POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Matematica

Tesi di Laurea Magistrale

Heuristic Algorithms for the Min-Max Traveling Salesman Problem

Supervisors
Prof. Federico Della Croce di Dojola

Candidate
Federico Rondano

Academic Year 2024-2025

Acknowledgements

A tutte le persone che hanno creduto in me.
Al lavoro svolto in questi anni.

Abstract

Optimization plays a fundamental role in solving real-world decision-making problems, in which the objective is to maximize efficiency while minimizing costs. The Traveling Salesman Problem (TSP) is a classical combinatorial optimization problem that results in numerous applications in logistics, manufacturing, and network design. Extending this to multiple agents, the Multiple Traveling Salesman Problem (mTSP) assigns a set of destinations (nodes of a graph) to multiple salesmen, introducing additional complexity in terms of workload distribution and coordination.

A critical variant is the Min-Max mTSP, where the objective is not only to construct good feasible tours, but to minimize the length of the longest one. This formulation is particularly relevant in scenarios that require balanced task allocations, such as vehicle routing, robotics, and job allocation. Unlike the classical TSP, where minimizing total travel cost leads to well-established heuristics, the Min-Max objective requires specialized techniques to ensure fairness across tours while maintaining computational efficiency.

This thesis explores new neighborhood search based approaches for tackling the Min-Max m-TSP, focusing on two key aspects: the initial distribution of nodes among subtours and the iterative refinement of solutions towards convergence. Traditional partitioning methods, such as clustering, do not fit this problem well, leading to highly unbalanced initial solutions. To address this issue, novel strategies will be introduced to build more structured node configurations. Furthermore, new ideas will be developed to iteratively explore and refine solutions, balancing the trade-off between exploration and exploitation in the solution search space.

In conclusion, by combining theoretical analysis with computational experimentation, this work aims to provide innovative methodologies and ideas for solving Min-Max m-TSP efficiently. The proposed techniques will be evaluated in terms of solution quality, stability, and scalability.

Contents

List of Tables

Chapter 1

Introduction

This thesis aims to explore the role of heuristic algorithms in solving the Min-Max Multiple Traveling Salesman Problem, a well-known combinatorial optimization challenge. By analyzing both the theoretical and practical aspects, this work attempts to offer insight into some approaches for solving such problem in a new way.

After a brief introduction on the role of optimization and its importance, we will discuss the algorithms and tests of this problem.

1.1 The concepts behind optimization

Optimization problems and models can be applied in many different areas of modern society. Whether we are looking for the shortest route between two or more locations, determining the optimal pricing strategy for a product, solving puzzles, or making decisions based on multiple criteria, optimization theory can provide the mathematical framework to address these challenges. The field spans various disciplines, including engineering, economics, operations research, and computer science.

The **goal of optimization** is to make the best possible decision given a set of constraints, resources, and objectives. This is represented by the objective function, where the criterion that needs to be optimized is expressed in different terms based on the objective to be achieved. For example: maximize efficiency, minimize cost, or find the most balanced configuration.

Mathematically the objective function to be optimized can be described as $f(x)$, where x is a vector of decision variables: x are the unknowns to be determined and represent the possible solutions to the problem. They could be continuous, discrete, or a combination of both.

For instance, in the traveling salesman problem (TSP), the variables represent the order in which cities are visited, while in a mixed-integer programming problem, some variables might represent binary decisions, and others could take continuous values.

The function lies on a domain Ω, which is the set of feasible solutions, bounded by constraints that may arise from real-world limitations or operational requirements. Constraints are essential in defining this domain and they can be either equality or inequalities.

Definition: Optimization Problem

An optimization problem consists of an objective function $f(x)$, a set of decision variables x, and a feasible region Ω defined by constraints. The goal is to find $x^* \in \Omega$ such that:

$$\max_{x \in \Omega} f(x).$$

It is important to note that, when applying the operators max and min, the result is the optimal value of the objective function, denoted $f^* = f(x^*)$, corresponding to a specific point x^* within the domain.

However, these operators do not return the point x^* itself. If the goal is to identify the point where the objective function reaches its optimum, one should instead use the arg max or arg min operators. These operators yield the set of points in the domain where the function achieves its optimal value.

Additionally, note that, when the objective function is linear, the problem can be framed as a minimization without loss of generality. This is because minimizing a function $f(x)$ is equivalent to maximizing its negation, that is,

$$\min_{x \in \Omega} f(x) = \max_{x \in \Omega} -f(x).$$

This provides flexibility in how optimization problems are expressed and solved.

1.2 Challenges in Solving Optimization Problems

You can see that, while optimization provides a powerful framework for decision-making, many real-world problems are difficult to solve. Some of the main challenges include:

- **Scalability**: as the size of the problem increases, the complexity of solving it grows. This is especially true for combinatorial problems such as the TSP, where the number of possible solutions grows factorially with the number of cities.
- **Constraints**: incorporating complex constraints increases the difficulty of solving the problem, requiring specialized algorithms or relaxation techniques.
- **Computational Resources**: even with advancements in computing power, solving large-scale optimization problems often requires significant computational resources and time.

1.3 Computational Complexity

As said above, some problems can be efficiently solved to optimality using specific algorithms. Other problems are computationally impractical, hence they cannot be solved efficiently. Here, computationally impractical means that, if the input is large enough, the expected running time quickly becomes unmanageable.

For this reason, computational complexity is fundamental in theoretical computer science and optimization. It refers to the amount of computational resources required to solve a problem, typically measured in terms of time (time complexity) and memory space (space complexity) used. The complexity of an algorithm is commonly expressed as a function of the input size n, using notation such as $O(n)$, $O(n^2)$, or $O(\log n)$, which describes the algorithm's behavior as n grows. See [10].

1.3.1 Decision Problems

A decision problem is a problem with a binary answer (yes or no). For instance, while the problem "How long is the shortest path from A to B?" is an optimization problem, it can be reformulated as a decision problem: "Does there exist a path from A to B shorter than 5 units?" In this form, it becomes a yes/no question. Formally, decision problems can be represented as sets of binary strings that satisfy certain properties.

1.3.2 Algorithms and Complexity

An *algorithm* A can be described as a computational procedure that transforms an input binary string x into an output binary string y. Given a decision problem Π, the algorithm A solves Π if the set of inputs for which $A(x) = 1$ corresponds to the set Π. Importantly, an algorithm must follow a finite sequence of well-defined operations to achieve this transformation.

The *complexity* of an algorithm A is denoted by $f(n)$, where n is the length of the input string, and $f(n)$ is the maximum number of operations needed to transform the input x into the output y. This complexity is thus a function of the input size rather than the specific input values.

1.3.3 The Classes P and NP

Two fundamental classes of computational complexity are:

- **P (Polynomial Time)**

  Definition: Class P

  A decision problem Π belongs to the class **P** if there exists an algorithm A that solves it in polynomial time. This means that the complexity function $f(n)$ of the algorithm is bounded by a polynomial function of input size n, i.e., $f(n) = O(n^k)$ for some constant k. The problems in **P** are considered tractable, as they can be solved efficiently even for large inputs.

- **NP (Nondeterministic Polynomial Time)**

Definition: Class NP

A decision problem Π belongs to the class **NP** if there exists an algorithm V (called a *verifier*) and a polynomial function $p(n)$ such that:

- If $x \in \Pi$, then there exists a binary string $y \in \{0,1\}^{p(n)}$ (called a *certificate*) such that $V(x, y) = 1$, meaning that y verifies that x is a valid solution.
- If $x \notin \Pi$, then for every possible certificate $y \in \{0,1\}^{p(n)}$, the verifier returns $V(x, y) = 0$.

In other words, *problems in NP* are those for which a proposed solution can be *verified* in polynomial time, but finding the solution may take longer.

An open problem in computational complexity is the **P vs. NP** question, one of the "Millennium Problems". Informally, it asks whether every problem for which a solution can be verified efficiently (i.e., in polynomial time) can also be solved efficiently (again, in polynomial time), or if there exist problems that are inherently difficult to solve but easy to verify. If $\mathbf{P} = \mathbf{NP}$, then all these problems would be solvable in polynomial time, but if $\mathbf{P} \neq \mathbf{NP}$, then some problems may require much more time to solve, but their solutions could still be verified quickly. Right now, it is formally said that $\mathbf{P} \subseteq \mathbf{NP}$. This remains one of the most important open questions in computer science.

Now, what can be said about those problems that are commonly called difficult?

1.3.4 NP-Complete and NP-Hard Problems

Definition: NP-complete problem

A decision problem Π is said to be *NP-complete* if it satisfies two conditions:

1. $\Pi \in \mathbf{NP}$, i.e., the problem can be verified in polynomial time by a deterministic algorithm, given a solution (a certificate).
2. Every problem in **NP** can be reduced to Π in polynomial time. This means that for any problem $\Pi' \in \mathbf{NP}$, there exists a polynomial-time computable function f that transforms an instance of Π' into an instance of Π such that the answer to Π' is "yes" if and only if the answer to Π is "yes".

This second condition is known as *polynomial-time reduction* and ensures that NP-complete problems are at least as difficult to solve as any other problem in NP.

An important theorem regarding NP-complete problems is Cook's Theorem:

Cook's Theorem

The Boolean satisfiability problem (SAT) is NP-complete.

Cook's Theorem was the first to demonstrate the existence of an NP-complete problem, providing a foundation for identifying other NP-complete problems via polynomial-time reductions.

Definition: NP-hard problem

A problem is classified as *NP-hard* if it is at least as hard as any problem in **NP**, but it is not necessarily a decision problem or may not belong to **NP**.

In other words, NP-hard problems may not be verifiable in polynomial time. However, any problem in **NP** can still be reduced to an NP-hard problem in polynomial time.

To better understand this concept, here is an **example**:
the decision version of *TSP* is NP-complete because it is in NP (a candidate solution can be verified in polynomial time) and any instance of the Hamiltonian Cycle Problem (which is NP-complete) can be reduced to an instance of *TSP*. Therefore, *TSP* is at least as difficult as any problem in NP.
The optimization version of *TSP*, which seeks to minimize the total distance of the tour, is NP-hard because it is not a decision problem, but is still at least as difficult as solving the decision version.

1.3.5 Algorithmic Complexity and Examples of Running Times

Definition: Time Complexity

Time complexity describes how the running time of an algorithm increases with the size of the input. This is expressed using the asymptotic notation $O(f(n))$, which gives an upper bound on the growth rate of the algorithm's running time as a function of the input size n.
We are considering the *worst case*, for example: $f(n) \in O(g(n))$ means there exist constants $\hat{n}$ and $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq \hat{n}$.

The table illustrates some common algorithmic complexities and provides examples of their expected running times for different input sizes:

Algorithm Complexity	**n=10**	**n=20**	**n=30**	**...**	**n=60**
$f(n) \in O(n)$	0.001 sec	0.002 sec	0.003 sec	...	0.006 sec
$f(n) \in O(n^2)$	0.001 sec	0.004 sec	0.009 sec	...	0.36 sec
$f(n) \in O(2^n)$	0.001 sec	1.0 sec	17.9 min	...	centuries

Table 1.1: Algorithms Computational Complexity

The complexity class $O(n)$ grows linearly with the input size, $O(n^2)$ follows a square trend, while $O(2^n)$ grows exponentially, quickly becoming infeasible even for relatively small inputs. This distinction underlines the importance of designing algorithms with efficient time complexity, especially for large-scale problems.

Example of Sorting Algorithms

To clarify the concept of time complexity, consider the example of sorting a list of n real numbers in ascending order. Different algorithms can solve this problem with varying levels of efficiency:

- A trivial way to solve this problem is to **iterate through all possible permutations** of the list and check if any permutation is sorted. Since there are $n!$ permutations of n elements, and checking if a permutation is sorted takes $O(n)$ time, the total number of operations would be $O(n! \cdot n)$. This approach is clearly inefficient and results in a non-polynomial time complexity, it is impractical even for small values of n.

- More practical sorting algorithms include, for example, the **Insertion Sort**. This algorithm iteratively inserts each element in its correct position in a sorted sub-list. Its time complexity is $O(n^2)$, which means that its execution time grows quadratically with the input size. Although better than $O(n!)$, it is still not optimal for large inputs.

- There are even more efficient algorithms, for example the **Heapsort**, which has a time complexity of $O(n \log n)$. This is in practice the best possible time complexity for comparison-based sorting algorithms, as it can be shown that no comparison-based sorting algorithm can achieve better than $O(n \log n)$ in the worst case.

This shows how different algorithms can solve the same problem with significantly different time complexities, making it crucial to choose efficient algorithms, especially when dealing with large data sets.

Why Complexity Matters in Optimization

In optimization, computational complexity is critical because many real-world problems involve large input sizes, and the time it takes to compute solutions can grow rapidly with the problem size.

For example, combinatorial optimization problems, such as the traveling salesman problem, are NP-hard, meaning they are at least as difficult as the hardest problems in NP. As a result, exact solutions are often impractical for large instances, inducing the use of heuristic algorithms that that can provide near-optimal solutions in a reasonable amount of time.

1.4 Exact vs approximate approaches

1.4.1 Exact Optimization Algorithms

Exact optimization algorithms are those that guarantee finding the global optimal solution of an optimization problem, provided that sufficient computational resources are available. These algorithms ensure that the computed solution is the best available satisfying all problem constraints.

Definition: Exact Optimization Algorithm

An *exact optimization algorithm* is a procedure that, for a given optimization problem P with an objective function $f(x)$ and feasible region Ω, finds the solution x^* such that:

$$x^* = \arg\min_{x \in \Omega} f(x) \quad \text{or} \quad x^* = \arg\max_{x \in \Omega} f(x)$$

where $f(x^*)$ represents the globally optimal value of the objective function within the feasible region.

Formally, consider a minimization problem of the form:

$$\min_{x \in \Omega} f(x)$$

An exact optimization algorithm will always produce a solution x^* such that $f(x^*) \leq f(x)$ for all $x \in \Omega$. For maximization problems, the algorithm produces x^* such that $f(x^*) \geq f(x)$ for all $x \in \Omega$.

Exact algorithms are typically applicable to problems where the structure of the feasible region or the properties of the objective function allow for efficient exploration of the solution space. Common exact optimization algorithms include:

- **Brute-force Search**: this approach, while being conceptually simple, involves enumerating every possible solution within the feasible region Ω and selecting the best one according to the objective function. This method guarantees finding the exact solution, since it exhaustively explores the solution space.

  For example, consider a simple minimization problem where $\Omega = \{x_1, x_2, \ldots, x_n\}$ is finite and small. The brute-force algorithm evaluates the objective function at every point:

  $$f(x_1), f(x_2), \ldots, f(x_n)$$

  and chooses x^* such that $f(x^*) = \min\{f(x_1), f(x_2), \ldots, f(x_n)\}$.

  While this method is guaranteed to be exact, it is obviously computationally impractical for large problems because the number of possible solutions grows exponentially with the size of the problem. For instance, the brute-force solution to the Traveling Salesman Problem (TSP) would require evaluating all $(n-1)!$ possible tours, which becomes infeasible for even moderately sized n.

- **Branch and Bound**: *Branch and Bound (B&B)* is a widely used exact algorithm for solving combinatorial optimization problems and is especially effective in solving mixed-integer and integer programming problems.

  The basic idea behind B&B is to divide the original problem into smaller subproblems (branching) and evaluate bounds on the objective function for these subproblems.

  Consider a minimization problem. The algorithm proceeds as follows:

  1. Start with the original problem and compute a lower bound on the optimal solution.
  2. Divide (branch) the problem into subproblems by adding constraints that simplify the problem.
  3. For each subproblem, compute lower bounds on the objective function. If the lower bound is worse than the current best feasible solution, discard (prune) the subproblem.
  4. Continue branching and bounding until all subproblems are either solved or pruned.

  Formally, let the original problem be denoted as P_0 with objective function $f(x)$ and feasible region Ω. The algorithm creates a tree of subproblems $P_1, P_2, \ldots$ where each node in the tree corresponds to a subproblem. Then evaluates a lower bound on each subproblem.

  Example: Solving an Integer Programming Problem with Branch and Bound.

  Consider the integer programming problem:

  $$\min 3x_1 + 4x_2 \quad \text{subject to} \quad x_1 + x_2 \geq 5, \quad x_1, x_2 \in \mathbb{Z}_+$$

  The B&B algorithm divides this problem into subproblems by branching on integer values of x_1 and x_2, computes lower bounds on each subproblem, and prunes subproblems where the bounds exceed the current best solution. This approach reduces the search space dramatically, making the problem tractable, even for larger instances.

- **Branch and Cut**: is an exact algorithm that extends the Branch and Bound (B&B) method by integrating additional techniques from cutting-plane methods. It is particularly well-suited for solving mixed integer programming (MIP) problems, where the goal is to optimize a linear objective function subject to both integer and continuous variables constrained by linear inequalities.

  The idea behind Branch and Cut is to start with a linear programming (LP) relaxation of the integer programming problem (where the integer constraints are temporarily ignored). If the LP solution is not feasible for the original integer problem (i.e., it produces fractional values for integer variables), the algorithm generates additional *cutting planes*. These are inequalities that "cut off" the fractional solution, tightening the feasible region of the LP relaxation without removing any feasible integer points. The process continues, alternating between branching (as in B&B) and adding cutting planes, until the optimal integer solution is found.

Steps of Branch and Cut:

1. **Relaxation**: Solve the LP relaxation of the original problem (correspondingly obtaining a lower bound for the problem).
2. **Cutting Planes**: If the LP solution is fractional, generate and add cutting planes that exclude the current solution while maintaining all integer solutions.
3. **Branching**: If no further cuts are effective, branch the problem into smaller subproblems by imposing constraints on the integer variables (as in Branch and Bound).
4. **Iteration**: Repeat the process of solving subproblems, adding cuts, and branching until the integer solution is obtained.

- **Dynamic Programming (DP)**: is a method that solves complex problems by breaking them down into simpler subproblems, solving each subproblem only once and storing its solution for future use. The idea is to exploit subproblems and optimal substructure, which means that the global optimal solution can be constructed from the optimal solutions of its subproblems.

  The Bellman equation, a recursive relation, characterizes dynamic programming. For instance, consider the shortest path problem where we need to minimize the cost $C(i)$ of reaching the destination from node i.

  The Bellman equation for this problem is:

  $$C(i) = \min_{j \in N(i)} \{c(i,j) + C(j)\}$$

  where $N(i)$ is the set of neighbors of node i, and $c(i,j)$ is the cost of traveling from node i to node j. This is also called Value Function for Dynamic Programming.

  Example: Knapsack Problem (KP)

  The knapsack problem can be efficiently solved using dynamic programming. The objective is to maximize the value of items placed in a knapsack of fixed and limitated capacity W. Let v_i be the value and w_i be the weight of the i-th item. The recursive equation for this problem is:

  $$V(i,w) = \max\{V(i-1,w), V(i-1,w-w_i) + v_i\}$$

  where $V(i,w)$ represents the maximum value that can be obtained with i items and capacity w.
  Using such strategy, we can represent our value function in a tabular form and solve KP.

- **Simplex Algorithm for Linear Programming**: is a well-known exact optimization algorithm for solving linear programming (PL) problems. Given, for example, a linear problem in standard form:

$$\min c^T x \quad \text{subject to} \quad Ax = b, \quad x \geq 0$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$. The simplex algorithm moves along the edges of the feasible region, defined by the polyhedron $Ax \leq b$, to find the optimal vertex, corresponding to the minimum value of the objective function. Despite its exponential worst-case time complexity, the simplex algorithm performs efficiently in practice.

Each exact algorithm has its advantages and limitations, depending on the structure and size of the optimization problem. The table below summarizes and makes a comparison of the key features of the exact algorithms discussed:

Algorithm	**Strengths**	**Weaknesses**
Brute-force approach	Guaranteed to find optimal solution	Exponential time complexity for large problems
Branch and Bound (B&B)	Efficient for certain integer and combinatorial problems	May still require exponential time in worst case
Branch and Cut	Combines B&B and cutting for MIP problems; handles large instances effectively	Complex to implement, may still require exponential time in worst case
Dynamic Programming (DP)	Solves problems with overlapping subproblems optimally	Requires problem-specific recursive formulation
Simplex algorithm	Efficient for LP problems with thousands of constraints	Exponential worst-case time complexity

Table 1.2: Comparison of some Exact Optimization Algorithms

1.4.2 Approximate Optimization Algorithms

In many real-world optimization problems, especially those classified as **NP-hard**, finding the optimal solution is often computationally infeasible for large input sizes. Even if an optimal solution could be found, proving its optimality may be impractical in a reasonable time.

This forces us to relax our requirements for exact solutions and instead aim for *suboptimal*, yet computationally efficient, solutions. These suboptimal solutions are obtained through the use of **heuristics** and **approximation algorithms**. They are approaches designed to find solutions that are "good enough" in situations where exact algorithms fail to scale.

1.4.3 Heuristics

Heuristics are algorithms designed to be effective in practice. They are generally based on intuitive strategies that work well in most real-life instances but come without guarantees on solution quality or running time. Famous examples include greedy algorithms, local search based approaches (such as, for instance, *simulated annealing*, *tabu search* and *greedy randomized adaptive search procedures*) and population based approaches (such as, for instance, *ant colony optimization* and *genetic algorithms*).

- **Greedy Algorithms**: they build a solution iteratively by selecting the locally optimal choice at each step (make the most profitable decision at each step, without considering the future impact of these decisions). Although the greedy strategy can fail to find the global optimum in some cases, it is often efficient and yields good results for certain types of problems, in particular if it is combined with other strategies.

- **Beam Search**: considered as an extension of the greedy algorithm, where the key difference is its ability to account for a *fixed amount of foresight.* While a greedy algorithm makes decisions based solely on the immediate state, Beam Search expands this by exploring a limited number of future possibilities, allowing it to avoid short-sighted choices that could lead to suboptimal solutions.

  This implementation provides a more informed choice than a purely greedy approach, yet it remains computationally feasible by restricting the search to a fixed number of alternatives, rather than exploring the entire decision tree.

  Despite this additional foresight, Beam Search still does not guarantee finding the optimal solution, as the fixed beam width may miss promising paths.

- **Local Search**: this method begin with an initial feasible solution and iteratively improves it by exploring neighbor solutions. In particular it explores a neighborhood $N(x)$ of the current solution x and moves to a better solution if one exists.

  While local search may converge to a local optimum rather than the global optimum, it is often used in practice because of its simplicity and effectiveness.

  Formally, let $x_1, x_2, \ldots$ be a sequence of solutions such that:

  $$x_{i+1} = \arg \min_{x \in N(x_i)} f(x),$$

  where $f(x)$ is the objective function. The search terminates when no better solution is found in the neighborhood.
  The general framework for local search can be described as follows:

  1. **Initialization**: start with an initial feasible solution x_1.
  2. **Neighborhood generation**: build a neighborhood $N(x_i)$ of the current solution x_i and select a candidate solution $\tilde{x} \in N(x_i)$.

It is a set of solutions that can be reached from x_i by making small changes to it. A simple example for TSP is the *2-exchange neighborhood*, where two cities in the tour are swapped to create a new tour.

3. **Acceptance test**: check if the candidate solution $\tilde{x}$ should replace x_i as the current solution. If accepted, set $x_{i+1} = \tilde{x}$; otherwise, keep $x_{i+1} = x_i$.
4. **Stopping test**: if a termination criterion is met (e.g., no improvement in the neighborhood), stop the search and return the best solution found.

It is often combined with other methods in order to explore different neighborhoods and improve its performance, such as *tabu search* and *simulated annealing*, to avoid local optima by introducing randomness or memory-based techniques.

- **Simulated Annealing (SA)**: is a probabilistic technique used to avoid local optima by allowing moves to worse solutions. It can be seen as an extension of Local Search that introduces randomness, mimicking the process of annealing in metallurgy.

  The main idea behind Simulated Annealing is that it explores the solution space by allowing the algorithm to accept worse solutions with a certain probability, which decreases as the algorithm progresses. The probability of accepting a worse solution depends on a parameter called the *temperature*, which gradually decreases over time.

  Formally, let x_i be the current solution and $x_{\text{new}} \in N(x_i)$ a candidate solution in the neighborhood of x_i. The acceptance probability $P(x_i, x_{\text{new}})$ is given by:

$$P(x_i, x_{\text{new}}) = \begin{cases} 1 & \text{if } f(x_{\text{new}}) < f(x_i), \\ \exp\left(\frac{f(x_i) - f(x_{\text{new}})}{T}\right) & \text{if } f(x_{\text{new}}) \geq f(x_i), \end{cases}$$

  where T is the current temperature and $f(x)$ is the objective function.

  The general framework for Simulated Annealing can be described as follows:

  1. **Initialization**: start with an initial feasible solution x_0 and an initial temperature T_0.
  2. **Neighborhood generation**: at each iteration, generate a candidate solution $x_{\text{new}} \in N(x_i)$, where $N(x_i)$ represents the neighborhood of the current solution.
  3. **Acceptance test**: compute the acceptance probability $P(x_i, x_{\text{new}})$. If the new solution is better than or equal to, accept it. If it is worse, accept it with a probability given by the Boltzmann distribution.
  4. **Cooling schedule**: decrease the temperature T according to a predefined cooling schedule (e.g., $T = \alpha T$ for some $0 < \alpha < 1$).
  5. **Stopping test**: the algorithm terminates when the temperature reaches a predefined threshold or no improvement is seen after a number of iterations.

- **Tabu Search (TS)**: is an iterative algorithm designed to maintain a memory structure, known as a *tabu list*, which restricts certain moves. With this approach it is possible to explore a wide solution space.

The main idea behind Tabu Search is that it avoids cycling back to recently visited solutions by marking certain moves as tabu (forbidden) for a fixed number of iterations. This encourages the algorithm to search in unexplored regions of the solution space, potentially finding or getting close to global optima.

Formally, let x_i denote the current solution and $N(x_i)$ the set of feasible candidate solutions in its neighborhood. The selection of the next candidate $x_{\text{new}} \in N(x_i)$ follows a specific rule:

$$x_{\text{new}} = \arg \min_{x' \in N(x_i) \setminus \text{Tabu List}} f(x'),$$

where $f(x)$ is the objective function and the *Tabu List* restricts moves by tracking recently altered solution attributes.

- **Greedy Randomized Adaptive Search Procedure (GRASP)**: is a multi-start or iterative process metaheuristic for combinatorial optimization. The method consists of two phases: construction (generating a solution) and local search (finding a local minimum). The construction phase builds a feasible solution by adding the best available component according to a greedy function that is randomized by limiting the choice to those components within the best candidates (Restricted Candidate List). The local search phase then attempts to improve the solution by exploring its neighborhood until a local optimum is achieved.

  The key to GRASP's effectiveness lies in its combination of greedy and random construction of solutions, providing a balance between exploration (diversity) and exploitation (intensity). Here is a brief overview of the steps involved in GRASP:

  1. **Initialization**: set parameters, including the number of iterations and the size of the Restricted Candidate List.
  2. **Iterative Process**:
     - Construction: build a randomized solution by greedily selecting components from a restricted list.
     - Local Search: apply a local search algorithm to improve the solution.
     - Update: if the new solution is better than the best found so far, update the best solution.

- **Genetic Algorithms (GA)**: they are a class of population based optimization algorithms inspired by the process of natural selection, in particular simulating the evolutionary process.

  The key components of a Genetic Algorithm are as follows:

  - **Population**: a GA operates on a population of candidate solutions, each of which is encoded as a string, or *chromosome*, that represents a potential solution to the problem.

  - **Chromosome**: each individual in the population is referred to as a chromosome. It represents the encoding of a candidate solution using a data structure that is problem-specific. For example, in the case of the TSP, a chromosome could be a permutation of cities.
  - **Fitness Function**: it evaluates the quality of each chromosome. For optimization problems such as TSP, the fitness function could be the total distance of the tour. The fitness function serves as the driving force for evolution, promoting better solutions over worse ones.
  - **Selection**: individuals are selected from the population for reproduction based on their fitness. Chromosomes with higher fitness have a greater probability of being selected, if the principle of *survival of the fittest* is followed. Various selection methods can be used, such as *roulette wheel selection*, *tournament selection*, or *rank-based selection.*

- **Ant Colony Optimization (ACO)**: it is inspired by the foraging behavior of ants and is effective in solving combinatorial optimization problems. Ants communicate via pheromones to indirectly coordinate their actions, a mechanism that ACO simulates, for example, to find good paths through graphs. It is particularly effective for problems where the solution can be constructed step by step and can benefit from a collective learning process, as done by ants in nature. This algorithm has proven successful in different applications, including the Traveling Salesman Problem, scheduling, and network routing problems.

  The ACO algorithm involves several key components and steps:

  1. **Initialization**: initialize pheromone levels on all edges, typically to a small positive value. Set parameters such as the number of ants, number of iterations, pheromone evaporation rate ρ, and pheromone deposit coefficient τ.
  2. **Construct Ant Solutions**: each ant constructs a complete solution based on the pheromone trails and a heuristic value (e.g. inverse of the distance in the Traveling Salesman Problem).
     - The probability p_{ij}^k that ant k moves from city i to city j is given by:

       $$p_{ij}^k = \frac{\tau_{ij}^\alpha \cdot \eta_{ij}^\beta}{\sum_{l \in \mathrm{A}_k} \tau_{il}^\alpha \cdot \eta_{il}^\beta}$$

       where τ_{ij} is the amount of pheromone on edge (i, j), η_{ij} is the heuristic value (e.g., $\frac{1}{\text{distance}_{ij}}$), α and β control the influence of pheromone and heuristic value respectively, and A_k is the set of cities yet to be visited by ant k.
  3. **Update Pheromones**:
     - After all ants have completed their tours, update the pheromones on all edges. This involves evaporating existing pheromones and depositing new

pheromones based on the quality of the tours found:

$$\tau_{ij} = (1-\rho)\cdot\tau_{ij} + \sum_{k=1}^{m}\Delta\tau_{ij}^{k}$$

where ρ is the pheromone evaporation rate, m is the number of ants, and $\Delta\tau_{ij}^{k}$ is the amount of pheromone deposited by ant k, typically given by:

$$\Delta\tau_{ij}^{k} = \frac{Q}{L_k}$$

if ant k uses edge (i,j) in its tour, and 0 otherwise. Here, Q is a constant, and L_k is the length of the tour of ant k.

4. **Termination**: the algorithm terminates when a stopping criterion is met, which could be a set number of iterations, or a satisfactory solution quality.

1.4.4 Approximation Algorithms

Approximation algorithms come with formal performance guarantees. These algorithms are designed to provide solutions that are provably close to the optimal solution, usually quantified using an **approximation ratio**. In particular, it quantifies how far the solution provided by the algorithm is from the optimal solution. Formally, let I represent an instance of an optimization problem, and let $OPT(I)$ be the value of the optimal solution. If $H(I)$ is the value of the solution provided by an approximation algorithm H, the *approximation ratio* ρ is defined as:

$$\rho = \max_{I}\left(\frac{H(I)}{OPT(I)}\right) \quad \text{for minimization problems.}$$

An approximation algorithm with an approximation ratio ρ guarantees that the solution will be at most ρ times worse than the optimal solution. If $\rho = 1$, the algorithm is said to be *optimal.* In general, a lower approximation ratio indicates a solution that is closer to the optimum.

1.5 The Concepts Behind the Traveling Salesman Problem (*TSP*)

The *Traveling Salesman Problem* (*TSP*, see [2]) is one of the most famous and studied problems in combinatorial optimization. The *TSP* asks: *given a set of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the starting point?* This simple question has a lot of implications for optimization, computational complexity, and many practical applications in logistics and transportation (optimizing delivery routes for trucks, planes, or drones), routing, and also scheduling. These real-world applications highlight the importance of solving or well-approximating *TSP*, as it directly impacts cost, time, and resource management.

Given n cities, there are $(n-1)!$ possible tours, which makes the problem grow exponentially in size. This rapid growth in complexity places *TSP* in the class of **NP-hard** problems, meaning that there is no known polynomial-time algorithm to solve all instances of TSP efficiently. Consequently, *TSP* represents a crucial challenge in optimization theory, serving as a benchmark for exact and heuristic algorithms.

1.5.1 Multiple Traveling Salesman Problem (*mTSP*)

A generalization of the classical *TSP* is the *Multiple Traveling Salesman Problem (mTSP)*, where multiple salesmen are involved, instead of only one. The *mTSP* can be described as follows:

- A common depot serves as the starting and ending point for all salesmen.
- Each salesman is assigned a subset of cities, and each city must be visited exactly once by one of the salesmen.
- The goal is to minimize an objective function such as the total distance traveled or, in the case of *Min-Max mTSP*, the longest distance traveled by any salesman.

The *mTSP* is similarly **NP-hard** and inherits the complexity of the classical TSP. It is relevant in contexts where multiple agents, such as delivery trucks or service vehicles, are deployed from a central location to serve multiple destinations.

Challenges and Complexity

Both *TSP* and *mTSP* are challenging due to the exponential growth of possible solutions with the number of cities or salesmen, making the exact algorithms infeasible for large instances.
For these reasons, heuristic and approximation methods, such as **local search**, and **tabu search** and others are often used to find near-optimal solutions in a reasonable amount of time.

Chapter 2

Variants of the Traveling Salesman Problem

2.1 Traveling Salesman Problem (*TSP*)

The *Traveling Salesman Problem (TSP)* can be formulated as follows: given a set of cities and the distances between each pair of cities, the goal is to determine the shortest possible route that visits each city exactly once and returns to the starting city. This route is known as a *Hamiltonian cycle.*

Formally [17], *TSP* is defined on a complete weighted graph $G = (V, E)$, where:

- V is the set of vertices, each vertex representing a city or a destination to be visited by the salesman;

- E is the set of edges, each edge corresponding to a direct route between two cities (destinations);

- c_{ij} is the weight (or cost) assigned to each edge (i, j), representing the travel cost or distance between city i and city j. These costs can be represented as a cost matrix $C = \{c_{ij}\}$, where each element c_{ij} denotes the distance between cities i and j.

The goal is to find a permutation σ of the cities that minimizes the total travel distance. The mathematical (integer) programming formulation of the *TSP* is presented below:

Mathematical Model for the *TSP*

Objective Function:

$$\min \sum_{i=1}^{n} \sum_{j=1,\, j\neq i}^{n} d_{ij}\, x_{ij}$$

Subject to:

$$\sum_{j=1,\, j\neq i}^{n} x_{ij} = 1 \qquad \forall i = 1, 2, \ldots, n \tag{2.1}$$

$$\sum_{i=1,\, i\neq j}^{n} x_{ij} = 1 \qquad \forall j = 1, 2, \ldots, n \tag{2.2}$$

$$\sum_{i\in S} \sum_{j\in S,\, j\neq i} x_{ij} \leq |S| - 1 \qquad \forall S \subset V,\, 2 \leq |S| \leq n-1 \tag{2.3}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j = 1, 2, \ldots, n,\, i \neq j \tag{2.4}$$

Here is an explanation of the model:

- **Decision Variables:** $x_{ij} \in \{0,1\}$ is a binary decision variable, indicating whether the salesman travels directly from city i to city j. Specifically:
  - $x_{ij} = 1$ if the route from city i to city j is part of the tour.
  - $x_{ij} = 0$ if the route is not included in the tour.
- **Objective Function**: minimize the total travel distance, where d_{ij} represents the distance (or cost) between city i and city j.
- **Constraint 1**: exactly one outgoing edge is selected for each city, i.e., the salesman departs from each city exactly once.
- **Constraint 2**: exactly one incoming edge is selected for each city, i.e., the salesman enters each city exactly once.
- **Constraint 3**: the **subtour elimination constraint**. This is crucial to make sure that only a single Hamiltonian cycle is formed.
- **Constraint 4**: binary constraint on the decision variables x_{ij}, specifying that the route between city i and city j is either included in the solution or not.

TSP is classified as an **NP-hard** problem, meaning that there is no known algorithm capable of solving all instances of *TSP* in polynomial time. As the number of cities n increases, the number of possible routes grows factorially ($n!$), due to the combinatorial nature of the problem, making brute-force approaches computationally impractical for large-scale instances.

To get a near-optimal solution fast, a greedy algorithm can be used, that just goes to the nearest unvisited city in each step. Solving the TSP in that way means that, at each step, the distances from the current city to all other unvisited cities are compared: correspondingly an $O(n^2)$ complexity holds for that approach.

But finding the shortest route among all possible routes requires a lot more operations, and the time complexity for that is $O(n!)$. Which means that, in the case of 4 cities, there are 4! possible routes, which is the same as $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. And, for just 12 cities for example, there are $12! = 12 \cdot 11 \cdot 10... \cdot 2 \cdot 1 = 479001600$ possible routes!

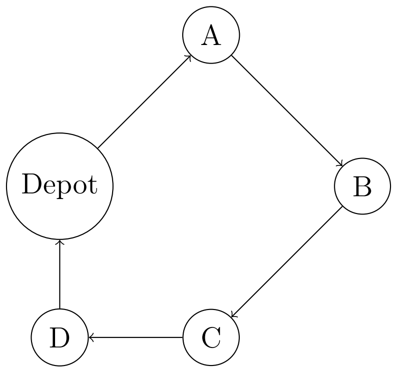


Figure 2.1: Graph Representation of a TSP

In Figure 2.1, an example of the TSP with one salesman is illustrated. It is an Hamiltonian tour, starting and ending at the depot.

2.2 Multiple Traveling Salesman Problem (*mTSP*)

The *Multiple Traveling Salesman Problem (mTSP)* is a generalization of the classical *TSP*. Instead of a single salesman visiting all cities, the problem involves multiple salesmen. Each of them must visit a subset of cities, starting and ending at a common depot (or home city).

Formally [9], *mTSP* can be defined on a complete weighted graph $G = (V, E)$ as before for the *TSP*, with a cost matrix $C = \{c_{ij}\}$ associated to each pair of nodes.
Now there are m salesmen available to cover the set of cities and the objective of the *mTSP* is to find m distinct tours, one for each salesman, such that:

- Each city is visited exactly once by one of the salesmen;
- each salesman starts and ends at the depot d;
- the objective is optimized according to a specific criterion.

One common objective is to minimize the *total distance* traveled by all salesmen, but other objectives, such as balancing the workload among salesmen, may also be considered. Similar to *TSP*, *mTSP* is also an NP-hard problem, and exact solutions become computationally impractical for large-scale instances. As a result, heuristic and metaheuristic methods are frequently used.

In Figure 2.2, an instance of the mTSP with two salesmen is illustrate. Each salesman is assigned a distinct subset of cities, starting and ending at the depot.

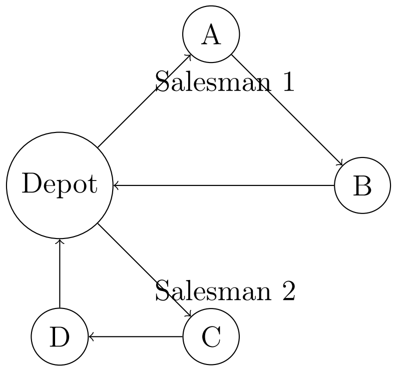


Figure 2.2: Graph Representation of a 2-Salesman mTSP

2.3 Min-Max Multiple Traveling Salesman Problem (Min-Max mTSP)

Min-Max optimization problems focus on minimizing the maximum value of a set of objective functions or decisions. They are particularly useful in scenarios where the worst-case performance must be optimized.

Definition: Min-Max Optimization

Let $f(x, y)$ be a bivariate objective function that depends on two variables $x \in \mathcal{X}$ and $y \in \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are feasible regions.
The goal is to find a decision $x^* \in \mathcal{X}$ that minimizes the *worst-case value* of the objective function, that is, the maximum value of $f(x^*, y)$ over all possible $y \in \mathcal{Y}$.
Formally, the problem can be written as:

$$\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} f(x, y)$$

A specific variant of the mTSP is the *Min-Max Multiple Traveling Salesman Problem (Min-Max mTSP)*. In this version, the objective is to minimize the maximum length tour among the m salesmen, rather than the sum of total distance traveled.

This problem is particularly relevant in cases where it is important to balance the workload among multiple agents, to make sure that no salesman has a disproportionately long route compared to the others. They are also used in work management models to optimize resources and fuel consumption in delivery by various operators.

Formally, let T_i denote the tour assigned to salesman i, and let the length of the tour be defined as the total travel cost of the cities visited by salesman i. The objective function could be expressed as:

$$\min \left(\max_{i=1}^{m} \text{length}(T_i) \right)$$

The challenge here is to distribute the cities among the salesmen in such a way that the workload is balanced and that no salesman is assigned an excessively long tour. Like *TSP* and *mTSP*, *min max mTSP* is NP-hard and requires heuristic and metaheuristic approaches to find near-optimal solutions.

Chapter 3

Related Work

3.1 Benchmark Datasets

Before starting with the various approaches and algorithms developed over time for this problem, it is important to mention the datasets defined in the literature and taken into account to evaluate the different solutions. They will also be used in this thesis to evaluate the solutions and make a comparison with other state of the art algorithms.

3.1.1 Dataset I

A total of eight instances are included in this set. It consists of a pair of *TSP* instances: berlin52 (52 cities, included depot) and eil76 (76 cities) from TSPLIB (1991) [1] that are solved with m = 2, 3, 5 and 7 salesmen for a total of 8 instances.

3.1.2 Dataset II

A widely studied set of instances defined by Carter and Ragsdale (2006) [5], which includes three Euclidean two-dimensional TSP instances with 51, 100, and 150 cities. Solving each instance for different numbers of salesmen (from m = 3, 5, 10 and 20) gives us 11 instances in total. In particular, MTSP-51 is solved with m = 3, 5 and 10 salesmen, MTSP- 100 and MTSP-150 are solved with m = 3, 5, 10 and 20 salesmen.

3.1.3 Dataset III

A set of 20 instances defined by Wang et al. (2017) [14] that consists of five larger TSP instances: rand100, ch150, kroA200, lin318 and rat783 from TSPLIB. In this set, m = 3, 5, 10 and 20 salesmen are used.

3.2 Different Tour Representations Proposed for *mTSP*

For *mTSP*, another challenge is to represent multiple salesmen, each with their own route, in a way that allows the algorithm to manipulate and evolve the solution efficiently.

In the literature several different encodings are proposed for *mTSP*. These representations differ in how they encode the sequence of cities visited by each salesman and how they allow for operations like shifts and swaps between nodes. In fact, for each variant, the operations' formulation carried out is different.

1. **Delimiter-based Encoding (Tang et al., 2000)**: cities are listed sequentially, and delimiters (such as the value "-1") are used to separate the cities assigned to different salesmen, for example:

4	1	3	5	-1	6	2	10	-1	8	9	7

   in this case, the sequence of cities is assigned to salesmen based on where the delimiter occurs. Each substring between two delimiters represents a separate salesman tour.

2. **Two-Way Encoding (Park, 2001)** [3]: this encoding uses two arrays, the first one encodes the sequence of cities and the second represents the salesman assigned to each city. For example: array 1 (cities):

6	4	1	8	2	3	9	5	7	10

   array 2 (salesman):

2	1	1	3	2	1	3	1	3	2

   here, city 6 is visited by salesman 2, city 4 is visited by salesman 1, and so on.

   - Advantages: allows flexibility in city assignment.
   - Disadvantages: complexity increases with the need to handle two separate structures.

3. **General Grouping Encoding (Brown et al., 2007)** [6]: the array carries information about which salesman is assigned to each city and the order in which the salesman visits the cities. Each slot represents a city, and its value encodes both the salesman and the position of the city in the route. For example:

1.2	2.2	1.3	1.1	1.4	2.1	3.3	3.1	3.2	2.3

   here, 1.2 in position one means that city 1 is visited by salesman 2 and is the second city on the salesmanâs route. 2.2 in position two means that salesperson two visits city two, that is the second on the route and so on.

4. **Simple Tour Representation (Mahmoudinazlou and Kwon, 2024)** [19]: the authors propose a simpler representation to address the challenges posed by the previous methods. The encoding used here is a single array that represents the sequence of all cities (a simple list of cities, without any delimiters or extra information). Here each salesman has a separate array that encodes the sequence of cities visited. An example is:

4	6	1	9	2	3	8	5	7	10

   which is a sequence of all cities.

3.3 Different Approaches Proposed for *mTSP*

3.3.1 Genetic Algorithms for *mTSP*

1. An early work by Tang et al. (2000) introduced a chromosome encoding scheme for *mTSP* as part of a rolling scheduling system. Similarly, Park (2001) [3] proposed a two-chromosome GA for the Vehicle Routing Problem with Time Windows (VRPTW), incorporating partially mapped crossover (PMX) and various mutation functions.

2. Carter and Ragsdale (2006) [5] and Chen and Chen (2011) [11] developed two-part chromosome representations for encoding mTSP solutions, focusing on evaluating different genetic operators for performance improvement. Brown et al. (2007) [6] and Singh and Baghel (2009) [8] further explored novel encoding techniques, such as the Grouping Genetic Algorithm (GGA) and many-chromosome representations, to optimize mTSP solutions.

3. Innovative hybrid approaches have also been proposed. Wang et al. (2017) [14] introduced a Memetic Algorithm with Sequential Variable Neighborhood Descent (MASVND) for the Min-Max mTSP, demonstrating significant improvements in solution quality. S. Mahmoudinazlou and C. Kwon (2024) [19] proposed a hybrid genetic algorithm (HGA) to solve the Min-Max mTSP. The genetic algorithm uses a unique *TSP* sequence to represent each individual and a dynamic programming algorithm is employed to evaluate the individual subtours and find the optimal mTSP solution for the given sequence of cities.

3.3.2 Other Heuristic for *mTSP*

1. Junjie and Dingwei (2006) [4] and Liu et al. (2009) [7] applied Ant Colony Optimization (ACO) algorithms to solve different variations of *mTSP*, including multi-objective versions. Metaheuristic algorithms such as Artificial Bee Colony (ABC) and Invasive Weed Optimization (IWO), introduced by Venkatesh and Singh (2015) [12], have also been combined with Local Search techniques to improve performance.

2. Soylu (2015) [13] proposed a General Variable Neighborhood Search (GVNS) for mTSP, employing multiple local search functions to explore the solution space efficiently. More recent approaches, such as the Hybrid Search with Neighborhood Reduction (HSNR) algorithm by He and Hao (2022) [16], utilize Tabu Search and TSP-specific heuristics to optimize *mTSP* solutions.

3. Zheng et al. (2022) [15] introduced an Iterated Two-Stage Heuristic Algorithm (ITSHA), which generates initial solutions using clustering techniques and then refines them through variable neighborhood searches. Their approach has yielded some of the best-known results on benchmark datasets.

Chapter 4

Methodology

This chapter presents the methodology for solving different instances of the Multiple Traveling Salesman Problem (*mTSP*). At first, the structure of the algorithm relied on clustering to assign nodes to subtours (initialization), as developed in the literature by Zheng et al. (2022) [15].

Then a new strategy was tried. In particular, following the intuition of Sasan Mahmoudinazlou and Changhyun Kwon (2024) [19] about the importance of removing the intersection in order to obtain optimal subtours, an algorithm has been developed that can divide the points into sectors based on their spatial and angular position, so as to obtain a strong initialization of the tours, but which at the same time allows an adequate exploration of the neighborhood. This and other developments will be discussed specifically.

After initialization, other algorithms iteratively explore neighborhoods to refine the solution, using a heuristic transfer strategy and a *TSP* exact solver for optimal tour calculations. It is made up of a mix of node movements between the tours of the various salesmen.

In all these cases, a common algorithm is used to solve the *TSP* related to the subtours, and it will be presented in this section.

4.1 Solving *TSP* for Each Salesman

In this section, we describe the mathematical formulation and computational approach used to solve the Traveling Salesman Problem (*TSP*) for the nodes within each salesman's tour.

The problem is tackled using the Google OR-Tools framework [18], which implements advanced algorithms to solve combinatorial optimization problems. The goal is to determine an optimal tour for each sector, that minimizes the total distance traveled while returning to the depot.

4.1.1 Formulation of *TSP*

As seen before, *TSP* can be defined as follows: given a set of N nodes (locations) and a distance matrix $\mathbf{D} = [d_{ij}]$, where d_{ij} represents the distance between node i and node j, find a permutation σ of the nodes such that the total distance traveled is minimized. Mathematically, the problem can be formulated as:

$$\min_{\sigma \in S_N} \sum_{i=1}^{N} d_{\sigma(i),\sigma(i+1)},$$

where S_N is the set of all possible permutations of the N nodes, and $\sigma(N+1) = \sigma(1)$ ensures that the tour returns to the starting node (depot).

4.1.2 Distance Matrix Computation

To solve *TSP*, a key input is the distance matrix $\mathbf{D}$, which encodes the pairwise distances between all locations, including the depot. Let the coordinates of the depot be $\mathbf{D} = (x_d, y_d)$, and the coordinates of the i-th node be $\mathbf{P}_i = (x_i, y_i)$. The distance d_{ij} between any two points $\mathbf{P}_i$ and $\mathbf{P}_j$ is computed using the Euclidean distance metric:

$$d_{ij} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}.$$

The distance matrix is constructed as:

$$\mathbf{D} = \begin{bmatrix} 0 & d_{01} & \dots & d_{0N} \\ d_{10} & 0 & \dots & d_{1N} \\ \vdots & \vdots & \ddots & \vdots \\ d_{N0} & d_{N1} & \dots & 0 \end{bmatrix},$$

where $d_{ii} = 0$ for all i, as the distance from a point to itself is zero.

4.1.3 Routing Optimization Using OR-Tools

The OR-Tools library, developed by Google, is used to solve the *TSP* for each salesman. The key steps in the optimization process are as follows:

Model Construction

The problem is modeled using the following components:

- **Nodes:** the depot and the nodes in the current sector are represented as indices in the distance matrix.
- **Distance Callback:** a callback function is defined to retrieve the distance d_{ij} between any two nodes i and j from the distance matrix.
- **Objective Function:** the total cost of the tour is defined as the sum of the distances between consecutive nodes in the tour. OR-Tools minimizes this cost using built-in optimization methods.

Search Parameters

The optimization process is guided by the following search parameters:

- **First Solution Strategy:** a Cheapest Arc heuristic is used to generate an initial solution. This heuristic selects the arc (edge) with the smallest cost at each step, building a tour incrementally.
- **Local Search:** it applies local search techniques to improve the initial solution by exploring neighboring solutions in the solution space.

Solution Extraction

Once the optimization is complete, the solution consists of:

- **Tour:** an ordered sequence of node indices representing the optimal tour. The tour starts and ends at the depot.
- **Tour Length:** the total distance of the optimal tour, computed as the sum of the distances between consecutive nodes in the tour.

Mathematically, the tour is represented as:

$$\text{Tour} = \{\sigma(1), \sigma(2), \dots, \sigma(N), \sigma(1)\},$$

where $\sigma(1)$ represents the depot.

In conclusion, the use of OR-Tools ensures that high-quality solutions are obtained for each salesman, leveraging state-of-the-art optimization techniques.

4.2 Data Clustering and *mTSP*

As mentioned above, the first attempt involves clustering the cities into groups, with each cluster representing a separate subtour. Let N denote the set of cities to be visited and n the number of clusters (salesmen). Clustering is defined as:

Definition: Clustering

Clustering is a technique for partitioning a set of objects $X = \{x_1, x_2, \ldots, x_n\}$ into k groups (clusters) $C_1, C_2, \ldots, C_k$ so that the objects in each cluster are more similar to each other than to those in other clusters.

The objective could be to minimize the variance within the clusters, often formulated as

$$\min_{C_1, C_2, \ldots, C_k} \sum_{j=1}^{k} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2,$$

where μ_j is the centroid of the cluster C_j.

In this case, given a set of nodes N and a desired number of clusters n, each node $i \in N$ is assigned to a cluster C_k where $k = 1, \ldots, n$, such that each cluster forms a distinct subproblem for the TSP.

4.2.1 The k-Means Algorithm

The k-means algorithm is one of the most widely used clustering methods, aiming to partition a set of n data points $X = \{x_1, x_2, \ldots, x_n\}$ in $\mathbb{R}^d$ into k clusters $C_1, C_2, \ldots, C_k$. The algorithm iteratively assigns points to clusters and updates the cluster centroids to minimize the within-cluster variance.

Objective Function

The objective function of k-means is to minimize the sum of squared Euclidean distances between points and their respective cluster centroids μ_j:

$$\min_{C_1, C_2, \ldots, C_k} \sum_{j=1}^{k} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2.$$

This objective encourages compact clusters, with points close to the cluster centroid.

Algorithm Steps

k-means Algorithm

1. **Initialization**: Randomly select k points as the initial centroids $\mu_1, \mu_2, \ldots, \mu_k$.
2. **Assignment Step**: For each data point x_i, assign it to the nearest cluster C_j based on the centroid distance:
$$C_j = \{x_i : \|x_i - \mu_j\| \leq \|x_i - \mu_\ell\| \quad \forall\, \ell = 1, \ldots, k\}.$$
3. **Update Step**: Recalculate each centroid μ_j as the mean of all points in C_j:
$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i.$$
4. **Repeat**: Alternate between the assignment and update steps until convergence (i.e., no changes in assignments or minimal decrease in the objective function).

As we can see from the example below (3 operators, 51 cities), k-mean gives a starting point for our algorithm, based on the clusters found.

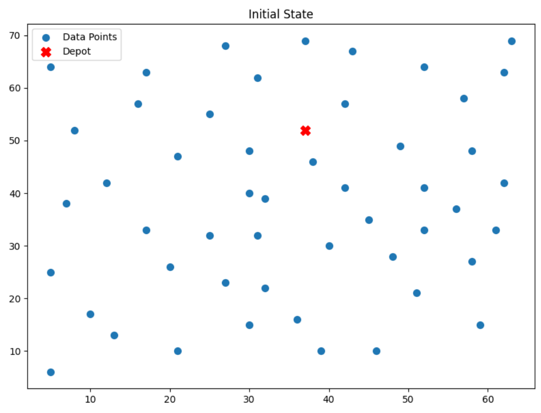


Figure 4.1: Initial State of Data Points and Depot

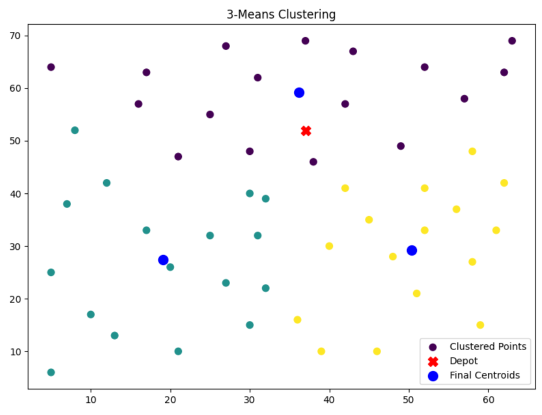


Figure 4.2: 3-Means Clustering with Final Centroids

We will now show benefits and critical issues of this approach, as well as why different paths have been tried.

4.2.2 Benefits and critical issues of the Clustering approach

Benefits of the Clustering Approach

Immediate advantages in the initial node separation (initial solution) for the Min-Max m-TSP:

1. **Proximity-Based Grouping**:
   Clustering methods, particularly those based on spatial metrics (like k-means), are effective in grouping closely located nodes together. This ensures that nodes within the same cluster are geographically near one another, which can reduce intra-cluster distances. Consequently, subtours formed within a cluster are likely to exhibit better local compactness.

2. **Capability to Identify Point Structures**:
   Clustering algorithms perform well in recognizing spatial structures, such as dense regions of points. In practice, clusters can be formed around cities that are part of a metropolitan area or a geographically isolated region. This capability aligns with the idea of creating simple and geographically intuitive subtours, which simplifies problem visualization and allows for more interpretable solutions.

Critical issues of the Clustering approach

Despite its advantages, this approach encounters several critical issues when applied to the Min-Max m-TSP. These limitations arise from the inherent characteristics of clustering algorithms:

1. **Ignoring Depot Proximity**:
   Clustering algorithms do not inherently consider the distance between the depot (the starting and end point for all salesmen) and the formed groups. This results in clusters that may be located arbitrarily far from the depot, leading to highly imbalanced subtour lengths.

   For example, a cluster positioned near the depot might result in a short tour, while a distant cluster would lead to a much longer one. This imbalance contradicts the goal of minimizing the maximum tour length and makes it difficult to achieve the optimum through genetic operators on the various subtours.

2. **Imbalanced Cluster Sizes**:
   These algorithms often produce clusters with significantly varying sizes. This is particularly problematic in the Min-Max m-TSP, as it leads to subtours with highly unequal numbers of nodes. A cluster with too many nodes can create a disproportionately long tour for one salesman, while a cluster with too few nodes results in underutilization of other salesmen. This undermines the principle of equity in the workload distribution across salesmen, which is crucial to solving the problem.

3. **Disregard for Inter-Cluster Intersection**:
   Clustering algorithms obviously operate independently of the interactions between

clusters and the tours they form. Specifically, the placement of clusters does not account for how the resulting subtours interact with one another, particularly in terms of overlaps or shared boundaries. This leads to the presence of many intersections between the subtours, which could increase a lot the overall maximum length.

4. **Distance Imbalance Across Tours**:
   Clustering algorithms do not optimize for the overall distribution of distances among subtours. While they may minimize intra-cluster distances, the inter-cluster distances and the resulting subtour lengths can vary widely and without control. This discrepancy may contradict the primary objective, which seeks to minimize the longest tour.

Conclusion: Limitations Outweigh Benefits

While the clustering approach provides an intuitive method for initial node separation and offers advantages such as local compactness and spatial structure recognition, its limitations significantly outweigh these benefits in the context of the Min-Max m-TSP. The neglect of depot proximity, imbalanced cluster sizes, disregard for inter-cluster interactions, and distance imbalances across subtours collectively hinder its effectiveness. These shortcomings lead to solutions that are suboptimal in terms of minimizing the maximum tour length.

Given the critical nature of these limitations, it is evident that alternative approaches must be explored to achieve more balanced and efficient initial solutions for the Min-Max m-TSP. Although clustering may provide a useful starting point for exploring the solution space, it is insufficient to address the complexities of this problem.

4.3 A New Proposal: Nodes Division into Angular Sectors

We now introduce a novel approach for partitioning nodes. The main idea is to divide them into different and separated angular sectors, providing an initialization strategy for solving Min-Max m-TSP.

The method dynamically allocates nodes into sectors based on their spatial orientation and position relative to a central reference point, in this case it is strategically the depot.

This strategy leverages geometric properties of the node distribution to organize the solution space effectively and provide a structured starting point for optimization.

4.3.1 Mathematical Framework

The core principle of the proposed method is the calculation of angular positions for all nodes relative to the depot. Let the coordinates of the depot be defined as:

$$\mathbf{D} = (x_d, y_d),$$

and the coordinates of a generic node $\mathbf{P}_i$ be:

$$\mathbf{P}_i = (x_i, y_i), \quad i = 1, 2, \ldots, N,$$

where N represents the total number of nodes in the dataset.

The angular position θ_i of node $\mathbf{P}_i$ with respect to the depot $\mathbf{D}$ is given by:

$$\theta_i = \arctan(y_i - y_d, x_i - x_d) \cdot \frac{180}{\pi},$$

where the arctan function is used to compute the angle in the standard Cartesian coordinate system. The output of arctan lies in the range $(-180°, 180°]$, which is then normalized to the interval $[0°, 360°)$ using the following transformation:

$$\theta_i = \begin{cases} \theta_i & \text{if } \theta_i \geq 0, \\ 360 + \theta_i & \text{if } \theta_i < 0. \end{cases}$$

This ensures that all angular measurements are positive and comparable within a single 360-degree framework.

4.3.2 Dynamic Allocation within an Angular Plane

To create an organized partitioning of the nodes, we define an angular plane limited by two boundaries: a minimum angle θ_{min} and a maximum angle θ_{max}. This allows for robustness in handling different instances (datasets) and cases (numbers of operators), with the most diverse point distributions compared to the depot.

The range of this angular plane is:

$$\Delta\theta = \theta_{\text{max}} - \theta_{\text{min}},$$

where $\Delta\theta$ represents the span of the plane in degrees. This plane is further subdivided into n_{sectors}. They are n equally sized angular regions or sectors, with each sector having an angular width of:

$$\Delta\theta_{\text{sector}} = \frac{\Delta\theta}{n_{\text{sectors}}}.$$

The boundaries of each sector are defined as:

$$\theta_k = \theta_{\text{min}} + k \cdot \Delta\theta_{\text{sector}}, \quad k = 0, 1, \ldots, n_{\text{sectors}},$$

where k denotes the sector index. Nodes whose angular positions θ_i lie within the bounds of a sector are assigned to that sector. Formally, the sector assignment for a node is determined as:

$$\text{Sector ID} = \max\{k : \theta_k \leq \theta_i < \theta_{k+1}\}.$$

It is important to handle boundary cases carefully. If a node's angle θ_i lies exactly on the upper boundary of a sector θ_{k+1}, it is assigned to the next sector, ensuring that no node remains unclassified.

4.3.3 Handling Points Outside the Angular Plane

Nodes with angular positions outside the defined plane $[\theta_{\min}, \theta_{\max}]$ require special handling. In particular, these nodes are assigned to the closest sector based on angular proximity.

To achieve this, we compute the centroid of each sector within the angular plane. The angular centroid of sector k is calculated as:

$$\bar{\theta}_k = \frac{\sum_{i \in \text{Sector } k} \theta_i}{|k|},$$

where $|k|$ represents the number of nodes assigned to sector k, and θ_i are the angular positions of nodes in that sector.

For each node outside the plane, its angular distance to each sector centroid is computed as:

$$\Delta\theta_{i,k} = \min\left(|\theta_i - \bar{\theta}_k|, 360 - |\theta_i - \bar{\theta}_k|\right),$$

which accounts for the circular nature of angular measurements. The node is assigned to the sector with the smallest angular distance:

$$\text{Assigned Sector} = \arg\min_k \Delta\theta_{i,k}.$$

4.3.4 Algorithm Description

The overall process for partitioning nodes into angular sectors can be summarized as follows:

Angular Allocation Algorithm Description

1. Compute the angular position θ_i of each node relative to the depot:

   $$\theta_i = \arctan(y_i - y_d, x_i - x_d) \cdot \frac{180}{\pi},$$

   .

2. Normalize all angular positions to the range $[0°, 360°)$.
3. Define the angular plane $[\theta_{\min}, \theta_{\max}]$ and divide it into n_{sectors} equal sectors.
4. Assign nodes within the angular plane to sectors based on their angular positions and the sector boundaries.
5. Compute the angular centroids of all sectors within the plane.
6. For nodes outside the angular plane, compute their angular distances to all sector centroids and assign them to the nearest sector.
7. Return the final sector assignments for all nodes.

4.3.5 Advantages of the Angular Sector Division Approach

The proposed method offers several advantages for initializing the Min-Max m-TSP solution:

- **Spatial Organization:** nodes are partitioned into geometrically meaningful sectors, which simplifies the initial solution space and reduces computational complexity in subsequent optimization phases.

- **Depot-Centric Division:** by using the depot as the reference point, the method ensures that the partitioning reflects the practical constraints of the problem, such as proximity to the starting point of each tour.

- **Possibility of Different Initializations:** by modifying the minimum and maximum angle parameters, various assignments in sectors (and therefore initializations) can be obtained. This allows to test both more promising solutions and to start from a worse conformation.

- **Scalability:** the method is computationally efficient and can handle large datasets due to its reliance on angular calculations and straightforward sector assignments.

- **Absence of Subtour Intersections:** by assigning nodes to distinct angular sectors, the method ensures that there are no intersections between arcs from different operators. In fact, each sector contains arcs between nodes that are spatially well grouped and non-overlapping with arcs in adjacent sectors subtours. This organization ensures that the initial solution is cleaner, simplifying the subsequent optimization process, as the initial solution is already well-structured and reflects a good layout of the problem space.

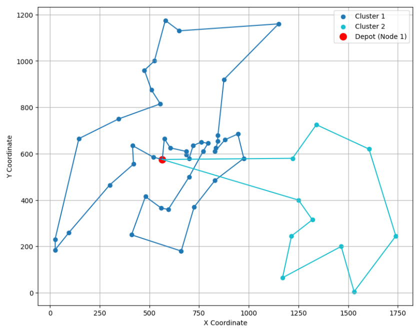
4.3.6 Preliminary testing: A Numerical and Graphical Comparison Between Clustering and Angular Sector Division

Below are reported preliminary results on the Angular Sector Division approach on the three considered datasets. Indeed, the approach provides a robust, structured and efficient method for initializing Min-Max m-TSP solutions. By leveraging the geometric properties of the nodes distribution, this strategy ensures an organized partitioning of the problem space. As seen in the comparison with the initialization via k-means (which does not take into account the position of the points with respect to the depot), Angular Sector algorithm obtain already limited gaps with respect to the best known solution available in literature (BKS) in every instance, and correspondingly represents a strong starting point for optimization algorithms. Also, the best results reached between the two initialization algorithms are highlighted.

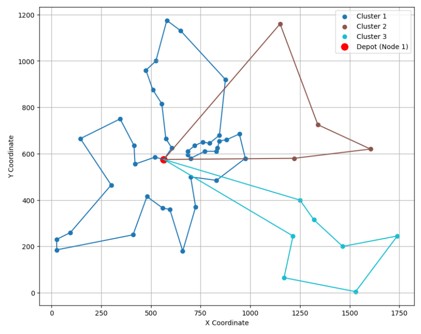
Initialization results on Dataset I instances

Instance	m	BKS	Clustering	Angular	Gap (BKS-Cl)	Gap (BKS-Ang)
berlin52	2	4110.2	5471.0	**4201.0**	33.11 %	**2.21** %
berlin52	3	3191.0	4689.0	**3222.0**	46.94 %	**0.97** %
berlin52	5	2441.4	4689.0	**2662.0**	92.06 %	**9.04** %
berlin52	7	2440.9	3495.0	**2440.0**	43.18 %	**-0.04** %
eil76	2	280.9	308.0	**290.0**	9.65 %	**3.24** %
eil76	3	197.3	223.0	**200.0**	13.03 %	**1.37** %
eil76	5	143.4	163.0	**154.0**	13.67 %	**7.39** %
eil76	7	127.6	**148.0**	154.0	**15.99** %	20.69 %

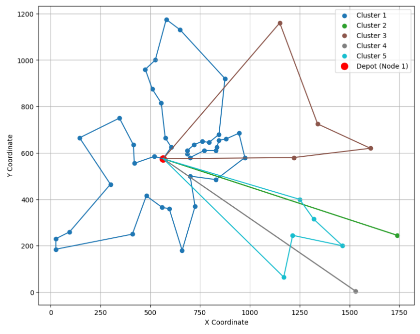
Table 4.1: Clustering-Angular Initialization Comparison on Dataset I



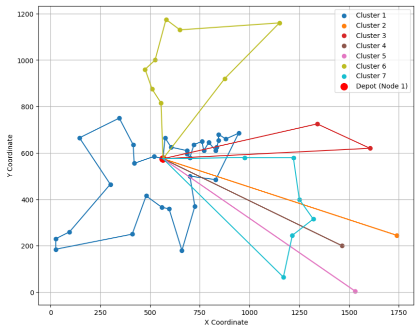
(a) Berlin52, m=2, Cl



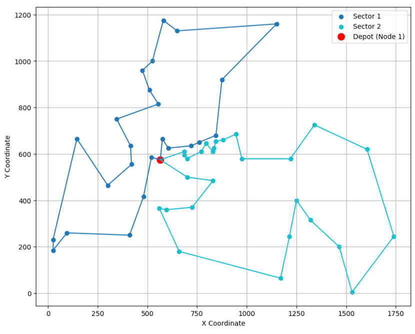
(b) Berlin52, m=3, Cl



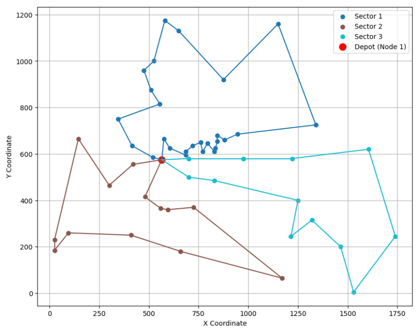
(c) Berlin52, m=5, Cl



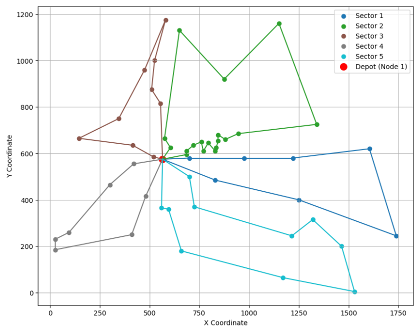
(d) Berlin52, m=7, Cl



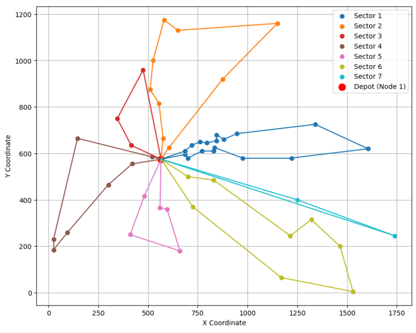
(a) Berlin52, m=2, Ang



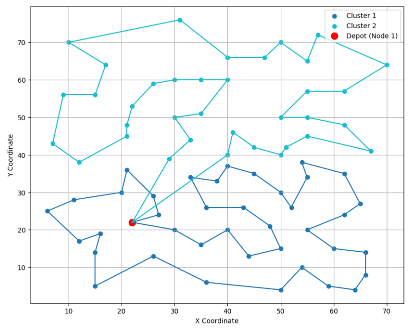
(b) Berlin52, m=3, Ang



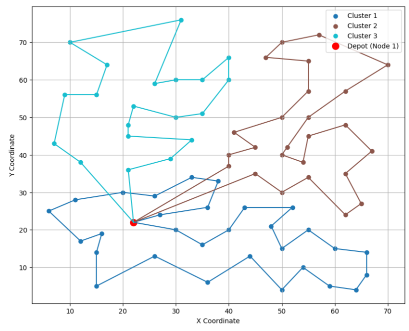
(c) Berlin52, m=5, Ang



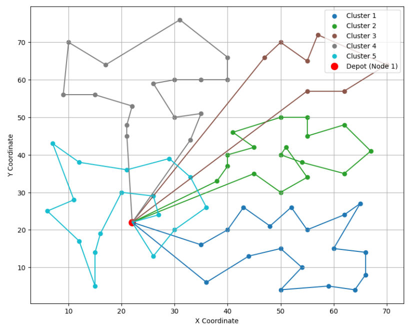
(d) Berlin52, m=7, Ang



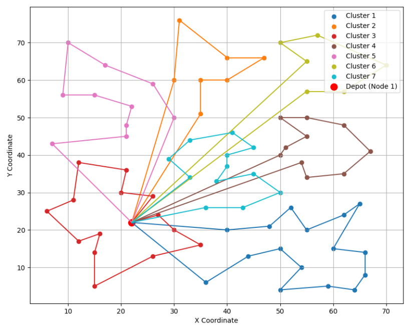
(a) eil76, m=2, Cl



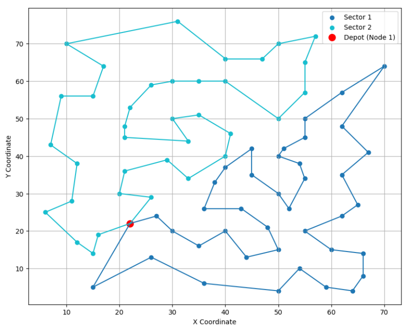
(b) eil76, m=3, Cl



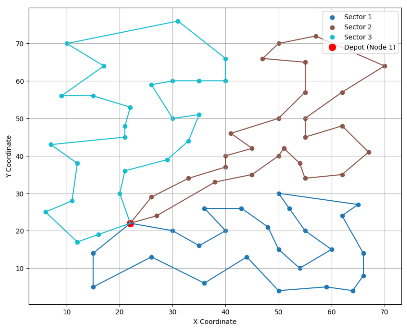
(c) eil76, m=5, Cl



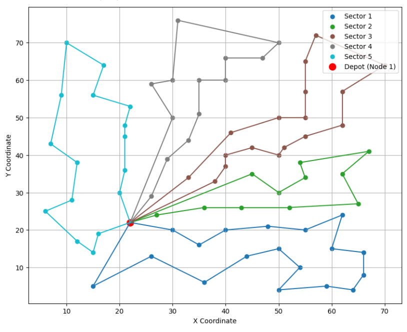
(d) eil76, m=7, Cl



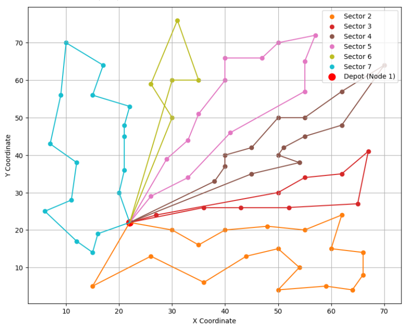
(a) eil76, m=2, Ang



(b) eil76, m=3, Ang



(c) eil76, m=5, Ang



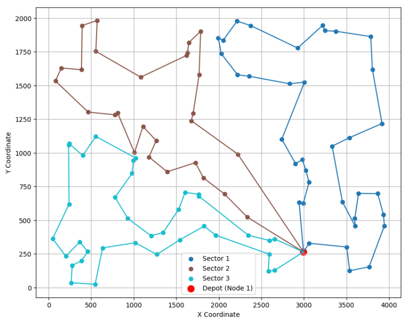
(d) eil76, m=7, Ang

Initialization results on Dataset II instances

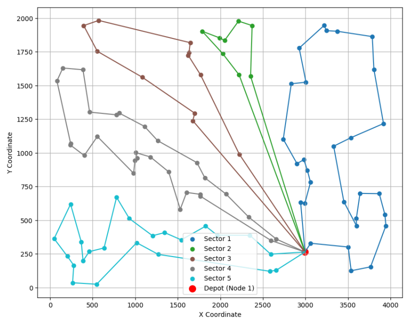
Instance	m	BKS	Clustering	Angular	Gap (BKS-Cl)	Gap (BKS-Ang)
mtsp51	3	159.0	**175.0**	183.0	**10.06** %	15.09 %
mtsp51	5	118.0	**132.0**	136.0	**11.86** %	15.25 %
mtsp51	10	112.0	**112.0**	**112.0**	**0.00** %	**0.00** %
mtsp100	3	8507.0	11161.0	**8648.0**	31.20 %	**1.66** %
mtsp100	5	6766.0	7584	**7371.0**	12.09 %	**8.94** %
mtsp100	10	6358.0	6842.0	**6602.0**	7.61 %	**3.84** %
mtsp100	20	6358.0	6878.0	**6379.0**	8.18 %	**0.33** %
mtsp150	3	13039.0	18260.0	**13675.0**	40.04 %	**4.88** %
mtsp150	5	8416.0	15148.0	**9968.0**	79.99 %	**18.44** %
mtsp150	10	5557.0	8761.0	**7082.0**	57.66 %	**27.44** %
mtsp150	20	5246.0	6470.0	**5606.0**	23.33 %	**6.86** %

Table 4.2: Clustering-Angular Initialization Comparison on Dataset II

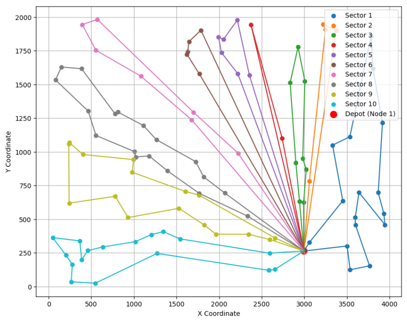
Salesman division obtained via Angular Initialization in Dataset II:



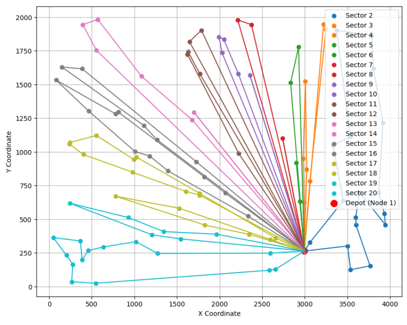
(a) mtsp100, m=3, Ang



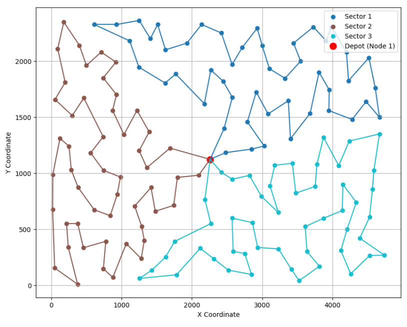
(b) mtsp100, m=5, Ang



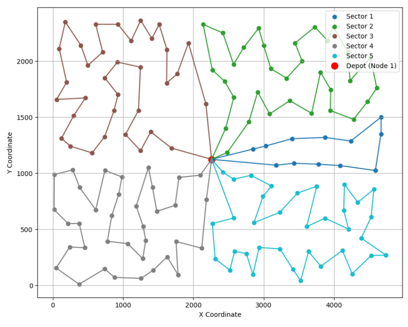
(c) mtsp100, m=10, Ang



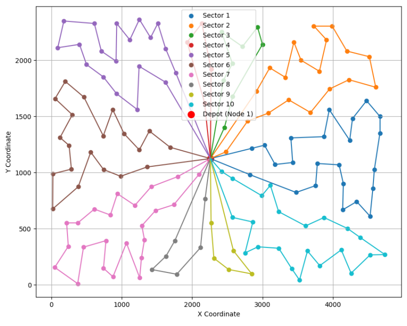
(d) mtsp100, m=20, Ang



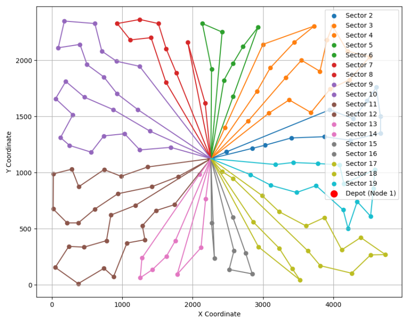
(a) mtsp150, m=3, Ang



(b) mtsp150, m=5, Ang



(c) mtsp150, m=10, Ang



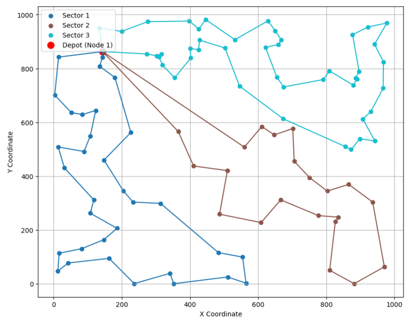
(d) mtsp150, m=20, Ang

Initialization results on Dataset III instances

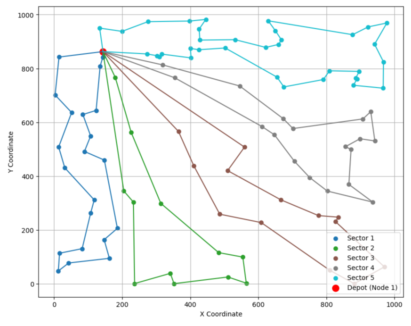
Instance	m	BKS	Clustering	Angular	Gap (BKS-Cl)	Gap (BKS-Ang)
rand100	3	3031.95	4203.0	**3162.0**	38.62 %	**4.29** %
rand100	5	2409.63	2678.0	**2545.0**	11.14 %	**5.60** %
rand100	10	2299.16	2391.0	**2322.0**	3.99 %	**0.96** %
rand100	20	2299.16	**2298.0**	2299.0	**-0.05** %	-0.01 %
ch150	3	2401.63	3101.0	**2477.0**	29.12 %	**3.14** %
ch150	5	1740.63	2274.0	**1805.0**	30.64 %	**3.70** %
ch150	10	1554.64	1705.0	**1690.0**	9.67 %	**8.71** %
ch150	20	1554.64	**1554.0**	1690.0	**-0.04** %	8.71 %
kroa200	3	10691.0	14246.0	**11048.0**	33.25 %	**3.34** %
kroa200	5	7412.12	12361	**9239.0**	66.77 %	**24.63** %
kroa200	10	6223.22	**6453.0**	6724.0	**3.69** %	8.03 %
kroa200	20	6223.22	7429.0	**6279.0**	19.38 %	**0.88** %
lin318	3	15663.5	18328.0	**16974.0**	17.01 %	**8.37** %
lin318	5	11276.8	15475.0	**12688.0**	37.23 %	**12.60** %
lin318	10	9731.17	10947.0	**10433.0**	12.49 %	**7.21** %
lin318	20	9731.17	**9740.0**	9843.0	**0.09** %	1.15 %
rat783	3	3052.41	4278.0	**3194.0**	40.15 %	**4.63** %
rat783	5	1961.12	2524.0	**2225.0**	28.70 %	**13.40** %
rat783	10	1313.01	1664.0	**1574**	26.73 %	**19.88** %
rat783	20	1231.69	1337.0	**1263.0**	8.55 %	**2.54** %

Table 4.3: Clustering-Angular Initialization Comparison on Dataset III

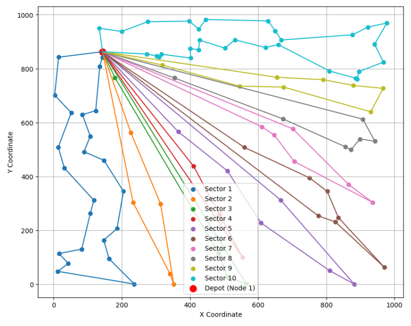
Salesman division obtained via Angular Initialization in Dataset III:



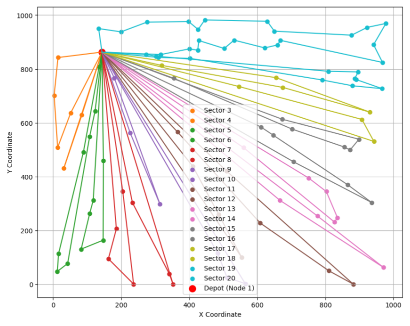
(a) rand100, m=3, Ang



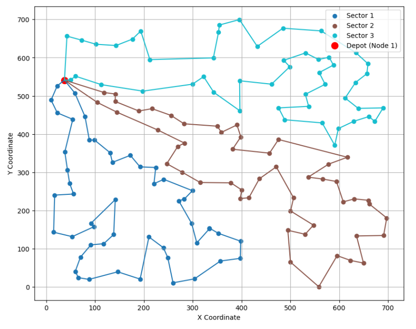
(b) rand100, m=5, Ang



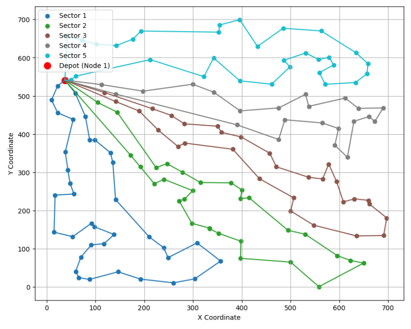
(c) rand100, m=10, Ang



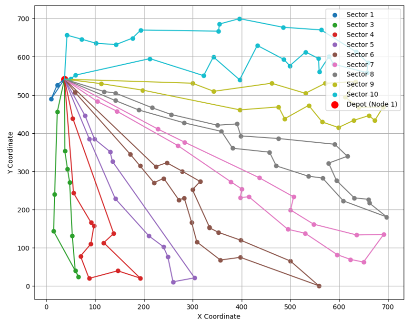
(d) rand100, m=20, Ang



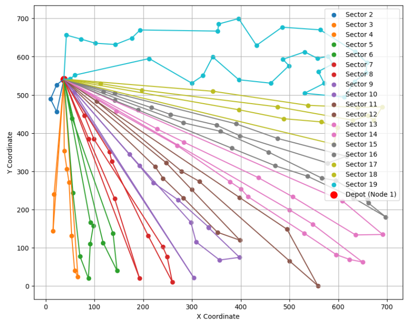
(a) ch150, m=3, Ang



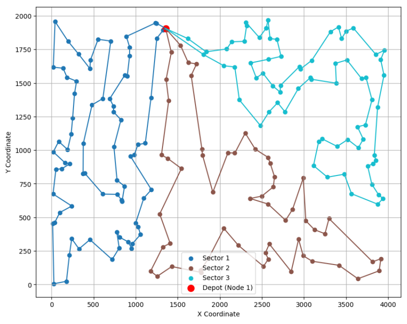
(b) ch150, m=5, Ang



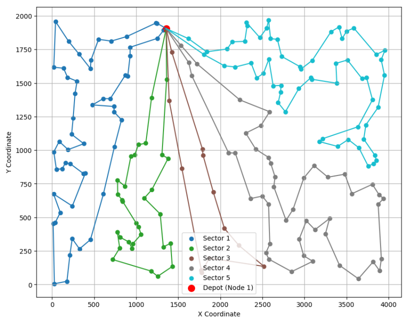
(c) ch150, m=10, Ang



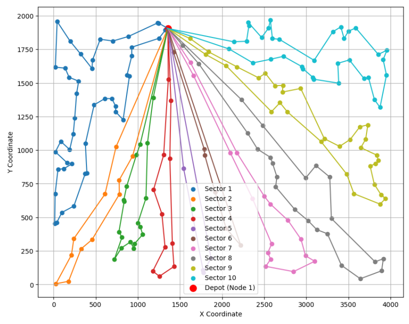
(d) ch150, m=20, Ang



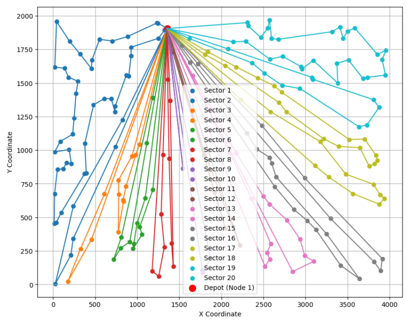
(a) kroa200, m=3, Ang



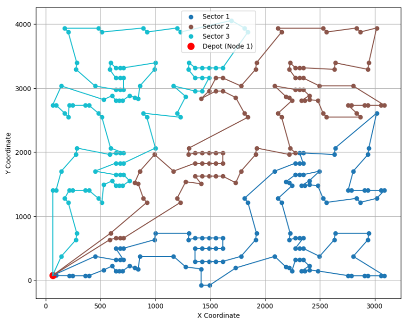
(b) kroa200, m=5, Ang



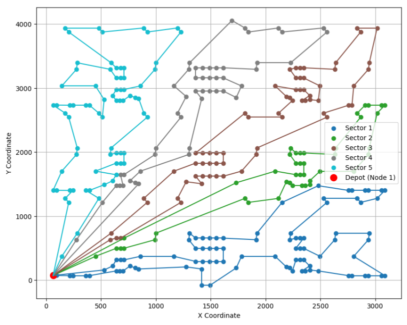
(c) kroa200, m=10, Ang



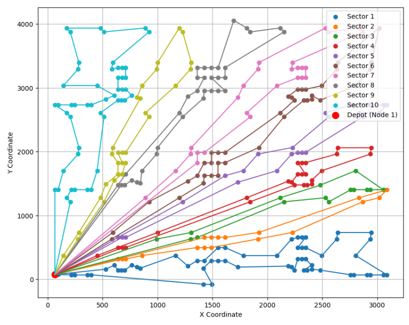
(d) kroa200, m=20, Ang



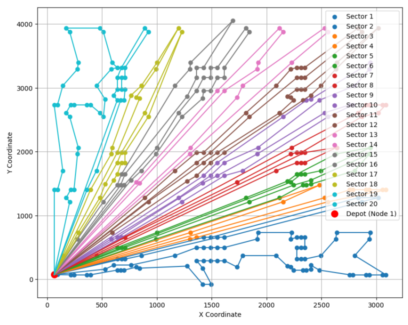
(a) lin318, m=3, Ang



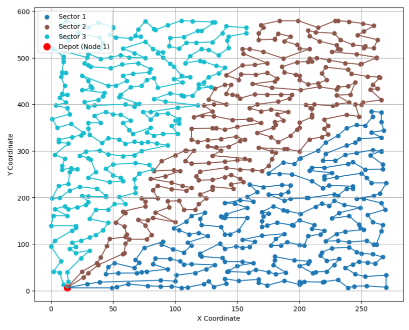
(b) lin318, m=5, Ang



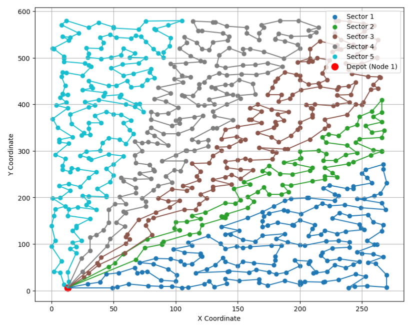
(c) lin318, m=10, Ang



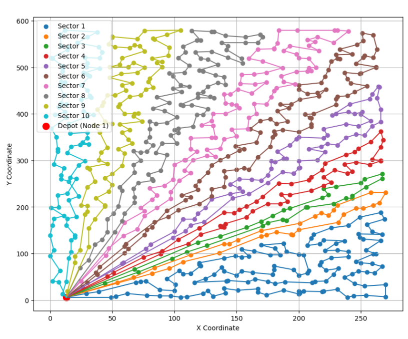
(d) lin318, m=20, Ang



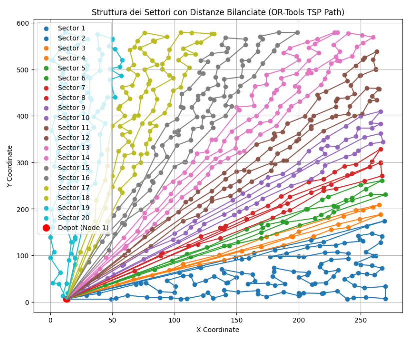
(a) rat783, m=3, Ang



(b) rat783, m=5, Ang



(c) rat783, m=10, Ang



(d) rat783, m=20, Ang

Chapter 5

Developed Algorithms

This chapter presents the three optimization algorithms developed to address the Min-Max m-TSP. Each algorithm focuses on redistributing nodes between subtours to balance workloads and minimize the length of the longest subtour. Although the fundamental goal is shared, the strategies for node selection, transfer, and optimization differ significantly among the approaches and this is also a reason of why they perform well combined.

In particular,the first algorithm employs a tabu-based heuristic to iteratively adjust the assignment of nodes between neighboring sectors. At each iteration it identifies the sector with the longest tour and analyzes its nodes to find a candidate for relocation. This algorithm is specially suitable for scenarios with minor imbalances, as it relies on direct swapping nodes from the worst subtour and focuses on short-term improvements.

The second algorithm introduces a more structured redistribution of nodes. Unlike the first algorithm, which focuses only on the longest sector, this method Simultaneously considers donor sectors (those with the longest tours) and receiver sectors (those with the shortest tours). So it moves nodes from the donor sectors to its neighboring sectors to reduce the longest tourand transfers nodes from adjacent sectors into the receiver sectors to balance the workload and improve the shortest tour. By addressing both the longest and the shortest tours, this algorithm provides a more stable approach to balancing the sector assignments.

Finally, the third algorithm refines the redistribution process by evaluating the overall improvement in tour length before finalizing a move. It is useful in the exploitation phase.

The chapter details the key steps of each of them.

5.1 Single Shift Algorithm

The first algorithm, referred to as the *Single Shift* algorithm, employs a tabu-based heuristic to iteratively adjust node assignments between subtours. The primary focus of this method is to identify the sector (or subtour) with the longest length (the worst one), find a candidate node within it, and reassign the node to a neighboring sector. By doing so, the algorithm aims to minimize the length of the longest subtour in the current solution.

The choice of the candidate node uses a greedy criterion (minimum distance) that locally reduces the cost, making maximum use of the precomputed distance matrix. This approach is based on the theory of nearest neighbors, which is often effective in TSP problems.

The main steps of the algorithm are the following:

1. **Preprocessing (Distance Matrix Calculation):** at the beginning of the algorithm, the distance matrix is computed for all nodes in the problem space, including the depot. This matrix is used throughout the optimization process to efficiently calculate distances between nodes and sectors, ensuring that the algorithm remains computationally efficient.

2. **Sector Tours Calculation:** at the first iteration, the current sector tours are calculated using the OR-Tool algorithm presented before. Then, at each iteration, tours of sectors that have been changed by a shift are recalculated. This step ensures that the current solution reflects the best possible arrangement of nodes within each sector, given the current assignments. The sector with the longest subtour is identified as:

$$\text{Longest Tour Sector} = \arg \max_{k \in \{1,\dots,n_{\text{sectors}}\}} \text{length}(T_k),$$

   where T_k denotes the tour of sector k.

3. **Gap Percentage Calculation:** the gap percentage relative to the best available tour length (Best Length) is computed to evaluate the quality of the current solution:

$$\text{Gap Percentage} = \frac{\max(\text{length}(T_k)) - \text{Best Length}}{\text{Best Length}} \cdot 100.$$

   This metric is used to monitor progress and compare the solution with the known benchmarks.

4. **Candidate Node Selection:** once the sector with the longest subtour is identified, the algorithm evaluates all nodes within it as potential candidates for reassignment. For each candidate node:

   - The neighboring sectors (i.e., the two adjacent sectors $k - 2$ and $k + 2$) are considered as potential targets for reassignment.
   - The Euclidean distance between the candidate node and all nodes in the adjacent sector is found by exploiting the pre-computed distance matrix.
   - The candidate node that minimizes the distance to the neighboring sector and is not currently restricted by the tabu list is selected for reassignment. This selection ensures that the reassignment minimizes the disruption to the solution while promoting balance across sectors.

5. **Node Reassignment:** the selected node is reassigned to the neighboring sector. The tabu list is updated to include the reassigned node, preventing it from being moved for a fixed number of iterations (e.g., 5 iterations). The tabu list promotes exploration of new solutions by avoiding repetitive moves and cycling.

6. **Iteration and Stopping Criteria:** the process is repeated for a fixed number of iterations or until no movement between nodes is possible. At each iteration:
    - If a candidate node is successfully reassigned, the algorithm updates the solution and marks the iteration as an improvement.
    - If no candidate node can be reassigned, the algorithm terminates early, indicating that no further improvement is possible within the current configuration.
7. **Best Solution Tracking:** throughout the process, the algorithm tracks the best solution found, defined as the solution with the smallest longest subtour length. The time required to achieve this solution is also recorded to evaluate the efficiency of the algorithm.

A recap of the algorithm is shown below:

Double / Multiple Shift Algorithm

```
Input: Set of nodes with their coordinates
Output: Best sector tours with their lengths, percentage gap, and iteration details

1. Initialize the Parameters and Precompute the distance matrix:
    • Initialize distanceMatrix, iterationCounter, tabuList and tabuResetFrequency
    • Compute current sector tours lengths using OR-Tools
2. Iterative process:
    • While (iterationCounter < maxIterations and not convergence):
        – Identify sector with the longest tour
        – for each node in the longest sector:
            ∗ for each neighboring sector:
                · Calculate distance to nodes in this sector using distanceMatrix
            ∗ Select node that minimizes the distance and is not in tabuList
              and reassign the selected node to this sector
        – Recompute sector tours if any sector has changed
        – Update tabuList with recent move, remove old entries based on permanence
        – Track the best solution:
            ∗ if (current solution better than bestSolution):
                · bestSolution = current solution
        – iterationCounter += 1
3. Stopping criteria:
    • If (iterationCounter == maxIterations or no nodes reassigned):
    • Stop the algorithm
4. Return the best solution:
    • Output best sector tours, their lengths and the percentage gap to the
      optimal solution.
```

Performance Analysis: The *Single Shift* algorithm's performance was studied with respect to the permanence parameter of the tabu list, which controls how long a node remains restricted from being moved again. This parameter influences the algorithm's exploration capabilities and convergence rate:

- **Low Permanence:** when the tabu list allows nodes to be reassigned after a short number of iterations, the algorithm explores the solution space more rapidly. However, this can lead to cycling, where nodes oscillate between sectors without improving the solution.

- **High Permanence:** a longer permanence of the tabu list prevents cycling and allows the algorithm to focus on exploring new solutions. However, it may slow down convergence as the search space is more constrained.

5.2 Double / Multiple Shift Algorithm

The second algorithm, referred to as the *Double / Multiple Shift* algorithm, extends the concept of the *Single Shift* approach by introducing a more structured redistribution process. Instead of focusing solely on the longest subtour, this algorithm simultaneously considers both the longest and shortest subtours in the current solution (and also, if requested, the second-longest and shortest, and so on). By redistributing nodes from overburdened sectors (donors) to underutilized ones (receivers) the algorithm also minimizes the variance of subtour lengths, bringing the system closer to a uniform distribution and equity in workloads.

The steps of the algorithm are as follows:

1. **Preprocessing (Distance Matrix Calculation):** As done before, at the beginning of the algorithm, a distance matrix is computed for all nodes using the Euclidean distance.
2. **Initialization:**
   - Set the iteration counter to 0 and define the maximum number of iterations.
   - Initialize an empty tabu list to track recently moved nodes, preventing them from being reassigned too quickly.
   - Define a reset mechanism for the tabu list, which clears its contents every fixed number of iterations.
   - Initialize variables to store the best solution found during the process.
3. **Iterative Redistribution Process:** Repeat the following steps for each iteration:
   (a) **Reset Tabu List:** Periodically reset the tabu list, as done before, to allow previously restricted moves to be reconsidered. This mechanism avoids excessive exploitation of the same solution space and promotes exploration of new solutions.
   (b) **Sector Tours Calculation:** Compute the current sector tours and their respective lengths using OR-Tools. Identify:
      - The **donor sector**, which is the sector with the longest subtour:
        $$\text{Donor Sector} = \arg\max_k \text{length}(T_k),$$
        where T_k denotes the tour of sector k.

- The **receiver sector**, which is the sector with the shortest subtour:

$$\text{Receiver Sector} = \arg\min_k \text{length}(T_k).$$

(c) **Phase 1: Node Transfer from Donor Sector.**

- Evaluate all nodes in the donor sector as potential candidates for transfer to neighboring sectors (e.g., sectors $k-2$ and $k+2$, as done before).
- For each candidate node:
  - Check its distance to nodes in the neighboring sector using the distance matrix.
  - Select the candidate node that minimizes the distance to the neighboring sector, ensuring it is not restricted by the tabu list.
- Reassign the selected node to the neighboring sector and update the tabu list with the move.

(d) **Phase 2: Node Transfer to Receiver Sector.**

- Evaluate nodes from neighboring sectors as candidates for transfer to the receiver sector.
- For each candidate node:
  - Compute its distance to nodes in the receiver sector using the distance matrix.
  - Select the candidate node that minimizes the distance and satisfies the tabu list constraints.
- Reassign the selected node to the receiver sector and update the tabu list.

(e) **Gap Percentage Calculation and Best Solution Tracking:** After both phases, compute the gap percentage of the current solution and, if the current solution improves upon the best solution found so far (i.e., has a smaller gap percentage), update the best solution variables, including the time to achieve it and the iteration number.

(f) **Stopping Criteria:** The iterative process stops when:

- The maximum number of iterations is reached, or
- No improvement is observed for a certain number of iterations (i.e., no nodes are reassigned).

4. **Return the Best Solution:** After the iterations, the algorithm outputs the best sector tours, their lengths, and the percentage gap relative to the optimal solution.

A recap of the algorithm is shown below:

Double / Multiple Shift Algorithm

Input: Set of nodes with their coordinates
Output: Best sector tours with their lengths, percentage gap, and iteration details

```
1. Initialize the Parameters and Precompute the distance matrix:
    • Initialize distanceMatrix, iterationCounter, tabuList and tabuResetFrequency
2. Iterative process:
    • While (iterationCounter < maxIterations and not convergence):
        – If (iterationCounter mod tabuResetFrequency == 0):
            ∗ Reset tabuList
        – Recompute sector tours if any sector has changed
        – Identify donor sector (longest tour)
        – Identify receiver sector (shortest tour)
        – Phase 1: Node transfer from donor sector
            ∗ For each node in donor sector:
                · For each neighbor sector:
                · Compute distances using distanceMatrix
                · Select node minimizing the distance not in tabuList
                · Reassign node to neighboring sector
        – Phase 2: Node transfer to receiver sector
            ∗ For each node in neighboring sectors:
                · Compute distances to receiver sector using distanceMatrix
                · Select node minimizing the distance not in tabuList
                · Reassign node to receiver sector
        – Track the best solution:
            ∗ Compute gap percentage relative to optimal solution
            ∗ If (current solution better than best solution):
            ∗ bestSolution ← current solution
        – iterationCounter += 1
3. Stopping criteria:
    • If (iterationCounter == maxIterations or no nodes reassigned):
    • Stop the algorithm
4. Return the best solution:
    • Output best sector tours, their lengths, and the percentage gap to the optimal solution.
```

Performance Analysis: Similar to the *Single Shift* algorithm, the effect of the tabu list's permanence was analyzed:

- **Exploration:** The use of the tabu list ensures that repetitive moves are avoided within a certain time window, preventing cycling.
- **Exploitation:** By periodically resetting the tabu list, the algorithm allows previously restricted moves to be revisited, after other shifts have been made.

This combination enables the algorithm to maintain a balance between improving the current solution and exploring new areas of the solution space.

5.3 Convergence Phase Algorithm

The third algorithm, referred to as the *Convergence Phase* algorithm, takes an exploitation approach to node reassignment. Rather than relying solely on local criteria, such as minimum distance between nodes, this algorithm evaluates the overall improvement in the solution before finalizing the transfer. By simulating the impact of each potential move on the longest subtour length, it ensures that only improving moves are executed. This focus on solution improvement emphasizes convergence and stability. To make this process faster, at each iteration the iteration with the greatest impact on the worst tour is selected and executed (mixed greedy approach).

Due to its complexity, this was tested and used on a solution already reworked by the first two algorithms. In particular, it was useful for bringing an already stable system towards convergence and to evaluate the contribution of every possible move within the worst sectors.

The steps of the algorithm are as follows (Steps 1 and 2 are the same as before):

1. **Preprocessing: Distance Matrix Calculation.**
2. **Sector Tours Calculation.**
3. **Simulated Node Reassignment.** For each node in the longest sector:
   - All potential moves to other sectors are simulated. This includes:
     - Temporarily reassigning the node to a target sector.
     - Recomputing the sector tours and their lengths after reassignment.
   - The improvement is calculated as the reduction in the longest subtour length:

     Improvement = Length of Current Longest Tour−Length of New Longest Tour.

   - The move that results in the greatest improvement (i.e., the largest reduction in the longest subtour length) is selected. Only moves that produce an improvement are considered valid.
4. **Execution of the Best Move.** Once the best move is identified the node is permanently reassigned to the target sector.

5. **Stopping Criteria.** The algorithm terminates when one of the following conditions is met:
    - The maximum number of iterations is reached.
    - No improving moves can be found during an iteration, indicating that the solution has converged.

Focus on Convergence: This algorithm places a strong emphasis on ensuring convergence by evaluating the impact of each move before execution. This approach avoids unnecessary or non-improving moves and ensures that the solution improves steadily over time. While this strategy may result in longer computation times per iteration with respect to the others (that are really fast), due to the simulation of potential moves, it provides reliable solutions in some cases where the other two had stopped.

5.4 Further considerations

This section investigates computational complexities of the algorithms developed to solve the Min-Max Multiple Traveling Salesman Problem. Each algorithm leverages specific strategies for the problem's characteristics, notably its combinatorial nature and the requirement to minimize the maximum tour length among multiple salesmen.

5.4.1 Tabu Search and Local Search

Complexity Analysis

The computational complexity of these algorithms primarily arises from the local search steps involved in exploring the neighborhood of solutions. Each iteration involves calculating moves within the neighborhood, governed by:

$$O(n \times m),$$

where n represents the number of nodes, m the number of salesmen (sectors) involved.

Algorithm Suitability

The application of local search combined with tabu search fits well with Min-Max m-TSP due to its ability to continuously improve the worst-case scenario (maximum tour length) and also explore many different solution possibilities. By iteratively shifting nodes between tours and reassessing the maximum tour length, the algorithm effectively reduces the imbalance among the salesmen's workload, aligning with the objective of the problem.

5.4.2 Mixed Global Search

Complexity Analysis

The complexity of the *Convergence Phase* algorithm is higher due to the necessity to simulate each potential move's effect on the entire system. The computational demand is characterized by:

$$O(n^2 \times m),$$

This comprehensive approach ensures that only beneficial reassignments are made unlike in previous cases, where pure exploration moves were also allowed.

Algorithm Suitability

The global search strategy is effective for Min-Max mTSP as it addresses the optimization of the maximum tour directly by considering the systemic impact of each reassignment. This is crucial in ensuring that the solution not only improves locally, but also moves towards a globally optimized configuration, minimizing the maximum tour length across all salesmen. Furthermore, the *Convergence Phase* algorithm is typically deployed after the refinement steps, carried out by the *Single Shift* and *Double/Multiple Shift* algorithms. This sequential application leverages the already substantially good solutions, which meet or exceed, in some cases, the optimal configurations with the preliminary phases alone. In some cases, the improvements achieved at the *Double/Multiple Shift* phase are sufficient to either reach the best solutions, rendering the activation of the more global search phase unnecessary, unless further refinement is required. This strategy not only improves the efficiency of the optimization process, but also ensures that computational resources are used only when needed, thus optimizing both solution quality and algorithmic performance.

Conclusions

The mathematical strategies employed by the *Single Shift*, *Double/Multiple Shift*, and *Convergence Phase* algorithms are designed to address the challenges presented by the Min-Max m-TSP. The combination of solution search methodologies within a tabu search framework provides a robust solution, reducing the maximum tour length effectively, demonstrating their suitability in handling complex combinatorial optimization problems and also different data distributions, as we will see in the next chapter.

Chapter 6

Results and Tables

In this chapter we will present the results obtained from this thesis.

To ensure an effective comparison with our results, the tables contain, in addition to the results of the state-of-the-art algorithms, the best known solution (BKS, previously presented), the gap between the thesis and the best current solution (Gap Th-Best, in case some recent algorithm has surpassed the BKS) and the time taken by the thesis to reach the solution (Time Th).

6.1 Results on Dataset I instances

Instance	m	BKS	HGA [19]	Thesis	Gap Th-Best	Time HGA (sec)	Time Th
berlin52	2	4110.2	4110.2	4116.0	0.14 %	4.10 s	3.55 s
berlin52	3	3069.6	3069.6	3145.0	2.44 %	3.30 s	1.02 s
berlin52	5	2440.9	2440.9	2465.0	0.97 %	2.90 s	2.34 s
berlin52	7	2440.9	2440.9	**2440.0**	**-0.04** %	2.80 s	**1.21 s**
eil76	2	280.9	280.9	**279.0**	**-0.68** %	5.20 s	**5.02 s**
eil76	3	196.7	196.7	197.0	0.15 %	5.00 s	2.44 s
eil76	5	142.9	142.9	**142.0**	**-0.63** %	6.40 s	**3.13 s**
eil76	7	127.6	127.6	**127.0**	**-0.47** %	4.70 s	6.42 s

Table 6.1: Results on Dataset I

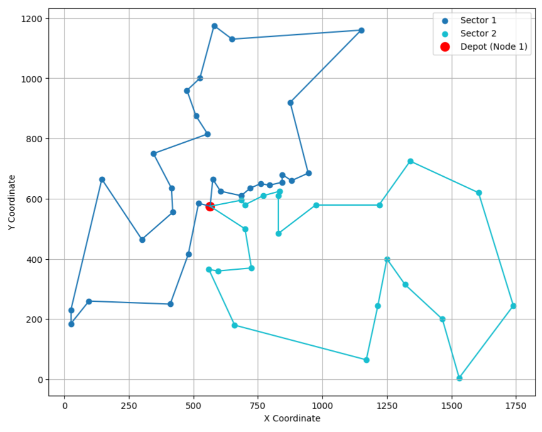


Figure 6.1: berlin52, m=2 after initialization

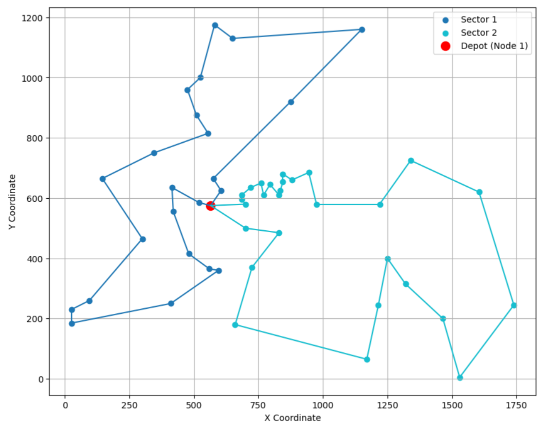


Figure 6.2: berlin52, m=2, after optimization process

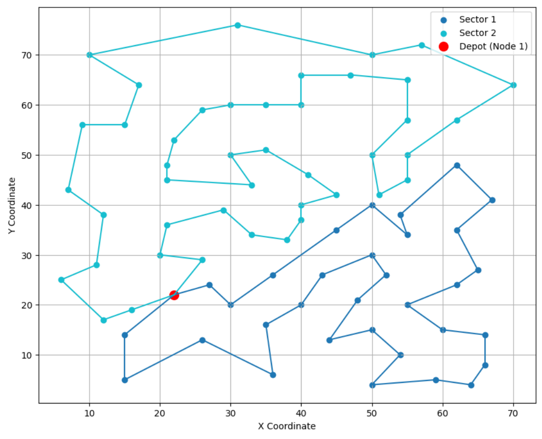


Figure 6.3: eil76, m=2 after initialization

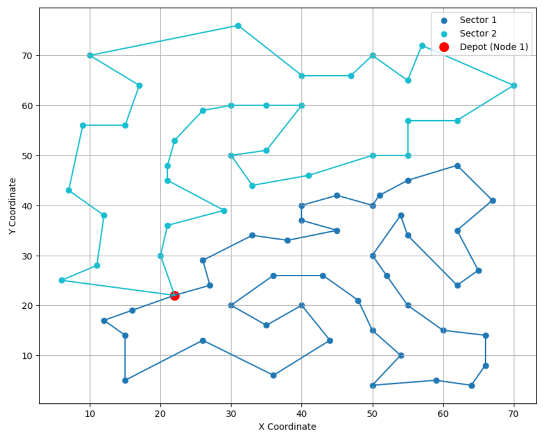


Figure 6.4: eil76, m=2, after optimization process

6.2 Results on Dataset II instances

For these instances and for those of Dataset III the comparison will be made between the best known solution (BKS, presented before), HGA [19], ITSHA [15] and HSNR [16] algorithms (the most recent and best in the literature) and the thesis. The results obtained with the thesis will also be compared in percentage with the best current solution (as seen in the previous table, some research has exceeded the BKS so far).

Note that these algorithms use the following cut-off times on these instances (they are the same also for Dataset III): $\frac{n}{5}$ for HGA and ITSHA, where n represents number of cities in the current dataset and $\frac{1.36*n}{5}$ for HSNR.

Instance	m	BKS	HGA [19]	ITSHA [15]	HSNR [16]	Thesis	Gap Th-Best	Time Th
mtsp51	3	159.0	159.0	159.0	159.0	**159.0**	**0.00** %	**3.87 s**
mtsp51	5	118.0	118.0	118.0	118.0	**118.0**	**0.00** %	**3.83 s**
mtsp51	10	112.0	112.0	112.0	112.0	**112.0**	**0.00** %	**0.90 s**
mtsp100	3	8507.0	8507.0	8507.0	8509.0	8623.0	1.36 %	10.95 s
mtsp100	5	6760.0	6760.0	6777.0	6766.0	7012.0	3.64 %	16.16 s
mtsp100	10	6358.0	6358.0	6358.0	6358.0	6472.0	1.79 %	11.21 s
mtsp100	20	6358.0	6358.0	6358.0	6358.0	**6358.0**	**0.00** %	**1.98 s**
mtsp150	3	13039.0	13039.0	13084.0	13174.0	13442.0	3.09 %	10.65 s
mtsp150	5	8416.0	8416.0	8466.0	8479.0	8715.0	3.55 %	8.41 s
mtsp150	10	5557.0	5564.0	5557.0	5616.0	5937.0	6.84 %	19.52 s
mtsp150	20	5246.0	5246.0	5246.0	5246.0	**5246.0**	**0.00** %	**9.62 s**

Table 6.2: Results on Dataset II

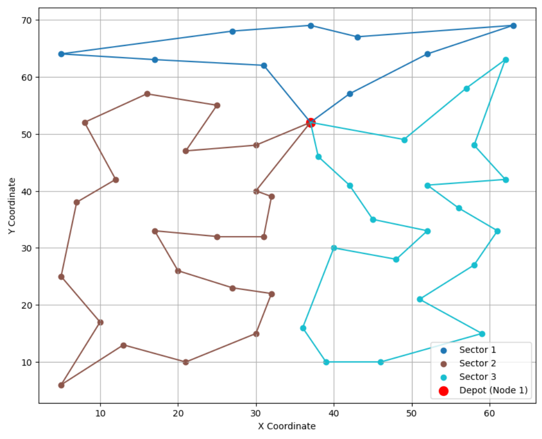


Figure 6.5: mtsp51, m=3 after initialization

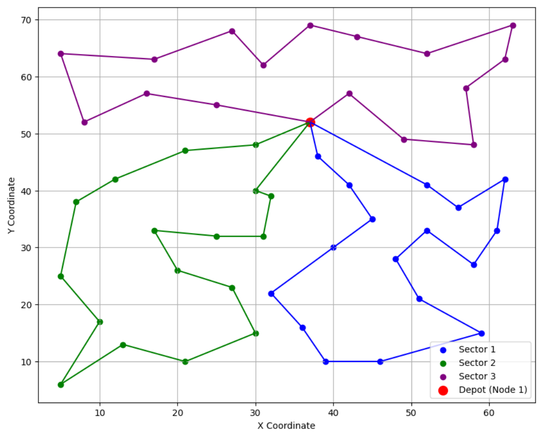


Figure 6.6: mtsp51, m=3, after optimization process

6.3 Results on Dataset III instances

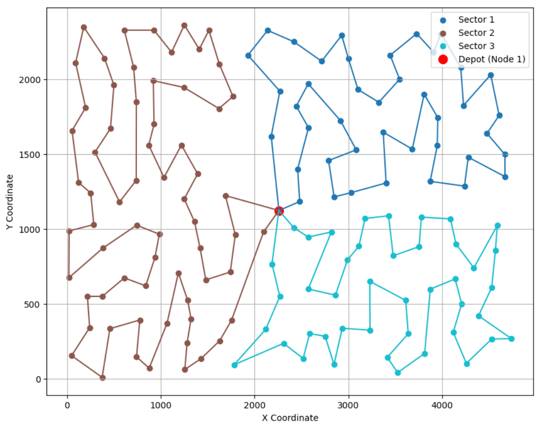


Figure 6.7: mtsp150, m=3 after initialization

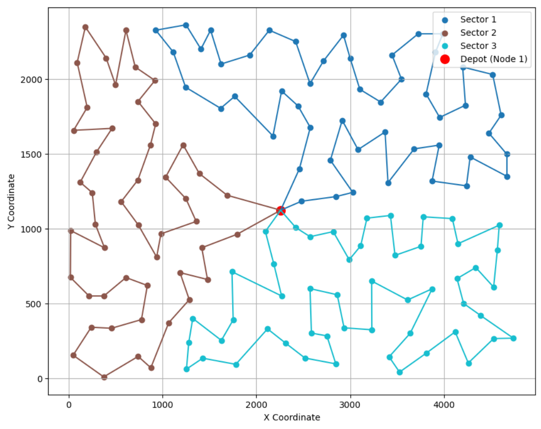


Figure 6.8: mtsp150, m=3, after optimization process

Instance	m	BKS	HGA [19]	ITSHA [15]	HSNR [16]	Thesis	Gap Th-Best	Time Th
rand100	3	3031.95	3031.95	3031.95	3031.95	3077.0	1.48 %	2.40 s
rand100	5	2409.63	2409.63	2411.68	2411.68	2428.0	0.75 %	13.51 s
rand100	10	2299.16	2299.16	2299.16	2299.16	**2298.0**	**-0.05** %	**3.79 s**
rand100	20	2299.16	2299.16	2299.16	2299.16	**2298.0**	**-0.05** %	**1.21 s**
ch150	3	2401.63	2405.94	2425.87	2407.34	**2397.0**	**-0.19** %	**29.08 s**
ch150	5	1740.63	1741.13	1740.63	1744.26	1791.0	2.89 %	2.11 s
ch150	10	1554.64	1554.64	1554.33	1554.64	1593.0	2.47 %	9.26 s
ch150	20	1554.64	1554.64	1554.33	1554.64	**1554.0**	**-0.04** %	**2.54 s**
kroa200	3	10691.0	10691.0	10760.69	10801.80	11014.0	3.02 %	33.71 s
kroa200	5	7412.12	7421.01	7470.78	7418.87	7818.0	5.85 %	40.24 s
kroa200	10	6223.22	6223.22	6223.22	6223.22	6375.0	2.43 %	29.48 s
kroa200	20	6223.22	6223.22	6223.22	6223.22	6277.0	0.85 %	3.72 s
lin318	3	15663.5	15698.61	15918.24	15918.24	16369.0	4.50 %	78.64 s
lin318	5	11276.8	11289.26	11548.44	11458.20	11609.0	3.03 %	65.71 s
lin318	10	9731.17	9731.17	9731.17	9731.17	9880.0	1.53 %	76.59 s
lin318	20	9731.17	9731.17	9731.17	9731.17	**9729.0**	**-0.02** %	**12.54 s**
rat783	3	3052.41	3128.69	3158.34	3262.52	3119.0	2.18 %	487.33 s
rat783	5	1961.12	1996.33	2024.27	2076.38	2081.0	6.07 %	487.89 s
rat783	10	1313.01	1393.77	1367.98	1358.06	1509.0	14.93 %	168.60 s
rat783	20	1231.69	1231.69	1231.69	1231.69	1259.0	2.22 %	252.53 s

Table 6.3: Results on Dataset III

6.4 Results comments and analysis

As can be seen from the numerical and graphical results, the presented algorithms are able to, in a short time, fix and improve our already good initialization basis results. The times

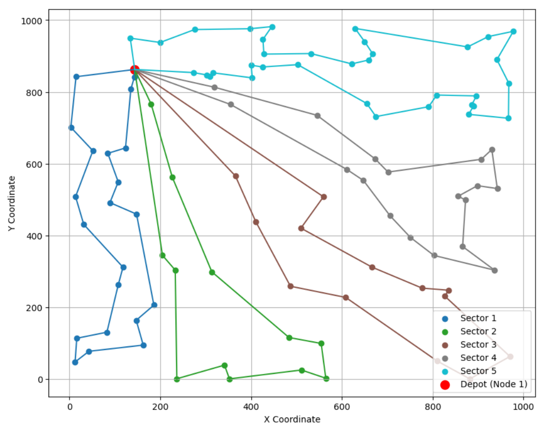


Figure 6.9: rand100, m=5, after initialization

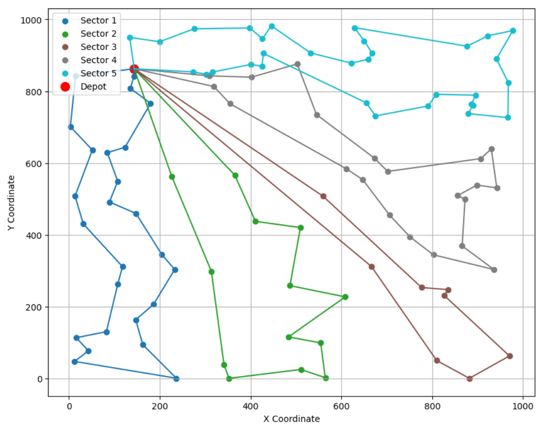


Figure 6.10: rand100, m=5, after optimization process

are better, in almost all cases, than the cut-off times applied by the algorithms in the literature for the various instances. This is possible thanks to a very good initialization, fast calculation of the subtours using OR-Tools and the application of the heuristic algorithms presented above. In particular, note that the spatial properties are well maintained even by the final assignments of points among the various salesmen.

In the analysis of the results it is also important to underline that, using this solution tool for every salesmen TSP, integer values are always obtained. It should therefore be kept in mind when comparing with other algorithms which, for various instances, use solvers that return floating-point values.

6.4.1 Fair Workloads: Final Tours' Length Analysis

In addition to reducing the maximum tour length as much as possible, another fundamental objective of this problem is fair assignment of work (distance to be covered by each operator).

The difference between the "best" (minimum distance) and "worst" (maximum distance) tours in the solutions obtained for the various instances will now be analyzed in detail.

For Dataset I instances:

Instance	m	Min Dist	Max Dist	Mean
berlin52	2	4095.0	4116.0	4105.5
berlin52	3	3123.0	3145.0	3136.67
berlin52	5	2443.0	2465.0	2451.8
berlin52	7	1960.0	2440.0	2205.86
eil76	2	277.0	279.0	278.0
eil76	3	185.0	197.0	192.0
eil76	5	137.0	143.0	140.0
eil76	7	124.0	127.0	125.71

Table 6.4: Analysis on Dataset I

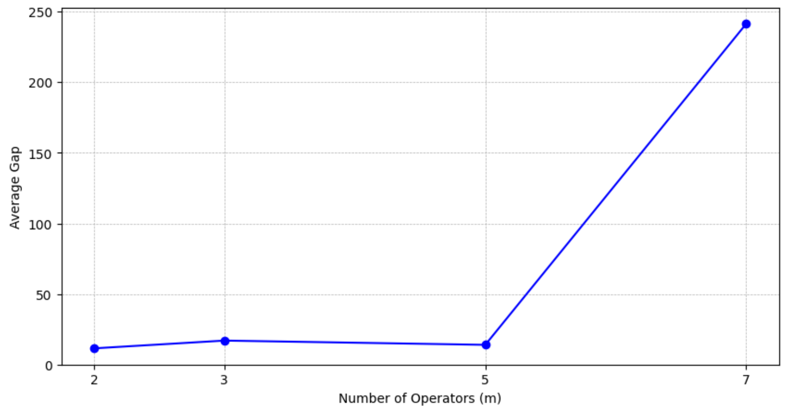


Figure 6.11: Average Gap vs. Number of Operators in Dataset I

For Dataset II instances:

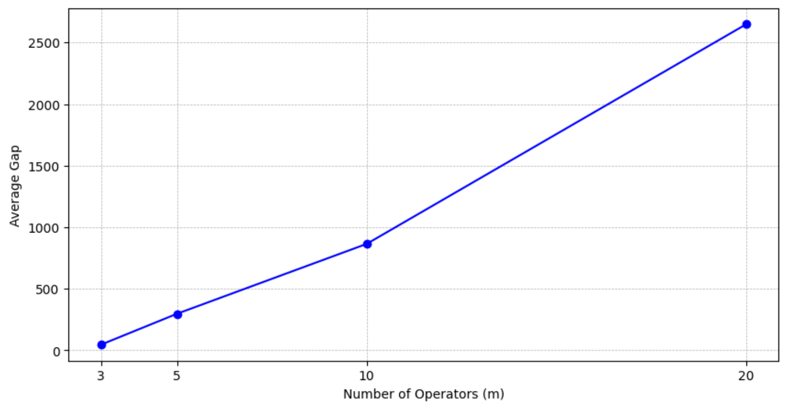


Figure 6.12: Average Gap vs. Number of Operators in Dataset II

Instance	m	Min Dist	Max Dist	Mean
mtsp51	3	153.0	159.0	156.33
mtsp51	5	111.0	118.0	116.0
mtsp51	10	79.0	112.0	90.6
mtsp100	3	8554.0	8623.0	8583.0
mtsp100	5	6304.0	7012.0	6819.6
mtsp100	10	4022.0	6472.0	5481.33
mtsp100	20	2552.0	6358.0	4702.63
mtsp150	3	13377.0	13442.0	13408.0
mtsp150	5	8538.0	8715.0	8664.2
mtsp150	10	5824.0	5937.0	5893.61
mtsp150	20	3752.0	5246.0	4723.42

Table 6.5: Analysis on Dataset II

For Dataset III Instances:

Instance	m	Min Dist	Max Dist	Mean
rand100	3	3064.0	3077.0	3071.0
rand100	5	2307.0	2428.0	2381.2
rand100	10	1746.0	2298.0	2012.4
rand100	20	1639.0	2298.0	1895.78
ch150	3	2388.0	2397.0	2393.33
ch150	5	1749.0	1791.0	1780.6
ch150	10	1541.0	1593.0	1562.3
ch150	20	803.0	1554.0	1255.65
kroa200	3	10920.0	11014.0	11032.33
kroa200	5	7671.0	7818.0	7785.6
kroa200	10	6017.0	6375.0	6231.5
kroa200	20	3661.0	6277.0	5327.4
lin318	3	16296.0	16369.0	16324.67
lin318	5	11518.0	11609.0	11578.4
lin318	10	7703.0	9880.0	9048.4
lin318	20	6214.0	9729.0	8258.84
rat783	3	3118.0	3119.0	3118.67
rat783	5	2052.0	2081.0	2063.6
rat783	10	891.0	1509.0	1262.0
rat783	20	707.0	1259.0	996.75

Table 6.6: Analysis on Dataset III

From this analysis, it is clear that, for a lower number of operators, balanced solutions are always generated, with the work being very well distributed. In instances with more

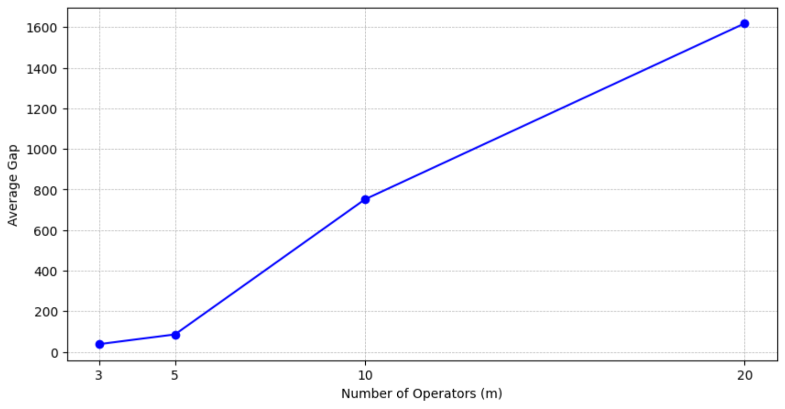


Figure 6.13: Average Gap vs. Number of Operators in Dataset III

salesmen (7, 10 and 20), the presence of many operators causes a more unbalanced distribution. It should also be taken into account that the algorithm stops when it reaches an optimal conformity. In some cases, for example, the optimum is reached and the solution is not handled further. Also, in cases with few cities, it is not always possible to reach the optimum with a different configuration from the one already found. However, in those with a greater number of nodes, a proposal could be to proceed with further refinement of the subtours. This operation should be performed with particular care, making sure not to cause a worsening of the currently longest tour.

Chapter 7

Conclusions

This thesis focused on the search for heuristic and innovative solutions for the Min-Max mTSP, a variant of the classical Traveling Salesman Problem that introduces the complexity of multiple agents and the objective of minimizing the maximum route length.

At the beginning, the steps taken were those towards a theoretical understanding and the reading of various papers in the literature, to better catch both the nature of the problem and the strategies used up to this point. This phase was very useful and led to the development of various scenarios and ideas.

The first practical attempts thus were those regarding the initialization algorithm. As shown in the thesis, initially we were focusing on the separation of nodes in space through clustering (in particular k-means, so as to be able to control the number of clusters, and therefore salesmen, generated). The problems encountered with this development were multiple: from the imprecise assignment of points to salesmen, to the complexity linked to the excess of intersections in the tours generated. So, after further reasoning, it was decided to try an alternative and completely new path.

The main inspiration here were the studies made by Sasan Mahmoudinazlou and Changhyun Kwon (2024) [19] on the detection and removal (via specific algorithms) of subtour intersections. Hence, the idea of an initialization algorithm capable of simplifying the creation of subtours, dividing the nodes among the salesmen following a simple but effective principle: their spatial position and angle with respect to the depot. This intuition, then developed through the *Angular Sector Division* algorithm, proved not only to be very effective in creating particularly advantageous initial subtours, but also extremely robust to different datasets and therefore different spatial distributions of points with respect to the depot.

For the optimization algorithms, the reasoning was to test various heuristic strategies aimed at improving, step by step, the longest tour (following the main objective of the problem) and at the same time thinking about the fairness of the distribution of work between the various operators. The very first attempt was to iteratively lighten the worst subtour of a node, as shown in the *Single Shift* algorithm. This was soon incorporated with that of *Double/Multiple Shift*, which instead allows manipulating multiple tours at the same time, removing nodes from the worst tours and adding ones to the best ones. Finally, a last attempt towards the analysis and convergence of tours was made with the

Convergence Phase, which combines a global approach with a greedy choice of the node to transfer.

In this work, even if in some cases we produced results dominated by the state of the art algorithms available in literature, still, by exploiting heuristic techniques and reasoning related to the nature of the problem, we were able to develop a robust and interesting solution for the Min-Max mTSP. This opens the door to possible future developments. In particular, towards algorithms that increasingly take more into account the spatial properties of the problem and exchange nodes in the most effective way possible.

Bibliography

[1] Gerhard Reinelt. 1991. *TSPLIB - A Traveling Salesman Problem Library.* INFORMS J. Comput., 376-384.

[2] Karla L. Hoffman, Manfred Padberg, Giovanni Rinaldi. 2001. *Traveling salesman problem.* George Mason University, New York University, CNR IASI Rome.

[3] Park, Y.-B. 2001. *A hybrid genetic algorithm for the vehicle scheduling problem with due times and time deadlines.* International Journal of Production Economics 73(2) 175-188.

[4] Junjie, P., W. Dingwei. 2006. *An ant colony optimization algorithm for multiple travelling salesman problem.* First International Conference on Innovative Computing, Information and ControlVolume I (ICICICâ06), vol. 1. IEEE, 210-213.

[5] Carter, A. E., C. T. Ragsdale. 2006. *A new approach to solving the multiple traveling salesperson problem using genetic algorithms.* European Journal of Operational Research 175(1) 246-257.

[6] Brown, E. C., C. T. Ragsdale, A. E. Carter. 2007. *A grouping genetic algorithm for the multiple traveling salesperson problem.* International Journal of Information Technology and Decision Making 6(02) 333-347.

[7] Liu, W., S. Li, F. Zhao, A. Zheng. 2009. *An ant colony optimization algorithm for the multiple traveling salesmen problem.* 2009 4th IEEE Conference on Industrial Electronics and Applications. IEEE, 1533-1537.

[8] Singh, A., A. S. Baghel. 2009. *A new grouping genetic algorithm approach to the multiple traveling salesperson problem.* Soft Computing 13 95-101.

[9] Donald Davendra. 2010. *Traveling Salesman Problem,Theory and Applications.* Intech Open.

[10] Roberto Tadei, Federico Della Croce. 2010. *Elementi di RICERCA OPERATIVA.* Societa' Editrice Esculapio.

[11] Chen, S.H., M.C. Chen. 2011. *Operators of the two-part encoding genetic algorithm in solving the multiple traveling salesmen problem.* 2011 International Conference on Technologies and Applications of Artificial Intelligence. IEEE, 331-336.

[12] Venkatesh P., A. Singh. 2015. *Two metaheuristic approaches for the multiple traveling salesperson problem.* Applied Soft Computing, 74-89.

[13] Soylu, B. 2015. *A general variable neighborhood search heuristic for multiple traveling salesmen problem.* Computers and Industrial Engineering 90 390-401.

[14] Wang, Y., Y. Chen, Y. Lin. 2017. *Memetic algorithm based on sequential variable neighborhood descent for the min-max multiple traveling salesman problem.* Computers and Industrial Engineering 106 105-122.

[15] Jiongzhi Zheng, Yawei Hong, Wenchang Xu, Wentao Li, Yongfu Chen. 2022. *An Effective Iterated Two-stage Heuristic Algorithm for the Multiple Traveling Salesmen Problem.* Computers and Operations Research 143 105772.

[16] Pengfei He., J. K. Hao. 2022. *Hybrid genetic algorithm for routing problems. Data Structures and Algorithms*, University of Angers.

[17] Paolo Brandimarte. 2022. *Ottimizzazione per la Ricerca Operativa.* CLUT editrice.

[18] Perron, L., V. Furnon. 2024. *OR-Tools v9.10.* URL https://developers.google.com/optimization/

[19] Sasan Mahmoudinazlou, Changhyun Kwon. 2024. *A Hybrid Genetic Algorithm for the min-max Multiple Traveling Salesman Problem.* Computers and Operations Research 162 106455.