Università degli Studi di Padova

School of Engineering Department of Information Engineering

Master Degree in Computer Engineering

Performance analysis on Traveling Salesman Problem solving techniques

Operations Research 2

Supervisor:
Matteo Fischetti

Candidates:
Ippolito Lavorati (2131127)
Andrea Bruttomesso (2120933)

Academic Year 2024/2025

Abstract

The Travelling Salesman Problem (TSP) is a fundamental and widely studied optimization problem in the field of Operations Research. It was originally formulated to find the shortest possible route that visits a set of cities exactly once and returns to the starting point. Despite its simple formulation, the TSP is NP-hard, making it computationally challenging to solve for large instances.

Over the decades, a variety of algorithmic approaches have been developed to tackle the problem, ranging from exact methods to heuristics and metaheuristics. Among these, Concorde represents the state-of-the-art in exact TSP solvers, leveraging advanced optimization techniques. However, such high-performance tools are often complex and computationally demanding.

This study aims to explore and compare a selection of more accessible algorithmic strategies, including classical heuristics, metaheuristics and exact methods. Through implementation and experimental evaluation, the work provides insights into their practical performance and highlights the trade-offs between computational efficiency and solution quality in solving the TSP.

Contents

Chapter 1

Introduction

The Travelling Salesman Problem (TSP) has long served as a benchmark in combinatorial optimization and computational complexity. Although its origins can be traced back to the 19$^{\text{th}}$ century with the work of Hamilton and Kirkman [1], it was not until the mid-20$^{\text{th}}$ century that the problem gained serious scientific traction. The TSP asks whether a minimal-cost cycle exists that visits each node in a graph exactly once and returns to the starting point—a problem later proven to be NP-hard by Richard M. Karp in 1972 [2].

The relevance of the TSP extends well beyond its theoretical appeal: it arises naturally in domains such as logistics, manufacturing, genome sequencing, and circuit design. Solving it efficiently has therefore been a longstanding goal in both academic and industrial research. Over the years, a wide variety of algorithms have been proposed, ranging from exact methods based on integer programming to heuristic and metaheuristic approaches.

One of the most advanced exact solvers is Concorde, which incorporates sophisticated techniques such as cutting planes, branch-and-bound, and polyhedral combinatorics [3]. While extremely powerful, tools like Concorde are complex to implement and computationally intensive.

This thesis focuses on more accessible algorithmic strategies for solving the TSP. Rather than aiming for optimality at any cost, the goal is to explore and compare classic heuristics and metaheuristics in terms of their practicality, computational efficiency, and solution quality across a range of problem instances.

1.1 Problem Formulation

Let us consider an undirected graph $G = (V, E)$, where V is the set of $|V| = N$ nodes (or vertices), and E is the set of $|E| = M$ edges. A *Hamiltonian cycle* of G, denoted by $G^* = (V, E^*)$, is a subgraph whose edges form a cycle that visits each node $v \in V$ exactly once. We define a cost function $c : E \to \mathbb{R}^+$ that assigns a non-negative cost $c_e = c(e)$ to each edge $e \in E$. The objective of the Travelling Salesman Problem (TSP) is to find a

Hamiltonian cycle in G that minimizes the total cost, defined as:

$$\text{cost(cycle)} := \sum_{e \in \text{cycle}} c(e)$$

This problem can be formulated as an Integer Linear Programming (ILP) model. We define binary decision variables x_e to indicate whether edge e is included in the optimal cycle:

$$x_e = \begin{cases} 1 & \text{if } e \in E^* \\ 0 & \text{otherwise} \end{cases} \quad \forall e \in E$$

The ILP model is as follows:

$$\text{minimize} \quad \sum_{e \in E} c_e x_e \tag{1.1}$$

$$\text{subject to} \quad \sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V \tag{1.2}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad \forall S \subset V,\, v_1 \in S \tag{1.3}$$

$$x_e \in \{0, 1\} \quad \forall e \in E \tag{1.4}$$

Constraint (1.2) ensures that each node has degree 2, i.e., it is entered and exited exactly once. However, these constraints alone are not sufficient to guarantee a single, valid Hamiltonian cycle: the solution may consist of multiple disjoint cycles. To prevent this, the model includes constraint (1.3), known as the *Subtour Elimination Constraints* (SECs), which ensure that the solution forms a single connected cycle where every node $v \neq v_1$ is reachable from v_1.

Despite their theoretical importance, SECs are defined for every subset $S \subset V$ containing v_1, and their number is exponential in N. Including all of them simultaneously is therefore computationally infeasible.

Throughout this report, all pseudocode and algorithmic approaches will assume an undirected, complete graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}^+$.

Chapter 2

Heuristic Methods

A heuristic is a practical problem-solving technique designed to produce good-enough solutions within a reasonable time frame, especially when exact methods are too slow or infeasible. Given the NP-hard nature of the Travelling Salesman Problem, computing the optimal solution can be computationally expensive even for moderately sized instances. For this reason, heuristic methods play a crucial role in tackling the problem efficiently. Heuristics do not guarantee optimality, but they can often provide solutions that are sufficiently close to the best possible one. This trade-off between accuracy and speed makes heuristics particularly valuable when timely decision-making is more important than absolute precision.

In this chapter, several heuristic strategies for the TSP are presented, highlighting their underlying principles, implementation, and performance.

2.1 Nearest Neighbour - Greedy

One of the most intuitive heuristic approaches to the Travelling Salesman Problem is the **Nearest Neighbor** (NN) algorithm, which follows a greedy strategy. A *greedy* algorithm builds a solution step by step by always choosing the locally optimal option, with the hope that this leads to a good global solution. In the context of the TSP, the algorithm starts from an initial node and, at each iteration, selects the nearest unvisited node as the next step in the tour. This process continues until all nodes have been visited exactly once, and the cycle is closed by returning to the starting point. However, the quality of the solution obtained using this method depends heavily on the starting node, as different starting points may lead to significantly different tours. While NN is fast and easy to implement, often produces suboptimal solutions. Still, it often produces a reasonable approximation in a fraction of the time required by exact methods, making it suitable for large instances where exact algorithms are computationally expensive.

Algorithm 1 Nearest Neighbor Heuristic

Require: Starting node $s \in V$
Ensure: Hamiltonian cycle of G, cost of cycle
  $cycle \leftarrow [s]$
  $cost \leftarrow 0$
  for $i \leftarrow 0$ to $|V| - 2$ **do**
    $next \leftarrow \arg\min_{v \in V \setminus cycle} c(cycle[i], v)$
    $cost \leftarrow cost + c(cycle[i], next)$
    $cycle[i+1] \leftarrow next$
  end for
  $cost \leftarrow cost + c(cycle[|V|-1], s)$
  return $cycle$, $cost$

2.1.1 Results Analysis

The greedy algorithm often selects locally optimal edges without considering long-term implications, which may lead to expensive connections, especially when few unvisited nodes remain. This behavior can result in unnecessarily long edges and poor-quality tours. Such issues are particularly evident in Euclidean instances, where the resulting path may include edge crossings, which are absent in optimal solutions.

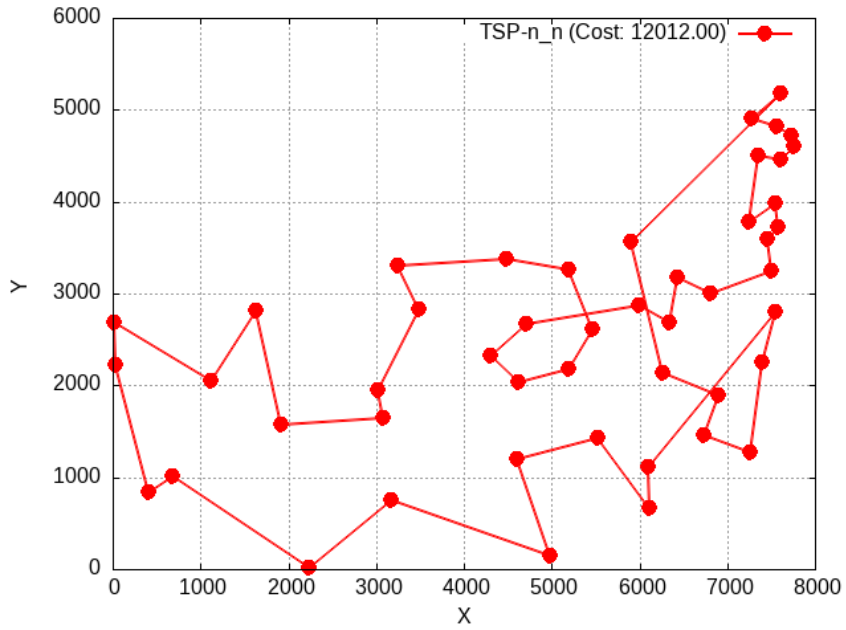


Figure 2.1: Nearest Neighbor heuristic solution.

Figure 2.1 illustrates the best tour obtained by running the Nearest Neighbor heuristic from each possible starting node on the *Att48* instance. Several edge crossings are visible, indicating inefficiencies. The resulting tour cost is **12012**, significantly higher than the optimal cost of **10628**.

To address these limitations, the next section introduces the 2-opt heuristic, a simple yet effective local search technique aimed at improving tour quality by eliminating such inefficiencies.

2.2 Two-Opt

The *2-opt* algorithm is a local search optimization technique designed to improve a given (possibly sub-optimal) tour in the Travelling Salesman Problem. It works by iteratively removing two edges from the tour and reconnecting the resulting paths in the opposite way, with the goal of reducing the overall cost of the cycle. Formally, we can interpret 2-opt as a particular case of a more general class of heuristics known as *k-opt*, where k edges are removed and reconnected to form a new valid tour. In the 2-opt case, we identify two edges (p_i, p_{i+1}) and (p_j, p_{j+1}) such that replacing them with (p_i, p_j) and (p_{i+1}, p_{j+1}), while reversing the intermediate segment between p_{i+1} and p_j.

The 2-opt move is beneficial and accepted only if replacing the two selected edges reduces the overall tour cost, i.e., if:

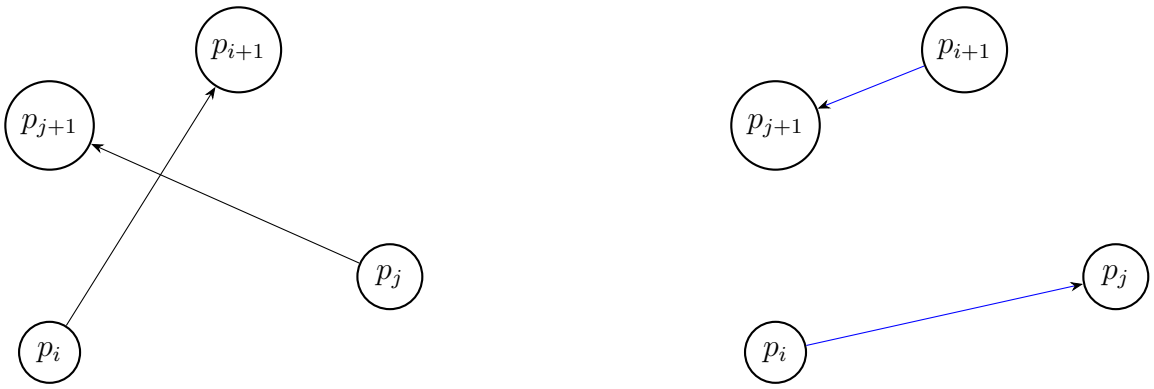
$$c(p_i, p_{i+1}) + c(p_j, p_{j+1}) > c(p_i, p_j) + c(p_{i+1}, p_{j+1}) \tag{2.1}$$

Equivalently, the cost change Δ resulting from this move can be computed as:

$$\Delta = [c(p_i, p_j) + c(p_{i+1}, p_{j+1})] - [c(p_i, p_{i+1}) + c(p_j, p_{j+1})] \tag{2.2}$$

If $\Delta < 0$, the move yields a shorter tour and is therefore accepted.

After identifying such a pair of edges, the segment of the tour between nodes p_{i+1} and p_j is reversed to maintain tour feasibility.



(a) Before the move **(b)** After the move

Figure 2.2: Example of a 2-opt move eliminating edge crossing.

As shown in Figure 2.2, this operation is particularly useful in Euclidean instances, where removing intersecting edges usually improves the tour cost significantly. The process is repeated iteratively: at each step, the algorithm scans the tour looking for a pair of edges satisfying the inequality. If found, the edges are swapped and the search continues. When no further improvement can be found, the procedure terminates. This leads to a solution that is often significantly better than the initial one, reaching a local optima.

To improve the Nearest Neighbor result, the 2-opt heuristic is applied as a post-processing step to the best tour found among all possible NN starting nodes. This selective

refinement avoids unnecessary computation while still producing a significantly improved solution.

2.2.1 Pseudocode

Algorithm 2 Two-Opt Heuristic for TSP

Input: A Hamiltonian cycle `solution` of a graph G, with associated cost
Output: An improved Hamiltonian cycle (locally optimal w.r.t. 2-opt), updated cost

```
procedure ApplyTwoOpt(solution, nnodes)
    improved ← true
    while improved do
        improved ← false
        for i ← 0 to nnodes − 2 do
            for j ← i + 1 to nnodes − 1 do
                Δ ← CostDelta(i, j, solution)
                if Δ < 0 then
                    ReverseSubsequence(solution, i + 1, j)
                    solution.cost ← solution.cost + Δ
                    improved ← true
                end if
            end for
        end for
    end while
    return solution
end procedure
```

Algorithm 3 CostDelta and Subsequence Reversal

```
function CostDelta(i, j, solution)
    Let p_i, p_{i+1}, p_j, p_{j+1} be the involved nodes
    return c(p_i, p_j) + c(p_{i+1}, p_{j+1}) − c(p_i, p_{i+1}) − c(p_j, p_{j+1})
end function
procedure ReverseSubsequence(solution, start, end)
    Reverse the order of nodes in solution between indices start and end (inclusive)
end procedure
```

The `ApplyTwoOpt` procedure iteratively applies improving 2-opt moves until no further improvement is possible. The cost difference `CostDelta` evaluates the gain of removing two edges and reconnecting them in the opposite direction. If an improvement is found, the affected segment of the tour is reversed.

2.2.2 Results Analysis

To prove the effectiveness of the 2-opt approach, we apply it to the best tour generated by NN heuristic presented in Section 2.1.1. Figure 2.3 shows how this technique is as simple as it is effective. All edge crossings have been eliminated, resulting in a visibly more efficient tour.

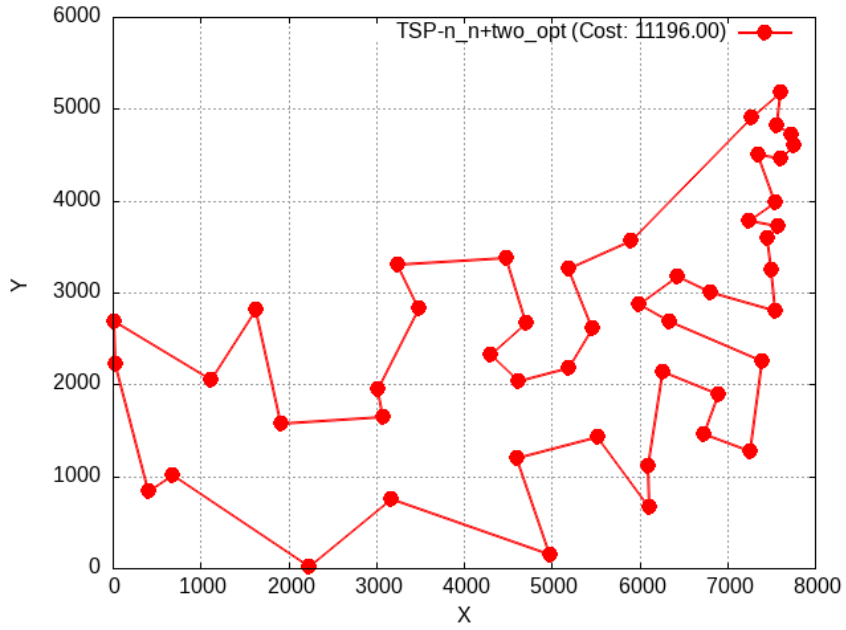


Figure 2.3: 2-opt solution applied to the best tour generated by NN.

To evaluate the improvement quantitatively, we can compare the cost of the tour before and after applying 2-opt and the computing time required to run both approaches. Table 2.1 summarizes the results of this comparison and highlights the significant reduction in tour cost achieved by the 2-opt heuristic, as well as the minimal increase in computation time.

Table 2.1: Comparison of tour cost and computation time.

Strategy	Cost	Time
NN	12012	0.006172 s
NN + 2opt	**11196**	**0.006461 s**

These results are only meant to give an idea of the effectiveness of the 2-opt heuristic, some more rigorous analysis will be presented in Section 6.2.1.

Chapter 3

Metaheuristic Methods

A metaheuristic is a high-level, abstract problem-solving strategy designed to efficiently explore and exploit large solution spaces in complex optimization problems. Unlike algorithms tailored to specific instances, metaheuristics are general-purpose frameworks that guide subordinate heuristics leveraging iterative improvement, intelligent search patterns, and adaptive behavior to find near-optimal solutions.

In our experiments, we employ metaheuristics that use the *2-opt* algorithm as a foundational local search heuristic. Specifically, we take greedy solutions refined by 2-opt as initial configurations and further enhance them using one of two metaheuristic methods: **Tabu Search** and **Variable Neighborhood Search (VNS)**.

These strategies offer the ability to explore a broader search space than simple heuristics by incorporating mechanisms that allow occasional acceptance of worse solutions—thereby escaping local optima. We demonstrate that even relatively simple metaheuristic ideas, when combined with previously introduced heuristics, can yield solutions remarkably close to the optimal ones.

3.1 Tabu Search

Tabu Search is a metaheuristic designed to efficiently explore the solution space while avoiding entrapment in local optima. Originally introduced by Fred Glover in 1986 and formalized in 1989 [4], it is particularly effective for combinatorial optimization problems such as the Traveling Salesman Problem. It enhances local search by allowing non-improving moves and uses a memory structure, known as the *tabu list*, to prevent cycling back to recently visited solutions.

3.1.1 Search Strategy

The algorithm alternates between two phases:

- **Intensification:** A standard 2-opt local search is applied to refine the current solution by exploring its neighborhood until a local minima.

- **Diversification:** When no improvement occurs for a fixed number of iterations, a perturbation is introduced (via a random move), and the tabu tenure is increased to promote broader exploration of the solution space.

3.1.2 Implemented Mechanism

Our implementation follows this procedure:

1. Generate an initial solution, either randomly or using a greedy heuristic.
2. At each iteration, evaluate all possible 2-opt moves and select the best one that is not tabu, unless it satisfies the aspiration criterion.
3. Apply the move; if it does not improve the solution, it is added to the tabu list.
4. If a better solution is found, and the tenure is greater than the minimum, the tenure is decreased to intensify the search locally.
5. If no improvement is observed for a specified number of iterations, a random move is performed and the tenure is increased to encourage exploration.

This approach results in a simple yet adaptive mechanism for dynamically adjusting the tabu tenure based on the search progress. It does not rely on predefined schedules or functions and allows the algorithm to adaptively balance between exploration and exploitation.

3.1.3 Tabu List Details

Each tabu entry represents a 2-opt move, recorded by storing the indices of the swapped edges. These entries remain in the tabu list for a number of iterations determined by the current tenure. The list prevents the immediate reversal of recent moves, guiding the search toward unexplored areas of the solution space. The aspiration criterion permits overriding the tabu status if a move leads to a better solution than any previously found.

3.1.4 Pseudocode

Algorithm 4 Tabu Search (simplified)

```
sol ← initial solution
best_sol ← sol
tabulist ← ∅
while time limit not exceeded do
    move ← best 2-opt move not in tabu list
    Apply move to sol
    if sol is better than best_sol then
        best_sol ← sol
        if tenure > minimum then
            Decrease tenure
        end if
    else
        Add move to tabu list
        Increase non-improvement counter
    end if
    if non-improvement counter exceeds threshold then
        Apply a random move
        if tenure < maximum then
            Increase tenure
        end if
    end if
end while
```

3.1.5 Search Behavior Analysis

To better understand the behavior of Tabu Search, we logged the solution cost and tabu tenure at each iteration and visualized them in Figure 3.1. The plot shows how the algorithm initially descends rapidly from a greedy starting solution, then undergoes phases of stagnation and diversification. The tenure increases during these periods to promote exploration, while aspiration criteria allow breakthroughs despite tabu constraints. As the plot illustrates, most of the improvement occurs early, but the dynamic tenure adaptation enables occasional improvements even in late iterations.

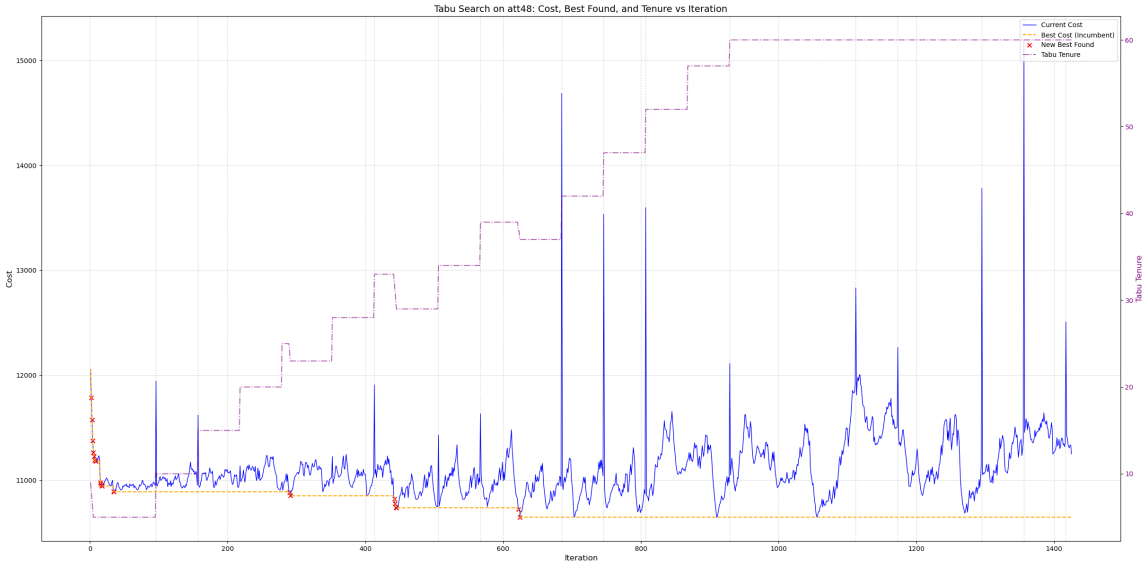


Figure 3.1: Evolution of solution cost (blue), best solution found (orange), and tabu tenure (purple, right axis) across iterations for Tabu Search on `att48`. Red crosses indicate when a new best solution was discovered.

3.2 Variable Neighborhood Search (VNS)

A known limitation of the Tabu Search algorithm is the need to carefully tune several hyperparameters, such as the size and policy of the tabu list (static or dynamic tenure), which may be unaffordable in real-world scenarios and can lead to overfitting. Furthermore, diversification steps in Tabu Search might waste computational resources without significantly improving solution quality.

A simpler alternative is the *Variable Neighborhood Search (VNS)* metaheuristic [5], which addresses the same problem from a different perspective. Rather than maintaining a tabu list to prevent reversals, VNS uses random *k-opt* swaps, called **kicks**, to escape local minima. Since a k-opt move alters more than two edges, it is less likely to be undone by a simple 2-opt move.

3.2.1 Algorithm Description

The VNS algorithm iteratively improves a solution using 2-opt until no better neighbor is found. Then, it performs a series of random 3-opt kicks to perturb the solution and continues the search. This avoids complete restarts (as in multistart heuristics) and ensures better locality preservation.

The number of kicks is randomly selected within a fixed range $[k_{\min}, k_{\max}]$ and scaled by a learning factor. This value is further adjusted based on the number of iterations since the last improvement, allowing the algorithm to increase its exploration as it stagnates, this trade-off is analyzed experimentally in Section 6.2.3.

3.2.2 3-Opt Kick Structure

Each kick is implemented as a 3-opt move, which removes three edges and reconnects the resulting segments using one of seven possible reconnection types. For each kick, a valid triple of split indices (i, j, k) is selected at random, and all reconnection variants are evaluated. The configuration with the lowest cost is retained and applied to the current solution. Figure 3.2 illustrates a typical 3-opt transformation.

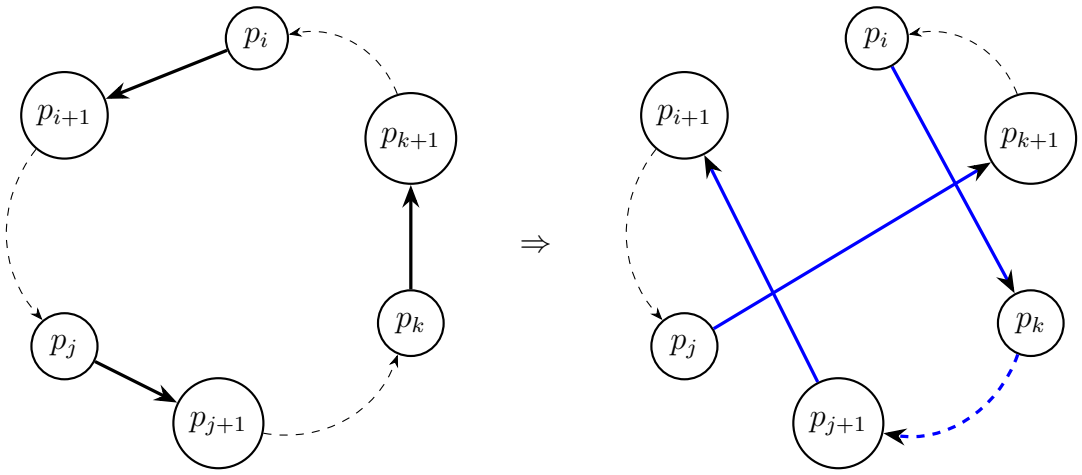


Figure 3.2: Effect of a 3-opt kick applied to a TSP tour.

3.2.3 Pseudocode

Algorithm 5 VNS high-level pseudocode

Input: Instance `inst`, initial solution `sol`
Output: Best solution found

if solution not initialized **then**
    Generate initial path (greedy or random)
end if
while time not exceeded **do**
    Apply 2-opt to refine `sol`
    if `sol` improves **then**
        Update best known solution
    else
        Compute number of 3-opt kicks (based on $[k_{\min}, k_{\max}]$, learning rate, stagnation)
        for each kick **do**
            Apply 3-opt with best of 7 reconnection types
        end for
    end if
end while
return best solution found

3.2.4 Search Behavior Analysis

To better understand the internal dynamics of the VNS algorithm, we tracked the solution cost, number of 3-opt kicks applied, and moments when new best solutions were discovered. Figure 3.3 shows a representative run on the `att48` instance. This execution was deliberately performed with a reduced time limit in order to produce a visually rich

and interpretable trace of the algorithm's behavior over a large number of iterations. The results should therefore be interpreted qualitatively rather than quantitatively.

Figure 3.3: Evolution of solution cost (blue), best solution found (orange), and number of 3-opt kicks (green, right axis) across iterations for VNS on `att48`. Red crosses indicate new best solutions.

As the plot illustrates, the algorithm alternates between local refinement phases using 2-opt and diversification phases driven by a variable number of 3-opt kicks. The current solution cost (blue) exhibits significant fluctuation, especially during early iterations, as the algorithm explores different regions of the solution space. These fluctuations are a natural consequence of aggressive perturbations, which often increase the solution cost before re-converging toward local optima.

The number of 3-opt kicks applied (green, right axis) tends to grow over time as the algorithm adapts to longer periods without improvement. This allows for progressively broader exploration of the solution space. Periodic improvements to the best solution (orange), marked by red crosses, highlight successful escapes from local minima.

While this run was not intended to produce optimal results, it clearly shows the iterative interaction between intensification and diversification mechanisms that define the VNS approach.

Chapter 4

Exact Methods

Despite the promising results achievable with a well-designed heuristic, it is necessary to consider strategies aimed at finding the optimal solutions.

Before going into the implementation of exact methods, we must first examine the problem formulation introduced in Section 1.1, particularly the Subtour Elimination Constraints (SECs) in Equation 1.3. It is important to highlight that the number of SECs grows exponentially with respect to the number of nodes in the instance, making it infeasible to generate all of them upfront, even for problems of moderate size.

This section presents the implementation of exact methods whose goal is to generate SECs in a more intelligent and efficient manner, leveraging CPLEX to solve the resulting mathematical model.

4.1 CPLEX

CPLEX is a high-performance solver for linear programming (LP), mixed-integer programming (MIP), and quadratic programming problems. It is widely used in both academic research and industrial applications due to its efficiency, scalability, and support for advanced solving techniques. [6]

It provides a rich set of APIs, allowing for integration into custom optimization workflows, along with powerful features like presolving, cutting planes, heuristics, and branch-and-bound algorithms. One of the most useful functionalities of CPLEX for combinatorial optimization problems, such as the Traveling Salesman Problem, is the support for *callbacks*. These allow users to interact with the solver during the optimization process, for example, by adding constraints dynamically (lazy constraints) or customizing branching decisions.

In this work, CPLEX is used to solve the integer linear programming (ILP) formulation of the TSP, taking advantage of its support for dynamic constraint generation to efficiently handle the exponential number of Subtour Elimination Constraints.

4.2 Benders Decomposition

Benders Decomposition is a general technique for solving large-scale mixed-integer problems by iteratively decomposing them into a master problem and one or more subproblems.

In our implementation, the main idea is to:

- Solve a relaxed version of the ILP model without SECs.
- Analyze the resulting solution to detect subtours (disconnected components).
- Add one or more SECs to eliminate these subtours.
- Repeat the process until a connected solution is found.

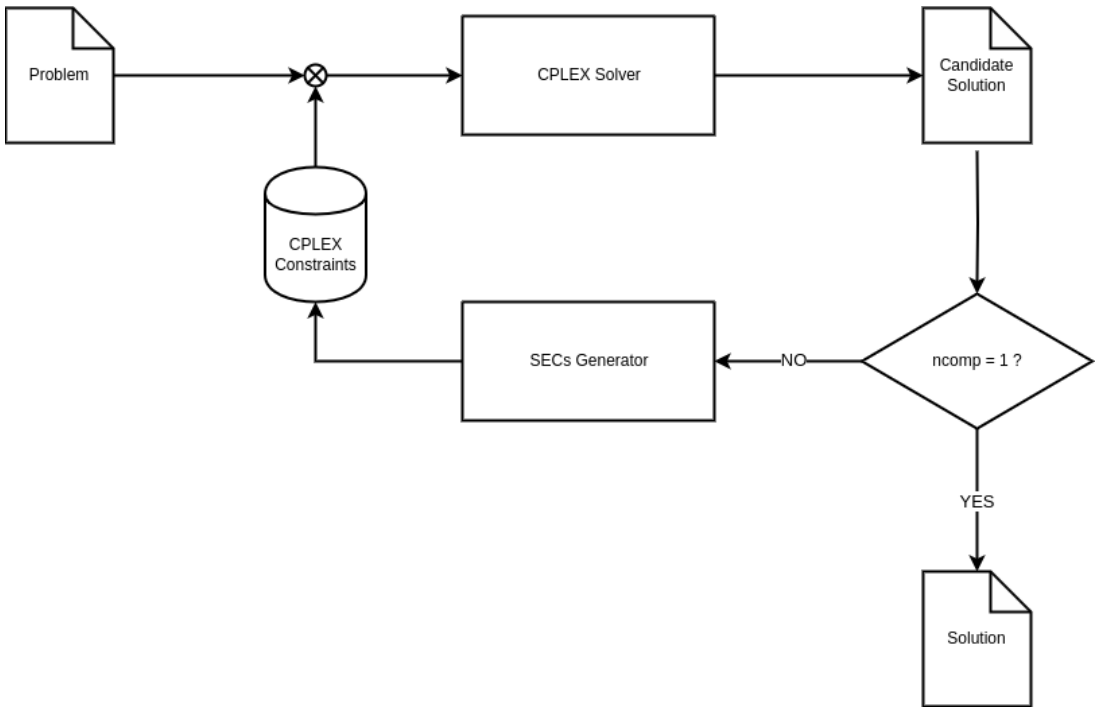


Figure 4.1: Flowchart of the Benders-like solving loop for the TSP.

4.2.1 Cycle Detection and SEC Addition.

The solution provided by CPLEX after each iteration is typically fractional or disconnected. To restore feasibility, we apply a component detection procedure: all edges x_{ij} such that $x_{ij} > 0.5$ are considered part of the current solution, and a union-find-like structure is used to identify the connected components.

For each component C_k, we add the following constraint:

$$\sum_{\substack{i,j \in C_k \\ i \neq j}} x_{ij} \leq |C_k| - 1$$

This ensures that in future iterations, such subtours will be avoided.

4.2.2 Pseudocode

Algorithm 6 Subtour Elimination Loop

Initialize `comp[i] = -1` for all nodes i, and set `ncomp = 0`
repeat
    Solve the relaxed ILP using `CPXmipopt`
    Extract solution and build components from $x^*_{ij} > 0.5$
    `ncomp` $\leftarrow$ number of connected components
    if $ncomp > 1$ **then**
        for each component C_k **do**
            Add SEC: $\sum_{i,j \in C_k} x_{ij} \leq |C_k| - 1$
        end for
    end if
until `ncomp` $= 1$ or time limit reached

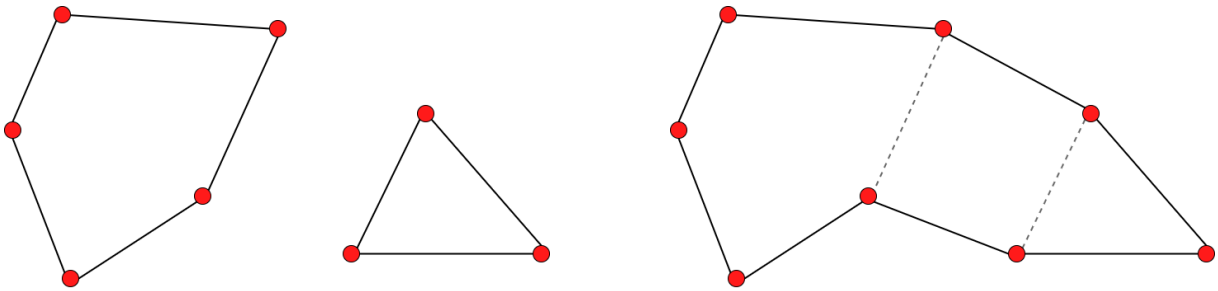
The entire loop is managed manually, CPLEX is invoked multiple times, at each iteration with an updated model that includes new constraints.

4.2.3 Patching Heuristic

In cases where the solving loop exceeds the time limit or fails to converge to a single connected component, we introduce a fallback heuristic called *patching*, which aims to merge disconnected subtours into a feasible Hamiltonian cycle.

The algorithm works as follows:

- Identify all components (subtours) in the current solution.
- For each pair of components, consider removing an edge from one component and connecting its endpoint to a node in the other component.
- Compute the extra cost for each reconnection and select the pair that minimizes the increase in total cost.
- Repeat the operation until all components are merged into a single tour.



(a) Disconnected subtours (MIP solution) **(b)** Final tour after patching

Figure 4.2: Visual example of the patching heuristic. The initial MIP solution contains multiple disconnected cycles (a), which are then merged into a single feasible tour (b) using edge reconnection.

Finally, once a patched tour is constructed, a local optimization phase (e.g., 2-opt) can optionally be applied to refine the solution quality. Obviously, this solution does not guarantee optimality, but it is a reliable fallback in case the Benders loop does not have enough time to reach the global minimum.

4.3 Branch-and-Cut

Branch-and-Cut is a state-of-the-art exact method for solving the TSP and many other combinatorial problems. It extends the classical Branch-and-Bound algorithm by dynamically adding valid inequalities (cuts) to the model, such as Subtour Elimination Constraints, during the exploration of the branch-and-bound tree.

In our implementation, we leverage CPLEX's built-in support for lazy constraint callbacks to detect and eliminate subtours on-the-fly. This allows us to start from a relaxed ILP formulation and enforce connectivity only when needed, without pre-generating the exponential number of SECs.

4.3.1 Lazy Constraint Mechanism

Rather than rebuilding the model after each solution, Branch-and-Cut integrates subtour elimination directly into the solving loop through a callback mechanism. After each integer-feasible solution is identified by the solver, a callback is triggered to verify whether the current edge selection forms a valid Hamiltonian cycle. This is achieved by analyzing the connected components of the solution. If multiple subtours are found, one or more violated SECs are generated and dynamically injected into the model as lazy constraints.

This logic can be visualized using the flow in Figure 4.1, with the only difference being that SECs are injected directly into the CPLEX solver, and the model is not rebuilt from scratch at each iteration.

This interaction happens entirely within the solver, reducing overhead and fully exploiting CPLEX's branch-and-bound engine. In our implementation, this is realized by registering a lazy constraint callback under the candidate solution context, which ensures

that only integer-feasible solutions are inspected. The integration is fully transparent to the solver's internal mechanisms, preserving performance.

4.3.2 Pseudocode

Algorithm 7 Callback-based Subtour Elimination

Define lazy constraint callback function:
    Extract x_{ij}^* from current solution
    Build connected components from $x_{ij}^* > 0.5$
if `ncomp` > 1 **then**
    for each component C_k **do**
        Add SEC: $\sum_{i,j \in C_k} x_{ij} \leq |C_k| - 1$
    end for
end if
Attach callback to CPLEX model
Solve the ILP using `CPXmipopt`

The callback logic relies on identifying connected components from the edge variables x_{ij} using a simple traversal procedure. Subtours are detected efficiently and corresponding SECs are added using CPLEX's lazy constraint API, avoiding the need to rebuild the model.

This structure ensures that any subtour detected during the search is immediately eliminated, allowing the solver to explore only feasible (or corrected) branches.

Chapter 5

Matheuristic Methods

In this section, we introduce a class of techniques known as *matheuristics*, which aim to combine the efficiency of heuristic methods with the mathematical models used in the exact methods presented in Chapter 4.

This hybrid approach has proven effective across a wide range of combinatorial problems, often requiring minimal problem-specific adaptation. The term *matheuristics* was originally coined during a workshop held in Bertinoro, Italy, in 2006, to describe the growing number of strategies situated at the intersection between classical optimization and heuristic design.

In the context of the TSP, matheuristics provide a valuable compromise: they allow us to exploit high-quality heuristic solutions while still benefiting from the precision and structure of mathematical models. In the next sections, we will focus on two such strategies: *Hard Fixing* and *Local Branching*.

5.1 Hard Fixing

Hard Fixing is a matheuristic strategy designed to balance the exploration power of mathematical programming with the speed of heuristics. The method is based on iteratively solving reduced Mixed Integer Programming (MIP) models where a subset of the decision variables is fixed, while the remaining ones are left free for optimization.

The process begins with the construction of an initial feasible tour; this solution serves as the starting point for the fixing phase. At each iteration, a percentage of edges from the current best solution is selected and fixed — that is, their corresponding variables are constrained to be part of the tour ($x_{ij} = 1$). The remaining variables are left unfixed, and CPLEX is invoked to solve the restricted problem with a reduced time limit.

Edges are selected for fixing probabilistically: each edge in the tour has a fixed probability (e.g., 30%) of being selected. This randomized selection introduces diversification, allowing the algorithm to explore different neighborhoods at each iteration.

After solving the subproblem, if a better solution is found, it replaces the current best, and the process repeats until the global time limit is reached. If the resulting solution is

disconnected, a fallback patching heuristic is invoked to merge multiple subtours into a single tour, as previously described in Section 4.2.3.

The main benefits of Hard Fixing lie in its ability to guide the solver toward promising regions of the search space without solving the full model from scratch. This makes it particularly suitable for medium-to-large TSP instances where exact methods alone may not converge in reasonable time.

5.1.1 Pseudocode

Algorithm 8 Hard Fixing Matheuristic

Generate initial solution x^0 using a greedy heuristic
$x^{\text{best}} \leftarrow x^0$
repeat
    Select subset $E_{\text{fix}} \subseteq \{e \in E : x_e^{\text{best}} = 1\}$ with fixed probability p
    Fix variables $x_e = 1$ for all $e \in E_{\text{fix}}$
    Solve the restricted MIP with CPLEX and time limit τ
    if solution is feasible and better than x^{best} **then**
        $x^{\text{best}} \leftarrow x$
    else if solution is disconnected **then**
        Apply patching heuristic to obtain a feasible tour
    end if
until global time limit is reached
return x^{best}

5.2 Local Branching

Local Branching is a matheuristic strategy that refines an existing solution by exploring its neighborhood through a restricted MIP model. Rather than using traditional local search operators such as 2-opt or 3-opt—whose complexity can grow exponentially with neighborhood size—this technique introduces a constraint that defines a neighborhood implicitly, allowing the solver to search within it efficiently.

Given a reference solution x^h, the local branching neighborhood $N(x^h)$ is defined as the set of solutions that differ from x^h in at most k variables. In the context of the TSP, this is typically implemented by enforcing the following constraint:

$$\sum_{(i,j)\in E: x_{ij}^h = 1} x_{ij} \geq n - k$$

This forces the solver to retain at least $n-k$ of the edges from the reference tour, effectively limiting the exploration to a region close to x^h.

The parameter k controls the size of the neighborhood:

- $k = 0$ leads to a fully fixed tour (no flexibility),
- $k = n$ allows for more exploration but with less structure,
- intermediate values like $k = 20$ often balance intensification and diversification effectively.

The method begins with an heuristic solution and iteratively solves restricted models using CPLEX. At each iteration:

- If a better solution is found, k is reset to its initial value.
- If no improvement is found, k is increased to allow broader exploration.
- The search is bounded by a global time limit, and each subproblem is further limited by a maximum number of branch-and-bound nodes.

If the resulting solution is disconnected, the fallback patching heuristic presented in Section 4.2.3 is applied.

5.2.1 Pseudocode

Algorithm 9 Local Branching Matheuristic

Generate initial solution x^0 using a greedy heuristic
$x^{\text{best}} \leftarrow x^0$
Set initial neighborhood size $k \leftarrow k_{\min}$
repeat
    Build local branching constraint centered at x^{best} with size k
    Solve the restricted MIP with node/time limits
    if a better solution x is found **then**
        $x^{\text{best}} \leftarrow x$, reset $k \leftarrow k_{\min}$
    else if solution is disconnected **then**
        Apply patching heuristic
    else
        Increase k (diversification)
    end if
until global time limit is reached or $k > k_{\max}$
return x^{best}

Chapter 6

Results

In this section, we present and analyze the results obtained from the experiments conducted on the methods introduced in the previous chapters. For each macro-category, two algorithmic approaches were implemented. The analysis is divided into two phases: first, the main parameters are fine-tuned (when applicable), and then the best-performing configurations within each category are compared.

6.1 Methodology

Each group of algorithms is analyzed independently, following a consistent methodology. Three main parameters are considered throughout the experiments:

- **Number of nodes:** This varies depending on the method's nature (e.g., fewer nodes for exact methods).
- **Time limit:** A fixed time budget is set for all experiments to ensure comparability.
- **Number of runs:** Each configuration is tested on 20 randomly generated instances to ensure statistical significance.

Specifically, the following settings were adopted:

- **Heuristics:** 1000-node instances, 120-second time limit.
- **Metaheuristics:** 1000-node instances, 120-second time limit.
- **Exact methods:** 300-node instances, 120-second time limit.
- **Matheuristics:** 1000-node instances, 180-second time limit.

Each test was repeated on 20 instances, initialized with distinct random seeds. To enable rigorous comparisons, we rely on **performance profiles**, a graphical benchmarking tool that allows us to assess and visualize the relative efficiency of different algorithms across a set of problem instances [7].

6.1.1 Instance Generation

The TSP instances used in our experiments are generated randomly. Each instance consists of a set of points placed on a 2D grid $[0, 10\,000] \times [0, 10\,000]$, with coordinates sampled from an independent and identically distributed model. The cost matrix for each instance is computed using the Euclidean distance between points, optionally rounded according to the ATT formula from TSPLIB [8].

6.1.2 Experimental Setup

All the methods have been implemented in C and the Exact and Matheuristic methods are built on top of **IBM ILOG CPLEX 22.1.2** [6], interfaced via its C API. Python was used for result visualization and data processing.

To ensure comparability, all algorithms within the same category were executed sequentially on the same hardware. However, different classes of methods were executed on different machines, as detailed below:

Processor	**RAM**	**Algorithms Executed**
Intel Core i7-10510U	16 GB	Exact, Matheuristic
Intel Core i5-8265U	16 GB	Greedy, Heuristic

6.1.3 Performance Profiles

To compare algorithm performance across a common set of instances, we used performance profiles. These were generated using a Python script developed by Professor Domenico Salvagnin. The script processes a CSV file containing the results of each algorithm on each instance and compares the performance to the best value achieved on that instance.

This approach enables the construction of plots that show, for each algorithm, the proportion of instances for which its performance is within a given factor of the best. For heuristic and metaheuristic methods, the comparison is based on the objective value of the best solution found; for exact methods, it is based on the time required to find the optimal solution.

6.2 Parameters tuning

Before comparing each method against another, we first explore a range of possible configurations for each algorithm to identify those that perform best under our fixed time and node constraints. This process follows the experimental methodology described above.

6.2.1 Nearest Neighbor - Greedy

In the case of Nearest Neighbor method we don't have any parameters to tune, but we can analyze if the Two-Opt step is beneficial or not. It is intuitive from the definion presented in section 2.2 that the 2-opt step is always beneficial, as it can only improve the quality of the tour. For completeness, we report the performance profiles of the Nearest Neighbor method with and without the 2-opt step in Figure 6.1.

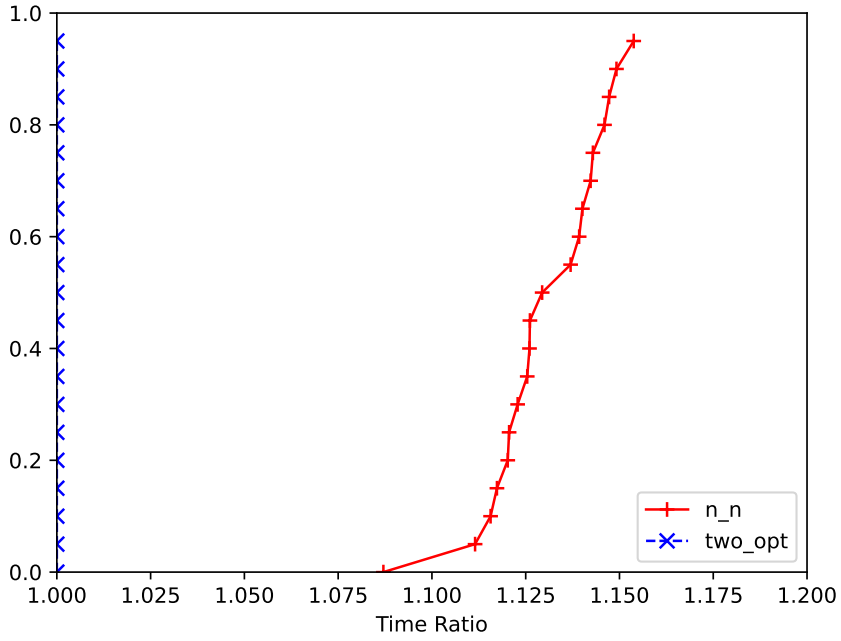


Figure 6.1: Performance profiles for the Greedy method.

6.2.2 Tabu Search

As discussed in Section 3.1, Tabu Search relies on a short-term memory of forbidden moves, whose size is governed by the base tenure T. In many works one selects a single ratio α and sets $T = \frac{N_{\text{nodes}}}{\alpha}$, but this hides the fact that the method can be tuned more flexibly by independently controlling the dynamic bounds $T_{\min}$ and $T_{\max}$.

In our implementation we therefore let the user to specify all parameters directly as absolute values, allowing also for setting the cutoff for forced diversification, N_{ni}. Conceptually, however, it is still suggested to leverage the ratio α to derive the tenure dimensions T, $T_{\min}$ and $T_{\max}$.

Table 6.1 reports the eight configurations where we tried to span as many different search behaviors as possible on our instance.

Table 6.1: Tabu Search configurations

T	$T_{\min}$	$T_{\max}$	N_{ni}	**Description**
227	10	227	500	Good balance, but no expansion beyond the initial tenure.
250	10	375	500	Moderate diversification via expanded max tenure.
208	5	416	1000	Broad tenure span allows prolonged exploration before reset.
278	20	278	250	Fixed high tenure intensifies local moves with quick shake-up.
167	10	167	500	Small tenure forces rapid neighborhood changes.
333	10	500	500	Max tenure scaled to problem size supports deeper history.
100	5	500	200	Very short base tenure and low cutoff for intense shaking.
400	50	400	2000	Large tenure and high cutoff delay diversification for global search.

Results presented in Figure 6.2 highlight that configurations with a large base tenure and a wide dynamic span tend to converge more quickly to high-quality incumbents. In particular:

- *High base tenure*: retaining a long short-term memory before forcing diversification produces the best overall performance, by exploiting promising regions.
- *Wide dynamic window*: allowing the tabu tenure to expand and contract over a large interval helps escape local optima when progress stalls.
- *Medium settings*: a moderate base tenure with a moderately sized dynamic range provides a reasonable trade-off between intensification and diversification, but does not match the extremes.
- *Small or rigid tenure*: very short or fixed tabu lists lead to slower discovery of good solutions and a greater risk of stagnation.

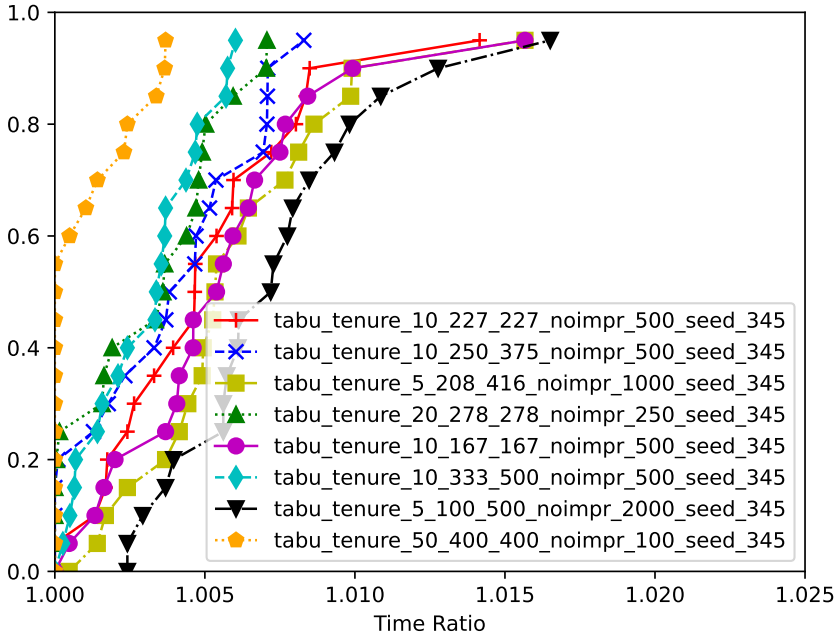


Figure 6.2: Performance profiles for the Tabu Search configurations.

6.2.3 Varable Neighborhood Search

To find the best configuration for the VNS algorithm, we focused on the parameters that control the number of 3-opt "kicks" applied per iteration. We let the user specify a fixed number of kicks J or an adaptive scheme that draws a number of kicks uniformly from a range $[k_{\min}, k_{\max}]$, scaled by a learning rate λ. Table 6.2 summarizes the settings we evaluated:

Table 6.2: VNS configurations

ID	$k_{\min}$	$k_{\max}$	λ	J	Description
1	2	4	1.00	-	Balanced adaptive kicks.
2	1	5	1.00	-	Wide range for broader exploration.
3	2	6	1.00	-	Extended range for more diversification.
4	1	10	1.00	-	Very wide span, high diversification.
5	2	4	0.50	-	Conservative kicks (half learning rate).
8	2	4	2.00	-	Aggressive kicks (double learning rate).
6	-	-	-	3	Fixed few kicks per iteration.
7	-	-	-	10	Fixed many kicks, conservative scale.

Results presented in Figure 6.3 suggest that a fixed number of kicks per iterations is more effective than adaptive schemes. In particular, the more kicks are applied the better the results, with the with the configuration applying 10 kicks outperforming all others.

Anyways, the adaptive schemes still perform reasonably well, especially when the range of kicks is wide enough to allow for sufficient exploration.

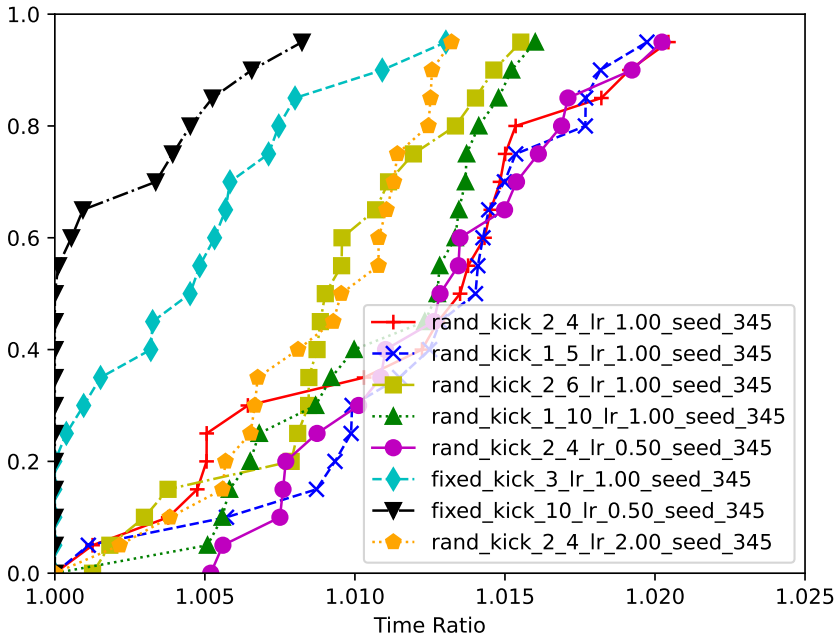


Figure 6.3: Performance profiles for the VNS configurations.

6.2.4 Hard Fixing

The Hard Fixing method depends on two key parameters: the proportion of edges to fix and the local time limit allocated to CPLEX for solving each subproblem. In preliminary experiments on 1,000-node instances with a global time limit of 180 s, we observed that fixing fewer than 50% of the arcs provided no appreciable benefit, as the CPLEX time limits were too restrictive. Consequently, we concentrated our study on fixing between 60% and 80% of the arcs. We also varied the local time limit to strike a balance between the number of iterations executed per run and the time CPLEX requires to solve each resulting subgraph, all proportionally scaled to the overall 180 seconds budget.

Figure 6.4 shows that fixing 70% of the arcs yields a more robust performance than the 60% configurations, while variations in the local time limit have only a marginal impact on the shape of the profile. To fine-tune the method, we therefore fixed the local time limit at 30 seconds, identified as the most robust among those tested, and extended our comparison to include the 80% fixing level.

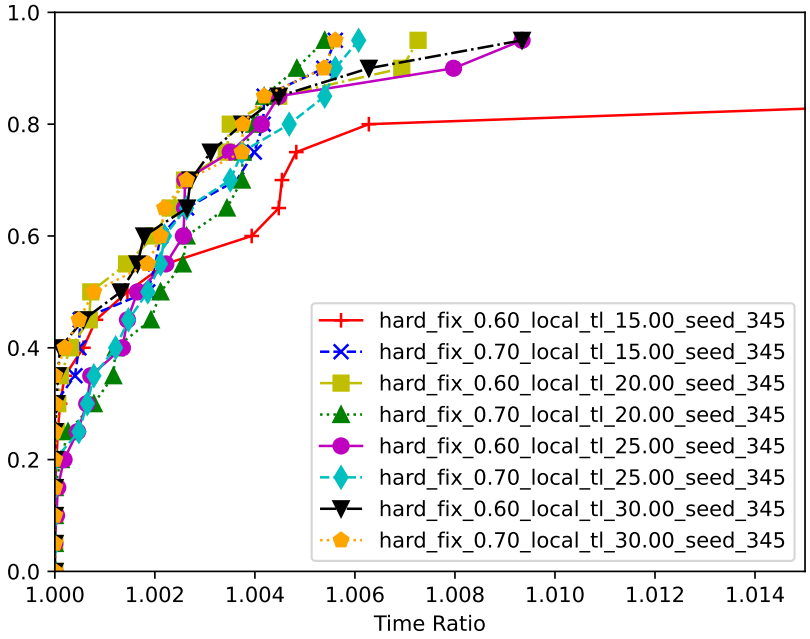


Figure 6.4: Performance profiles for Hard Fixing with varying fixing ratios and local time limits.

Figure 6.5 compares the three fixing ratios with the local time limit held constant at 30 second. It confirms that constraining the arcs too tightly degrades performance, whereas the 70% fixing level still achieves the highest coverage across those tested.

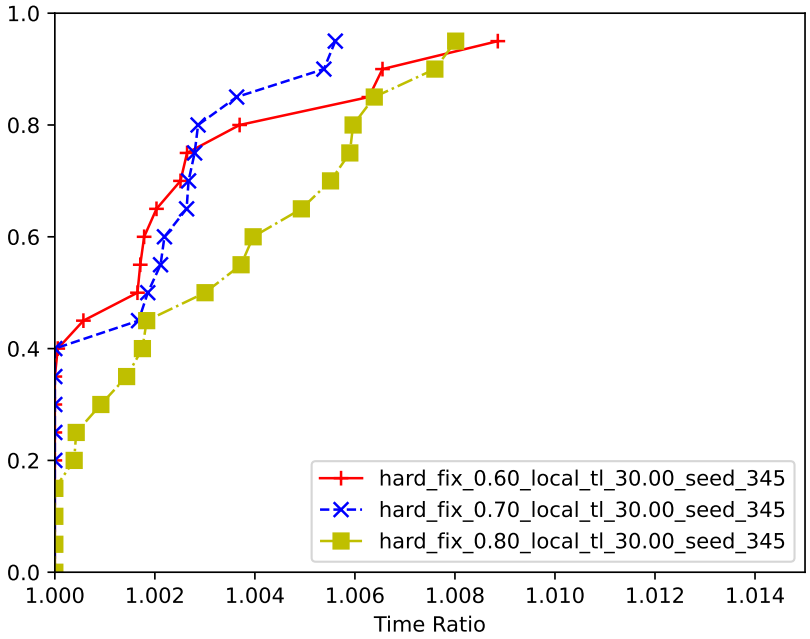


Figure 6.5: Performance profiles for Hard Fixing with a fixed 30 s local time limit.

Overall, the combination of fixing 70% of the arcs and allowing a 30 sec local solve time delivers the best trade-off between solution quality and robustness.

6.2.5 Local Branching

We evaluate the impact of two tuning parameters on the Local Branching heuristic: the initial neighborhood size K and the incremental step by which K is adjusted at each iteration. Figure 6.6 presents performance profiles comparing various (K, step) configurations under a fixed time budget.

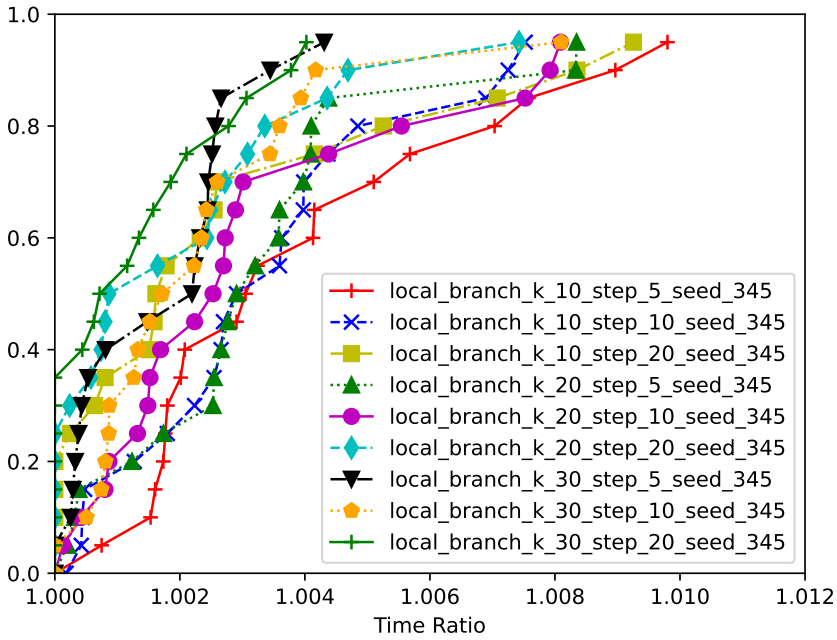


Figure 6.6: Performance profiles for Local Branching across different choices of K and step size.

The results show that larger neighborhoods yield more consistent improvements: configurations with $K = 30$ uniformly outperform those with $K = 20$ and $K = 10$. Likewise, a step size of 20 delivers the fastest early gains, although its marginal benefit diminishes over longer runs. Overall, the combination $K = 30$ and step $= 20$ achieves the best final costs under our time constraint.

6.3 Methods comparison

After introducing theoretically and tuning each method we're now ready to compare each family of methods to state whether one can be considered better than another. It is important to note that the results may be influenced by factors such as random seed selection, implementation quality, and hardware differences; thus, the findings should not be interpreted as absolute but rather as indicative of relative performance within our experimental framework.

6.3.1 Best Heuristic - Full time limit

After tuning the parameters of the Tabu Search and VNS algorithms we now compare their performance to determine the best heuristic for solving the TSP. We assume that the experiments performed during the parameters tuning phase cover a wide range of search behaviors and thus considering the four best configurations of each method is sufficient to draw conclusions on the overall performance of the two methods.

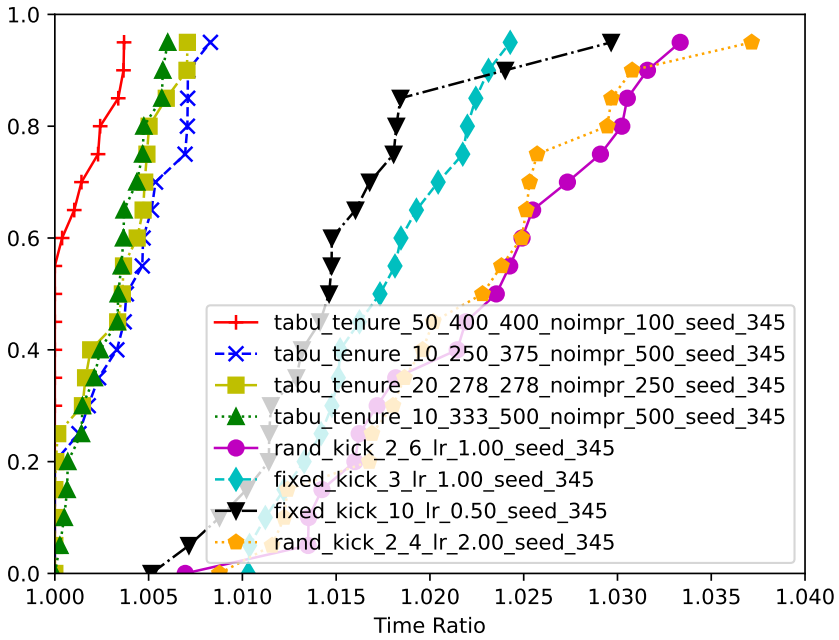


Figure 6.7: Performance profiles to compare VNS and Tabu Search.

Looking at Figure 6.7 there is no doubt on the superiority of the Tabu Search algorithm, the method configurations used in the comparison cover different search behaviours and all outperform the VNS configurations.

One more interesting comparison is to check how an approach as simple as the Nearest Neighbor with 2-opt compares to the best heuristic. The results are shown in Figure 6.8 and clearly show that the complicated and time-demanding search performed by Tabu is justifide by the quality of the solutions found, which is significantly better.

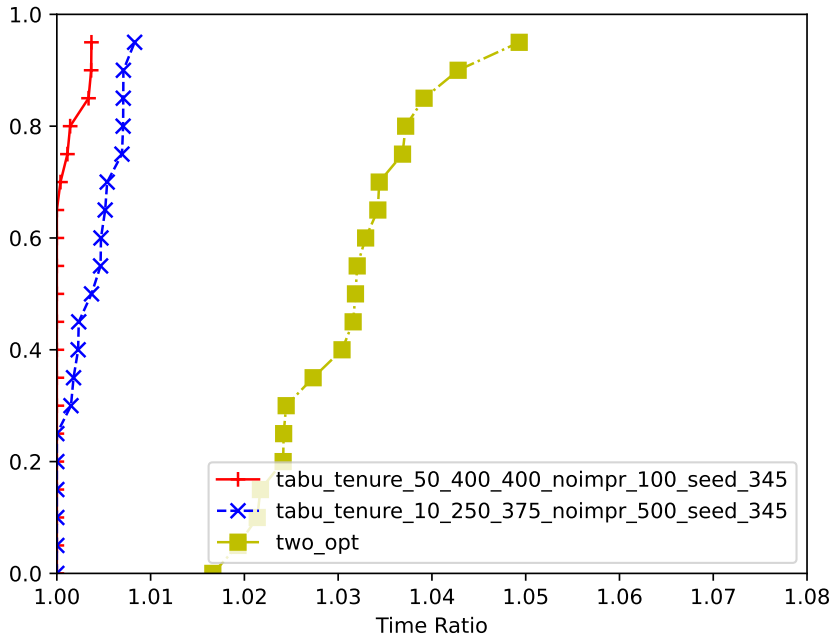


Figure 6.8: Tabu Search compared to Nearest Neighbor with 2-opt.

6.3.2 Best Heuristic - Short Time Budget

All CPLEX-based exact methods are initialized with a heuristic solution obtained by allocating 10% of the total time limit to the heuristic. To identify the most effective warm-start configuration, we therefore evaluate our top Tabu Search settings using a reduced time budget of 12 seconds. The best-performing configuration from this experiment will be used to warm-start the exact methods, which are compared in the next chapter. Since the instances of exact methods will be smaller, we are now searching for percentages of the number of nodes to set the tenure size, rather than absolute values.

From Figure 6.9 we can see that the best-performing configuration now is the one with a tenure size of around 28% of the number of nodes and a minimum size of 2%. This is a sigificant difference from the previous experiments, where the bigger tenure size was the best, this is likely due to the fact that with a short time limit the algorithm needs to explore more and thus a smaller tenure size allows for more diversification.

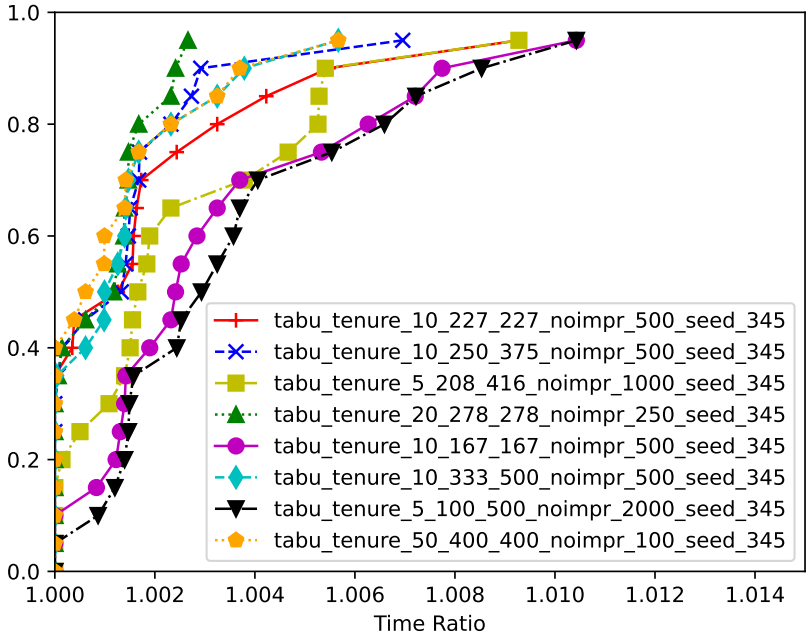


Figure 6.9: Tabu Search performance profiles under a 12-second time limit.

6.3.3 Best Exact Method

This section compares the Exact Methods introduced in Chapter 4. While these methods do not involve any fine-tuning, we still explore variations in their configurations to evaluate performance differences. Specifically, we examine three different approaches for initializing the solver:

- **No Warm Start**: In this baseline approach, CPLEX performs the optimization from scratch without any preprocessing or initial solution.
- **Heuristic Warm Start**: As discussed in Section 6.3.2, we identified the best configuration for a time-limited heuristic. The solution obtained from this heuristic is used as the starting point for the exact solver.
- **Two-Opt Warm Start**: We observed that a simple greedy solution refined with the two-opt technique yields a reasonable solution in under one second. Although typically worse than the heuristic solution, we aim to test whether this quick-start approach can help the exact method reach optimality faster.

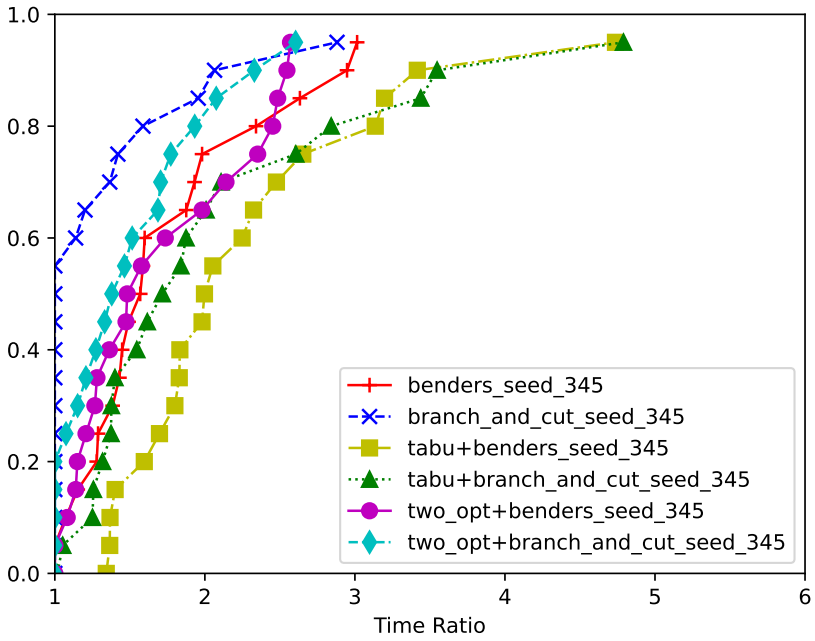


Figure 6.10: Exact Methods performance profiles.

Figure 6.10 shows the performance profiles of the exact solver under different initialization strategies. The standard **Branch-and-Cut** method performs best overall, solving the highest fraction of instances quickly and consistently across all time ratios. Its variant initialized with a **Two-Opt** solution performs comparably in the long run but lags slightly in early time windows, indicating that the warm start introduces some initial overhead without clear benefits in this setting.

In contrast, **Two-Opt** offers a more tangible advantage when paired with **Benders**. While it initially solves slightly fewer instances than plain Benders, it eventually overtakes it, demonstrating that lightweight local search can improve convergence over time.

Warm starts based on the **Tabu** heuristic consistently underperform. Whether combined with Benders or Branch-and-Cut, these configurations solve fewer instances across all time ratios, likely due to the higher overhead of Tabu not being compensated by solution quality.

Overall, default configurations—particularly Branch-and-Cut—remain highly effective. Lightweight warm starts like Two-Opt may be beneficial, especially for Benders, but more expensive ones like Tabu tend to degrade performance.

6.3.4 Best Matheuristic

In this section we compare the matheuristic methods from Chapter 5 after parameter tuning. Figure 6.11 clearly shows that Hard Fixing clearly outperforms Local Branching.

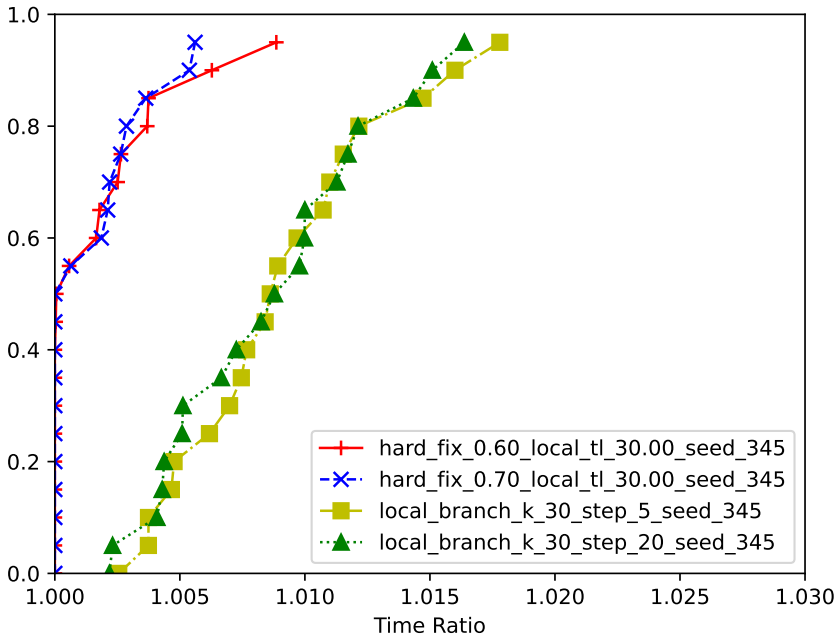


Figure 6.11: Performance profiles for the tuned matheuristics.

These results can be attributed to Hard Fixing's strategy of locking in high-quality variable assignments, which focuses the solver on the most promising regions of the search space and produces strong incumbents much faster than the incremental neighborhood adjustments of Local Branching.

Chapter 7

Conclusions

In this thesis, we have explored a wide range of algorithmic approaches to solve the Travelling Salesman Problem, including heuristic, metaheuristic, exact, and matheuristic methods. The objective was not only to assess the quality of the solutions generated by each method but also to evaluate their computational efficiency across different instance sizes and time budgets.

Among the heuristic approaches, the Nearest Neighbor algorithm combined with the 2-opt local search proved to be an effective baseline, offering rapid improvements with minimal computational cost. However, its limitations became evident when compared to more sophisticated techniques. In the field of metaheuristics, Tabu Search emerged as the most effective strategy. Thanks to its adaptive memory-based mechanism and dynamic tenure control, it consistently outperformed Variable Neighborhood Search, particularly in scenarios with extended time budgets. The performance profiles confirmed its ability to produce high-quality solutions across a broad range of instances, justifying the added complexity of its design.

Regarding exact methods, the Branch-and-Cut algorithm demonstrated superior performance and robustness compared to Benders Decomposition. While warm starts based on Tabu Search did not yield significant improvements-likely due to their overhead-lighter strategies such as 2-opt proved to be beneficial, especially in combination with Benders.

The matheuristic strategies offered an interesting compromise between heuristic guidance and mathematical rigor. Among them, Hard Fixing achieved the best results by effectively narrowing the search space while allowing diversification through probabilistic fixing schemes. Local Branching, while conceptually appealing, showed more limited effectiveness in comparison. Overall, the results highlight the advantages of hybrid and adaptive approaches in solving NP-hard problems like the TSP.

Future work could investigate the integration of machine learning techniques for automatic parameter tuning, such as Optuna [9], or explore new combinations of methods to further enhance performance on large-scale instances.

Bibliography

[1] Norman L. Biggs, E. Keith Lloyd, and Robin James Wilson. *Graph theory, 1736-1936*. Clarendon Press, 1986. ISBN: 0198539169.

[2] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[3] David Applegate et al. "The Traveling Salesman Problem: A Computational Study". In: *The Traveling Salesman Problem: A Computational Study* (2006), p. 606.

[4] Fred Glover and Manuel Laguna. *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers, 1997. ISBN: 079239965X.

[5] Pierre Hansen and Nenad Mladenovic. "Variable Neighborhood Search Methods". In: vol. 22. Jan. 2009, pp. 3978–. ISBN: 978-0-387-74758-3. DOI: 10.1007/978-0-387-74759-0_694.

[6] IBM Corporation. *IBM ILOG CPLEX Optimization Studio, version 22.1.2*. Available at https://www.ibm.com/products/ilog-cplex-optimization-studio. 2023.

[7] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Mathematical Programming* 91.2 (2002), pp. 201–213. DOI: 10.1007/s101070100263.

[8] Gerhard Reinelt. *TSPLIB 95*. Technical Report. Universität Heidelberg, 1995. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf.

[9] Optuna Development Team. *Optuna: A hyperparameter optimization framework*. Version 4.4.0. Optuna Contributors. 2025. URL: https://optuna.readthedocs.io/en/stable/.