implementation of the Nearest Neighbor Algorithm, which demonstrates how the algorithm works step-by-step to solve the TSP.

```
def nearest_neighbor(matrix):
n = len(matrix)
unvisited = set(range(n))
current_city = 0
tour = [current_city]
unvisited.remove(current_city)
total_distance = 0

while unvisited:
next_city = min(unvisited, key=lambda city: matrix[
    current_city][city])
total_distance += matrix[current_city][next_city]
current_city = next_city
tour.append(current_city)
unvisited.remove(current_city)

total_distance += matrix[current_city][tour[0]]
tour.append(tour[0])
return tour, total_distance
```

Listing 3.3: Nearest Neighbor Algorithm

3.3 Comparison of Complexity for All Algorithms

Table 3.1: Complexity Comparison of TSP Algorithms

Algorithm	**Time Complexity**	**Space Complexity**	**Optimal Solution?**
Brute Force	$O(n!)$	$O(n)$	Yes
Branch and Bound	$O(n!)$ (Worst case)	$O(n)$	Yes
Nearest Neighbor	$O(n^2)$	$O(n)$	No

Chapter 4

Practical Implementation Considerations

4.1 Programming Approaches

When implementing the Traveling Salesman Problem (TSP), the choice of programming approach plays a crucial role in the effectiveness of the solution. The implementation approach will depend on the algorithm selected (exact, approximation, or metaheuristic) and the desired performance. Below are some key programming considerations:

4.1.1 Exact Methods

Exact methods require careful attention to how computations are performed, especially given the exponential complexity of many algorithms. For instance, methods like brute force and branch and bound involve exploring many possible solutions, which may result in high memory usage and long execution times for large instances.

4.1.2 Approximation Algorithms

Approximation algorithms such as Nearest Neighbor, Christofides, and 2-opt heuristics are often faster and simpler to implement. However, since they do not guarantee optimality, the focus shifts towards ensuring that the approximation is good enough for practical purposes. These methods benefit from efficient iterative updates and use of greedy or local search strategies.

4.1.3 Metaheuristic Algorithms

Metaheuristic approaches, such as Genetic Algorithms, Simulated Annealing, and Ant Colony Optimization, require additional attention to parameter tuning. The quality of solutions generated by these algorithms depends heavily on the setup of the initial population, mutation rate, cooling schedule, pheromone decay, etc. Implementing these algorithms requires iterative testing and fine-tuning to achieve acceptable performance.

4.2 Performance Optimization

For large-scale TSP instances, optimizing the performance of your algorithm is critical. Here are some strategies for optimizing the performance of TSP algorithms:

4.2.1 Parallelization and Multi-threading

In many exact and heuristic algorithms, portions of the computation are independent and can be executed in parallel. For example, in metaheuristic algorithms like Genetic Algorithms or Ant Colony Optimization, different individuals (in genetic terms) or ants (in the case of ACO) can be processed concurrently. Multi-threading can significantly reduce computation time.

4.2.2 Pruning and Early Stopping

For exact methods like Branch and Bound, pruning suboptimal solutions early can drastically reduce computation time. For example, if the lower bound of a branch is higher than the best-known solution, that branch can be discarded. Similarly, heuristics can benefit from early stopping criteria, where the algorithm halts once a sufficiently good solution has been found.

4.2.3 Efficient Data Structures

Optimizing the choice of data structures is crucial for both time and space efficiency. For example, when implementing dynamic programming (like Held-Karp), using a hash table or a matrix to store subproblem results can improve both lookup time and memory usage. Similarly, priority queues or heaps are often used in algorithms like A* or metaheuristics to efficiently retrieve the best solution candidates.

4.2.4 Memory Management and Caching

For large instances of TSP, efficient memory management is essential. Memory usage can be optimized by using more compact data representations, such as adjacency lists instead of adjacency matrices for sparse graphs. Caching results of repeated computations (e.g., distance calculations in heuristics) can save time when the algorithm revisits the same states.

4.2.5 Use of Approximate Solutions for Large Instances

For very large problem sizes, it may not be feasible to compute the exact solution within a reasonable timeframe. In such cases, it may be appropriate to use approximation algorithms or metaheuristics, which can provide good solutions in a fraction of the time compared to exact methods. In practice, approximation algorithms like Christofides or metaheuristic algorithms like Simulated Annealing can strike a balance between solution quality and performance.

4.3 Summary of Considerations

Implementing the TSP algorithm effectively involves not only selecting the right algorithm but also optimizing the approach for performance and efficiency. This requires making informed decisions about programming methods, performance optimizations, and appropriate data structures. In general:

- Exact algorithms are ideal for small to medium-sized problems but are computationally expensive for large instances.
- Approximation algorithms and metaheuristics offer feasible solutions for larger instances but do not guarantee optimality.
- Efficient data structure selection is crucial for handling large problem sizes, particularly when working with sparse graphs or complex algorithms.

Chapter 5

Variations of TSP

The Traveling Salesman Problem (TSP) has several variations, each with its own set of constraints and complexities. These variations arise based on the structure of the graph and the nature of the distances between cities. In this section, we will explore the main variations of TSP and their characteristics.

5.1 Symmetric vs. Asymmetric TSP

The Traveling Salesman Problem can be categorized into symmetric and asymmetric types, based on the structure of the distance matrix.

5.1.1 Symmetric TSP

In Symmetric TSP, the distance between any two cities is the same in both directions. This means that the distance from city i to city j is equal to the distance from city j to city i, i.e., $d(i, j) = d(j, i)$. The problem is usually represented by a symmetric distance matrix, and the goal is to find the shortest possible tour that visits each city exactly once and returns to the starting city.

Advantages: The symmetry of the distance matrix simplifies the problem and reduces computational complexity. This makes Symmetric TSP more straightforward to solve using various algorithms.

Consequences: The symmetry assumption may not always be realistic in real-world applications, where travel times or distances may differ depending on the direction (e.g., one-way streets, differing road conditions).

5.1.2 Asymmetric TSP

In Asymmetric TSP, the distance between cities is not necessarily the same in both directions. This means that $d(i,j) \neq d(j,i)$ for some pairs of cities. Asymmetric TSP is represented by a non-symmetric distance matrix, and the objective remains the same: to find the shortest tour that visits all cities exactly once and returns to the starting city.

Advantages: Asymmetric TSP models real-world scenarios more accurately, such as road networks with one-way streets or flights between airports that do not follow the same routes in both directions.

Consequences: Solving Asymmetric TSP is generally more challenging than Symmetric TSP due to the lack of symmetry in the distance matrix, which increases computational complexity.

5.2 Weighted TSP

In the Weighted TSP, the edges between cities are not necessarily the same in terms of distance or cost. Instead, they are weighted, meaning that each edge has an associated weight that could represent a variety of factors such as distance, time, or cost of travel. The goal is to find the minimum-weight tour that visits all cities exactly once and returns to the start.

Advantages: This variation of the TSP is more flexible and can model a variety of real-world scenarios where the cost of traveling between cities may vary. For example, the weight of an edge could represent fuel cost, time, or monetary expense.

Consequences: The introduction of edge weights adds a layer of complexity, as the problem must now account for the varying costs between cities, which can make it harder to find an optimal solution, especially for large instances.

5.3 Metric TSP

The Metric TSP is a variation in which the distances between cities satisfy the triangle inequality. This means that the direct distance from city i to city j is always less than or equal to the sum of the distances from city i to city k and from city k to city j, i.e.,

$$d(i,j) \leq d(i,k) + d(k,j)$$

Advantages: The triangle inequality helps to simplify the problem, as it ensures that no ”shortcut” between cities can result in a smaller total distance. This property allows for more efficient algorithms and stronger approximation guarantees, such as the 1.5-approximation guarantee for Christofides' algorithm.

Consequences: The Metric TSP still remains NP-hard, but the triangle inequality helps to ensure that approximation algorithms can provide near-optimal solutions within a known factor.

5.4 Euclidean TSP

In the Euclidean TSP, cities are represented as points in a Euclidean plane, and the distances between cities are computed using the Euclidean distance formula. The goal is to find the shortest possible path that visits each city exactly once and returns to the starting point, while traveling along the straight-line (Euclidean) distances between cities.

Advantages: The Euclidean TSP has a clear geometric interpretation, making it easier to visualize and model. It is commonly used in applications where cities are located on a plane, such as in route planning for vehicles, drones, or delivery services.

Consequences: Although the Euclidean TSP is often easier to visualize and understand, it remains NP-hard, and finding the exact solution still requires substantial computational resources. In practice, heuristics and approximation algorithms are often used to obtain near-optimal solutions.

5.5 Summary of Variations

The different variations of the TSP are suited to different problem domains and have varying levels of complexity. In general:

- Symmetric TSP is simpler and more commonly encountered in problems where the distance between cities is the same in both directions.
- Asymmetric TSP is more accurate for modeling real-world scenarios with one-way roads or different travel times between cities.
- Weighted TSP adds complexity by considering the cost or weight of traveling between cities, which can be applied to problems involving time, cost, or resource constraints.

- Metric TSP benefits from the triangle inequality, which allows for more efficient algorithms and approximation guarantees.
- Euclidean TSP is a special case where cities are in a plane and distances are based on the Euclidean metric, making it ideal for certain geometric or spatial applications.

Chapter 6

Real-world Applications

The Traveling Salesman Problem (TSP) has many real-world applications, particularly in fields where optimization of routes, costs, and time is crucial. Below are several key areas where TSP or its variations are applied:

6.1 Logistics and Transportation

In logistics and transportation, the TSP helps optimize routes for delivery trucks or freight, ensuring that goods are delivered in the shortest possible time, reducing fuel consumption and costs. This is critical in industries such as shipping, parcel delivery, and freight management, where optimizing delivery routes leads to significant cost savings and improved customer satisfaction.

6.2 Circuit Board Drilling

The TSP is applied in the manufacturing of circuit boards, where the goal is to minimize the time spent on drilling holes in the board. By optimizing the drilling path, manufacturers can reduce wear on machinery, improve production efficiency, and minimize the time required to complete the drilling process.

6.3 Warehouse Optimization

In warehouse management, TSP is used to optimize the picking paths for workers. By minimizing the distance a worker travels while picking items, the overall efficiency of warehouse operations can be improved. This is especially important in large warehouses where the scale of operations can lead to significant inefficiencies without optimization.

6.4 Vehicle Routing

TSP is widely used in vehicle routing problems, such as those faced by delivery services (e.g., UPS, FedEx) or waste collection. The problem involves determining the optimal route for a fleet of vehicles to visit multiple locations while minimizing travel time, fuel consumption, or operational costs. The TSP provides an essential framework for solving these types of logistical problems.

6.5 Supply Chain Management

Supply chain management often requires optimizing routes and deliveries between suppliers, warehouses, and retailers. TSP can help companies plan efficient routes for transportation, thereby reducing costs and improving the overall supply chain performance. This application is particularly beneficial in industries that rely on fast and cost-effective distribution, such as e-commerce and food distribution.

Chapter 7

Conclusion and Future Perspectives

7.1 Summary of Key Findings

In this report, we have explored various algorithmic approaches to solving the Traveling Salesman Problem (TSP), from exact methods to approximation and metaheuristic approaches. The exact methods, including Brute Force, Branch and Bound, Dynamic Programming, and Integer Linear Programming, guarantee the optimal solution but face exponential time complexity, making them impractical for large instances. Approximation algorithms like Nearest Neighbor and Christofides provide near-optimal solutions with more efficient time complexities, while metaheuristic approaches such as Genetic Algorithms, Simulated Annealing, and Ant Colony Optimization are well-suited for large-scale problems, although they do not guarantee optimality.

We also discussed practical considerations for implementing TSP algorithms, including programming approaches, performance optimization, and the selection of appropriate data structures. Furthermore, the real-world applications of TSP in logistics, circuit board drilling, warehouse optimization, vehicle routing, and supply chain management were highlighted, demonstrating the importance of TSP in various industries.

7.2 Emerging Research Directions

Despite significant advancements in solving the TSP, there are still many areas for further research:

- **Hybrid Approaches:** Combining exact methods with approximation or metaheuristics could yield more efficient algorithms that balance optimality and computational feasibility. Research into hybrid approaches may lead to better solutions for large-scale instances of TSP.

- **Parallel and Distributed Computing:** With the growing availability of parallel and distributed computing resources, future research could focus on parallelizing TSP algorithms to improve their scalability and performance on modern hardware.
- **Quantum Computing:** Quantum algorithms, such as quantum annealing, offer the potential for solving TSP more efficiently than classical algorithms. Research in quantum computing for combinatorial optimization problems could lead to breakthroughs in TSP solutions.
- **Real-time and Dynamic TSP:** Many real-world problems involve dynamic environments where new cities or routes are added during the optimization process. Research into real-time algorithms that can adjust to these dynamic changes is a promising direction.
- **Stochastic and Uncertainty-based TSP:** In real-world applications, data may be uncertain or probabilistic. Research into TSP variants that account for uncertainty in distances or travel times is an emerging area of interest.

7.3 Potential Improvements

While the existing methods for solving TSP have proven useful, there are several areas where improvements can be made:

- **Algorithmic Efficiency:** Although approximate and metaheuristic algorithms perform well on large instances, their efficiency can still be improved. Further research into more efficient heuristics and algorithms could help address larger and more complex TSP instances.
- **Scalability:** As the number of cities increases, the solution time for exact methods grows exponentially. Future developments may focus on enhancing the scalability of exact methods, such as using parallel processing or more efficient data structures.
- **Better Bounds for Approximation Algorithms:** While algorithms like Christofides provide a bounded approximation, there is still room

for improvement in terms of tighter bounds and faster approximation algorithms for TSP. Research into improving these bounds can help increase the reliability of solutions.

- **Integration with Other Optimization Problems:** TSP is often part of more complex optimization problems, such as Vehicle Routing Problems (VRP) and Multi-Depot Problems. Future research could focus on integrating TSP with these problems to create more comprehensive solutions for real-world applications.

- **Improved Metaheuristics:** While metaheuristics such as Genetic Algorithms, Simulated Annealing, and Ant Colony Optimization provide good solutions, their performance could be enhanced by incorporating more advanced techniques like adaptive parameter tuning, multi-objective optimization, and hybridization with other methods.

In conclusion, TSP continues to be a critical problem in optimization, with significant advancements made in solving the problem using both exact and heuristic methods. However, challenges remain in terms of scalability, computational efficiency, and applicability to real-world scenarios. Future research promises to further improve the efficiency and practicality of TSP solutions, with exciting developments in hybrid methods, quantum computing, and dynamic problem-solving approaches.