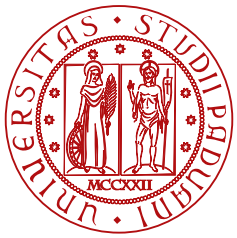


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN COMPUTER ENGINEERING

Exact and Heuristic Techniques for the Traveling Salesman Problem: A Comparative Analysis

Operations Research 2

Umberto Bianchin 2139222

Francesco De Nicola 2160329

ACADEMIC YEAR 2024-2025

Contents

Chapter 1

Introduction

The Traveling Salesman Problem (TSP) is a combinatorial optimization problem that has been studied by mathematicians, computer scientists, and operations researchers for decades. As mentioned in the book by Daventra, the TSP can be defined as follows: *"Given a set of cities and the cost of travel (or distance) between each possible pairs, the TSP, is to find the best possible way of visiting all the cities and returning to the starting point that minimize the travel cost (or travel distance)"* [7].

In this work, we implemented and compared three families of algorithms for the TSP:

- **Heuristic algorithms** (e.g. Nearest Neighbor, Extra-Mileage), which are fast but do not guarantee optimality;
- **Metaheuristics** (e.g. GRASP, Variable Neighborhood Search, Tabu Search), which employ randomized procedures and adaptive memory to balance exploration and exploitation;
- **Exact methods** (Branch-and-Cut, Benders decomposition), based on integer linear programming formulations and dynamic generation of sub-tour elimination constraints.
- **Matheuristics** (Hard-Fixing, Local Branching), which provide a powerful compromise by embedding heuristic principles within a mathematical programming framework.

To ensure a fair comparison, all implementations share: a common data structure for representing TSP instances and solutions; a performance profiler that constructs Dolan–Moré performance profiles on a standard benchmark set; identical test instances with fixed time limits and parameter settings.

The document is organized as follows:

1. **Chapter 2** introduces the Traveling Salesman Problem, presenting its formal definition, mathematical formulation, and some history.

2. **Chapter 3** describes the performance profiling framework and benchmarking methodology.
3. **Chapter 4** introduces the design and implementation of the TSP data structures.
4. **Chapter 5** presents the heuristic algorithms along with their pseudocode.
5. **Chapter 6** details the implementation and parameter tuning of the metaheuristics.
6. **Chapter 7** covers the integer programming models, use of CPLEX, warm starts, and dynamic cut generation.
7. **Chapter 8** explores matheuristics that combine heuristic and exact techniques.
8. **Appendix A and B** provide installation instructions for CPLEX and some utility functions.

After each chapter, we report experimental results and comparative analyses to highlight the best algorithms to use for instances like the one we tested with. All the source code described in this work can be found on: `https://github.com/umberto-bianchin/OperationsResearch2`

Chapter 2

TSP Preliminaries

2.1 Problem Definition

The Traveling Salesman Problem can be formally defined as follows: given a set of n cities and the distances between each pair of cities, the TSP consists of finding the shortest route, starting from an initial city, that visits each city exactly once and returns to the starting city.

The TSP is an NP-hard problem, meaning that no known efficient (polynomial-time) algorithm exists to optimally solve it for large instances. Consequently, research has focused on developing approximation algorithms and heuristics that provide good-quality solutions in a reasonable amount of time.

The standard version of the TSP renders the asymmetric version of the problem, where the cost of traveling from city i to city j can be different from that of city j to city i. Instead, in this work we focus on the symmetric Traveling Salesman Problem (TSP), that can be defined as follows: given a complete undirected graph $G = (V, E)$, where:

- $V = \{v_0, v_1, \ldots, v_{n-1}\}$ is the set of vertices (cities).
- E is a family of unordered pairs of elements of V, is the set of edges (arcs).
- c_e is the cost (distance) associated with the edge e.

We want to find a Hamiltonian cycle (a tour) of minimal total cost. Let x_e be a binary variable defined as:

$$x_e = \begin{cases} 1 & \text{if edge } e \text{ is in the tour,} \\ 0 & \text{otherwise.} \end{cases}$$

The TSP can be formulated as an integer linear programming (ILP) problem:

Minimize: $\sum_{e \in E} c_e x_e$

Subject to:

$$\sum_{e \in \delta(v)} x_e = 2, \quad \forall v \in V \tag{1}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1, \quad \forall S \subset V : 3 \leq |S| \leq |V| - 1 \tag{2}$$

$$x_e \in \{0, 1\}, \quad \forall e \in E \tag{3}$$

Explanation of Constraints:

1. With $\delta(v)$ we denote the set of edges incident to vertex v. Each vertex is touched by exactly 2 edges, this ensures that the tour visits every node once.

2. Sub-tour elimination constraint (SEC): it prevents any smaller cycle confined to a subset S of vertices, forcing a single Hamiltonian cycle over all of V. $E(S)$ denotes the set of edges whose endpoints are both in S.

3. Binary constraints: each x_e is either 0 or 1.

The subtour elimination constraints are needed to ensure that the solution forms a single tour and not multiple disconnected subtours.

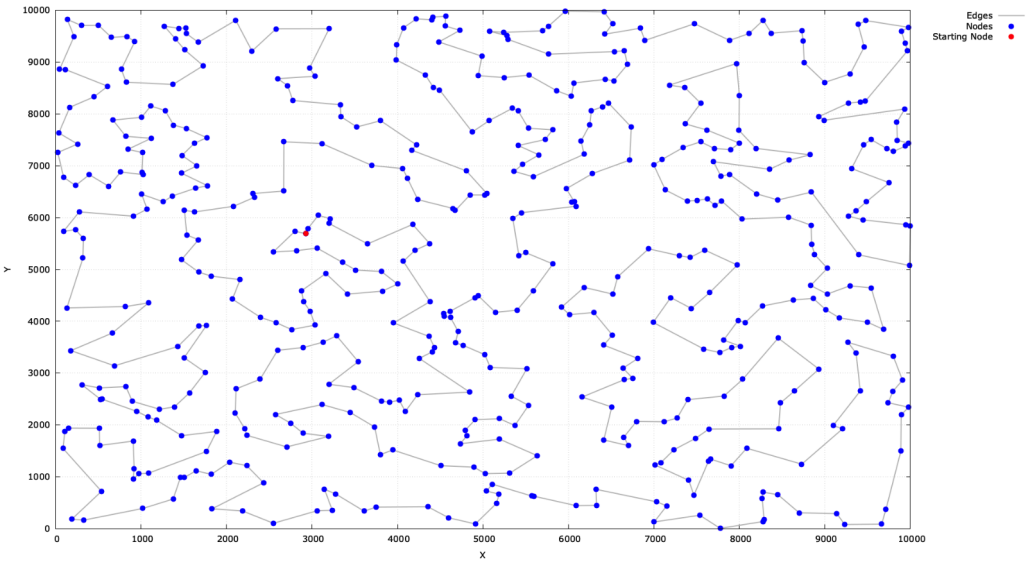


Figure 2.1: Example of a solution for a TSP instance

2.2 History of the TSP

Sir William Rowan Hamilton's 1856 *Icosian game* [12] challenged players to find a Hamiltonian cycle on a dodecahedral graph, making it the first popular puzzle linked to what we now call the Traveling Salesman Problem (TSP). Scientific interest took shape in the 1930s, when Karl Menger posed the "Messenger-boy problem" and matured with Dantzig, Fulkerson, and Johnson's 1954 cutting-plane tour of 49 U.S. cities, effectively inaugurating the modern study of the problem [6]. The contemporary era is characterized by the **Concorde** project of Applegate, Bixby, Chvátal, and Cook (1997–present). Concorde uses a powerful branch-and-cut algorithm, integrated with the CPLEX LP solver, to eliminate fractional solutions. It has successfully found optimal tours for all TSPLIB instances to date[1, 2]. TSPLIB is a widely used benchmark library of TSP and related combinatorial-optimization instances, originally assembled by Reinelt (1991) to standardize and facilitate the comparison of algorithmic performance across a diverse set of problem classes [18].

Chapter 3

Performance Profiler

3.1 Introduction

In order to compare the efficiency of different algorithms or to tune the various parameters of each algorithm, we needed a benchmarking methodology that accounts both for solver speed and robustness across different instances. Traditional aggregate statistics can be influenced by a small number of hard instances or by failed runs. To overcome these limitations, we employ a tool introduced by Dolan and Moré called *performance profiles* [8], which represents for each solver the cumulative distribution function of its runtime ratios relative to the best solver on each instance.

3.2 Performance evaluation

Let S be the set of solvers under consideration and P the set of benchmark problems. Denote by $t_{p,s}$ the CPU time (or other performance measure, like the cost of the optimal solution found by a heuristic algorithm) needed by solver $s \in S$ to solve instance $p \in P$. We define the *performance ratio* as

$$r_{p,s} = \frac{t_{p,s}}{min\{t_{p,s'} : s' \in S\}}$$

so that $r_{p,s} \geq 1$ and $r_{p,s} = r_{max}$ if solver s fails on p. The performance profile of solver s is the function

$$\rho_s(\tau) = \frac{1}{|P|}|\{p \in P : r_{p,s} \leq \tau\}|$$

which gives the proportion of the problems that solver s solves within a factor τ of the best solver. Key properties include:

- $\rho_s(1)$ is the fraction of instances on which s is the fastest;

- $lim_{\tau \to r^-_{max}} \rho_s(\tau)$ is the fraction of instances s eventually solves;
- profiles are insensitive to outliers (one instance can affect ρ by at most 1/—P—) and to small perturbations in runtimes.

By plotting $\rho_s(\tau)$ for all $s \in S$, one obtains a clear visualization of comparative performance across the entire benchmark; an example can be seen in figure 3.1, where `Alg5` is the best algorithm

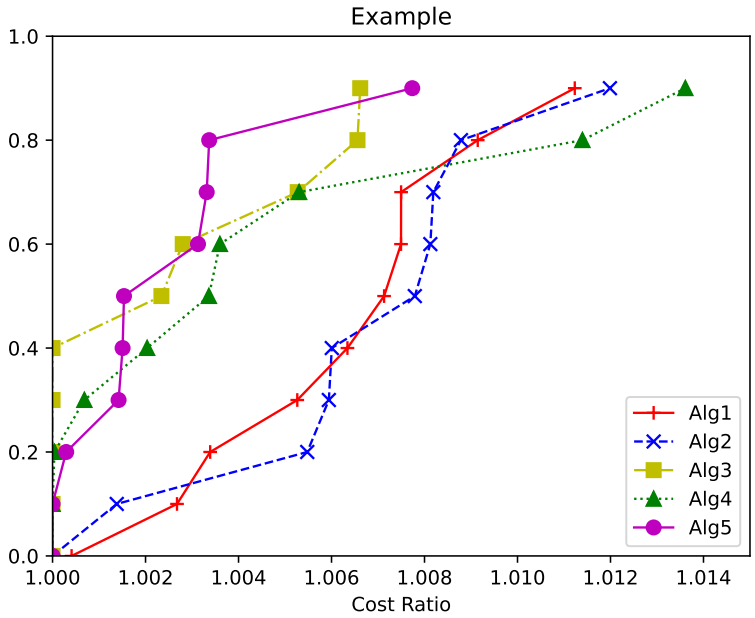


Figure 3.1: Example of performance profile

3.3 Implementation in Python

To automate the generation of performance-profile plots, we used a script developed by prof. Domenico Salvagnin from the University of Padua called *perfprof.py*. The script read a CSV file containing runtime data and produces a publication-quality PDF plot using Matplotlib. The CSV file must have the following structure:

- **First line:**

```
<ncols>, solver1, solver2, ..., solver_n
```

where `<ncols>` is the number of solver and each `solver_i` is the label for column i.

- **Subsequent lines:**

```
instance_name,t_1,t_2, ...,t_n
```

  where $t_{p,s}$ is the CPU time (or other performance metrics) required by solver s on instance p.

3.3.1 CSV Parser and Writer Implementation in C

To automate creation and updating of the CSV file, we implemented in C a custom CSV reader and writer. The key functions are:

- `int parse_csv(char ***csvData, char *fileName);` reads the existing CSV into a 3D array of strings, skipping the first column (instance name) and returning the number of data columns.
- `void write_csv(double bestCosts[], char *algorithmID, char alg);` appends a new column headed by `algorithmID` to the CSV and writes the updated table back to disk.

These routines allocate a `MAX_ROWS` $\times$ `MAX_COLS` buffer, tokenize each line on commas, and ensure proper null-termination and memory deallocation.

3.4 Application to TSP Experiments

We evaluated each solver on a suite of 10 pseudo-random generated TSP instances. Each instance was created with its own fixed random seed, which remained the same across all solver runs to ensure comparability.

We call `srand(inst->seed)` once before generating coordinates. Because `srand()` initializes the pseudo-random number generator with a given seed, every time you use the same `inst->seed`, the subsequent sequence of `rand()` values is identical.

Each node's x and y coordinate is drawn via

`rand() % MAX_COORDINATES`,

which yields an integer uniformly distributed in $[0, MAX\_COORDINATES - 1]$. Here

$$MAX\_COORDINATES = 10000$$

fixes the "play area" to a $10\,000 \times 10\,000$ square.

Heuristic algorithms

- **Number of nodes:** 1000 nodes per instance.
- **Time limit:** 60 s per instance.
- **Metric:** cost of the best tour found within the time limit.

Matheuristic algorithms

- **Number of nodes:** 1000 nodes per instance.
- **Time limit:** 180 s per instance.
- **Metric:** cost of the best tour found within the time limit.

Exact algorithms

- **Number of nodes:** 300 nodes per instance.
- **Metric:** time required to prove optimality.

To tune each set of parameters for each algorithm, we generated a separate CSV results file by changing its key parameters and then we plotted the performance profiles. After that, we re-run each algorithm with its best configuration and we created different files, one for the results of heuristic algorithms, one for the metaheuristic algorithms, one the exact algorithms and one for the matheuristic algorithms, and again we compared them by plotting the performance profiles.

Chapter 4

TSP Data Structure

Our implementation of the Traveling Salesman Problem is designed with three data structures that encapsulate all the necessary information about a TSP instance. These data structures serve as the basis for all algorithms, facilitating consistent data management, monitoring solution progress, and comparing results.

Solutions Collection

To track all the solutions found by any algorithm execution, we use a dynamic array structure:

```
typedef struct solutions_struct {
    double *all_costs;  // Array of costs for different solutions
    int size;           // Current number of solutions stored
    int capacity;       // Maximum capacity of the array
} solutions;
```

This structure enables us to track the solutions history and performance over time.

Solution Route

Each TSP solution is represented as:

```
typedef struct solution_struct {
    int *path;          // Array containing the node sequence
    double cost;        // Total cost of the tour
} solution;
```

A solution consists of a path (sequence of nodes visited) and its associated cost. This structure is used to allow multithread execution, each algorithm modify it's own solution accessing the main instance only if a better solution is found.

TSP Instance

This is the central data structure that holds all information related to a specific TSP problem:

```
typedef struct instance_struct {
    // Input data
    int nnodes;                     // Number of nodes
    double *xcoord;                 // Node X coordinates
    double *ycoord;                 // Node Y coordinates
    char algorithm;                 // Selected algorithm identifier
    char running_mode;              // Running mode flag
    int *params;                    // Algorithms parameters

    int ncols;                      // Variables in the CPLEX model

    // Configuration
    int seed;                       // Random seed
    char input_file[1000];          // Input file path
    int integer_costs;              // Flag for integer costs

    solution best_solution;         // Best route found so far
    double *costs;                  // Edge costs matrix

    double time_limit;              // Time limit in seconds
    double t_start;                 // Start time of computation

    solutions history_best_costs;   // Record of best costs found
    solutions history_costs;        // Record of all costs found
} instance;
```

- **Input Data:**
  - `nnodes`: The total number of nodes in the TSP instance.
  - `xcoord` and `ycoord`: Arrays of doubles that store the x and y coordinates of each node.
  - `algorithm`: A character indicating which algorithm is selected for solving.
  - `running_mode`: A character flag indicating the execution mode, it can be Normal, Benchmark, or CPLEX.
  - `params`: An integer array containing algorithm hyperparameters.

- **Configuration:**
  - `seed`: An integer used to initialize the random number generator.
  - `input_file`: A character array storing the name of the input file (if any).
  - `integer_costs`: An integer flag indicating whether costs should be treated as integers.
- **Solution Information:**
  - `best_solution`: Contains the best tour found and its cost.
- **Cost Matrix:**
  - `costs`: An array of doubles representing the cost (or distance) between each pair of nodes. The element in position (i, j) is the cost of the edge that starts from i and goes to j.
- **Time Constraints:**
  - `time_limit`: A double value specifying the maximum allowed computation time in seconds.
  - `t_start`: A double value that marks the starting time of the algorithm.
- **Solution History:**
  - `history_best_costs`: A solutions structure tracking the best costs found during execution.
  - `history_costs`: A solutions structure tracking all costs evaluated during execution.

4.1 Data Structure Design Considerations

The design provides several advantages for the implementation of the TSP algorithm.

1. **Multi-threaded execution safety**: The solution structure is designed to support the execution of parallel algorithms. Each algorithm works on its own independent solution structure and only updates the best global solution when necessary, minimizing thread synchronization requirements.
2. **Unified representation**: All algorithms operate on the same data structure, enabling fair comparison.

3. **Solution history**: The structure tracks all solutions and their costs, facilitating analysis of the convergence of the algorithms.

4. **Parameter flexibility**: Algorithm-specific parameters are stored in a unified array, allowing easy parameter tuning.

5. **Algorithm independence**: The core data structures are independent of specific algorithms, allowing new algorithms to be added easily.

Chapter 5

Heuristic

Heuristic algorithms, unlike exact methods which explore the full solution space to prove optimality, sacrifice completeness for speed and simplicity, making them indispensable when solving large instances or when rapid decision making is required. In this chapter, we present two construction heuristics (Nearest Neighbor and Extra Mileage) and a 2-OPT local-search refinement.

5.1 Nearest Neighbor Algorithm

The *Nearest Neighbor* algorithm is a greedy constructive heuristic that, at each step, appends to the current partial tour the closest unvisited node, repeating until every node is visited and the salesman returns to the start [19]. In our implementation, we start with an empty tour, then we add in the first and last positions the given initial node and iteratively choose the nearest node to add to the tour. At the end we compute the path cost and check if the tour is feasible.

Algorithm 1: Nearest Neighbor Algorithm

Input: TSP instance with cost matrix, starting node s

Output: Tour solution and its cost

Initialize array `visited` of size n with all entries set to 0;
solution[0] $\leftarrow s$, solution[$n-1$] $\leftarrow s$;
visited[s] $\leftarrow 1$;
for $i \leftarrow 1$ **to** $n-1$ **do**
    last_selected $\leftarrow$ solution[$i-1$];
    nearest_node $\leftarrow -1$, *min_cost* $\leftarrow \infty$;
    for $j \leftarrow 0$ **to** n **do**
        if *visited[j]* $= 0$ ***and*** *cost(last_selected, j)* $<$ *min_cost* **then**
            min_cost $\leftarrow$ cost(*last_selected*, j);
            nearest_node $\leftarrow j$;
        end
    end
    if *nearest_node* $\neq -1$ **then**
        solution[i] $\leftarrow$ *nearest_node*;
        visited[*nearest_node*] $\leftarrow 1$;
    end
end
free(visited);
Compute the total solution cost and check feasibility;

5.1.1 Multi-Start Approach

We extend the basic heuristic into a multi-start version by launching it from every node $s \in V$, which serves to diversify the search and increase the probability of finding high-quality tours. We monitor elapsed time to ensure the computation stays within the prescribed limit, after that we stop the computation and only the tours found within the time limit are considered.

Algorithm 2: Multi-Start Nearest Neighbor Algorithm

Input: TSP instance with cost matrix and time limit t

Output: Best tour solution found within time limit

```
for i ← 0 to n do
    Apply Nearest Neighbor procedure with starting node i;
    Check feasibility and update best cost if this solution is better;
    t2 ← current time in seconds;
    if t2 − instance.t_start > T then
        break;
    end
end
```

5.2 2-OPT

The *2-OPT* algorithm is a **refinement method** commonly used to improve solutions to the Traveling Salesman Problem (TSP). Introduced by Croes in 1958 [5], the method iteratively enhances a given tour by eliminating crossing edges while reducing the total travel distance. The fundamental idea is to replace two non-adjacent edges with two different edges that reconnect the tour in a shorter configuration.

Consider two edges in the tour: (π_i, π_{i+1}) and (π_j, π_{j+1}), with $0 \leq i < j - 1 < n$. Replacing these with the edges (π_i, π_j) and (π_{i+1}, π_{j+1}) results in a cost difference given by:

$$\Delta = \left[c_{\pi_i,\pi_j} + c_{\pi_{i+1},\pi_{j+1}}\right] - \left[c_{\pi_i,\pi_{i+1}} + c_{\pi_j,\pi_{j+1}}\right].$$

If $\Delta < 0$, the swap leads to a shorter tour. In that case, the segment between π_{i+1} and π_j is reversed to create a new valid tour. This process is repeated until no further improvements can be made (i.e., until no pair of edges can be swapped to reduce the tour cost).

Example of a 2-OPT Move

Consider the tour:

$$\pi = (0, 1, 2, 3, 4, 5, 0)$$

We select two non-adjacent edges:

$$(\pi_1, \pi_2) = (1, 2) \quad \text{and} \quad (\pi_4, \pi_5) = (4, 5)$$

The 2-OPT move will remove these two edges and reconnect the path using:

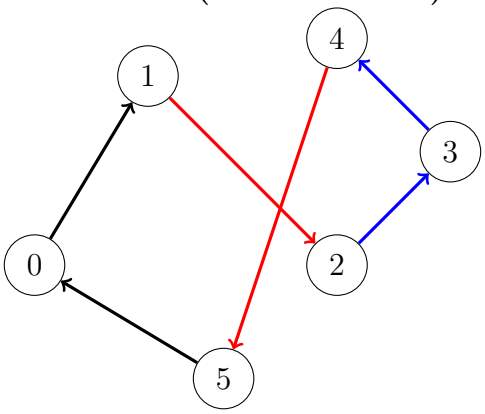
$$(\pi_1, \pi_4) = (1, 4) \quad \text{and} \quad (\pi_2, \pi_5) = (2, 5)$$

This requires reversing the subpath between π_2 and π_4, i.e. the segment $(2, 3, 4)$ becomes $(4, 3, 2)$.

The new tour becomes:

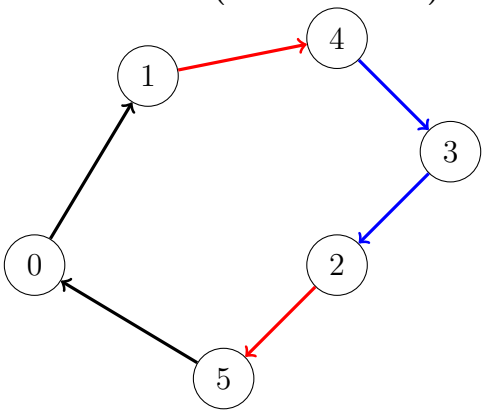
$$\pi' = (0, 1, 4, 3, 2, 5, 0)$$

Old Tour (Before 2-OPT)



The old tour before 2-OPT move

New Tour (After 2-OPT)



The new tour after 2-OPT move

In this visualization:

- The red edges are the edges that were changed during the 2-OPT move.
- The blue segment $(2, 3, 4)$ is reversed to $(4, 3, 2)$ in the new tour.

The 2-OPT algorithm starts with an initial feasible tour and iteratively improves it through local modifications. At each step, it examines pairs of non-adjacent edges and checks whether swapping them and reversing the intermediate path leads to a shorter tour. If an improvement is found, the tour is updated accordingly. This process continues until no further improvements can be made or until the time limit is reached. The algorithm is simple yet effective, and often leads to significant reductions in tour cost, especially when applied to a poor initial solution [15].

In our code we use the 2-OPT refinement in the multi-start nearest neighbor, after each solution is found.

Algorithm 3: 2-OPT Refinement Method

Input: TSP instance with cost matrix, time limit t

Output: Improved tour (if an improvement is found)

```
improved ← true;
while improved do
    improved ← false, min_delta ← ∞;
    for i ← 1 to n − 1 do
        for j ← i + 2 to n − 1 do
            δ ← calculate_delta(i, j);
            if δ < min_delta then
                min_delta ← δ;
                swap_i ← i, swap_j ← j;
            end
        end
    end
    if min_delta < 0 then
        reverse segment between i and j;
        update_solution_cost();
        set improved to true and check time-limit
    end
end
```

5.3 Extra-Mileage Algorithm

The *Extra-Mileage* (cheapest-insertion) heuristic iteratively inserts the city whose placement between two consecutive tour nodes minimizes the additional travel cost—called the "extra mileage" [19]. The heuristic prioritizes local decisions that cause the least disruption to the current tour, aiming to produce an efficient overall route.

The algorithm starts by selecting the pair of cities with the higher distance to initialize the tour. These two cities are marked as inserted. Then, at each iteration, the algorithm evaluates every non-inserted city and considers all possible positions it could be inserted between two consecutive cities in the current tour. It computes the increase in cost Δ that each possible insertion would cause:

$$\Delta = c_{a,h} + c_{h,b} - c_{a,b}$$

where a and b are two consecutive nodes in the tour, and h is a candidate node to insert. The algorithm selects the node and insertion point that yield the smallest Δ, performs the

insertion, and repeats this process until all cities are included. Finally, the tour is closed by connecting the last city back to the first.

Algorithm 4: Extra-Mileage Construction Algorithm

Input: TSP instance with cost matrix
Output: Feasible tour s

```
1  Initialize s as empty tour;
2  Find (i, j) such that their distance is maximal;
3  Set s.path[0] ← i, s.path[1] ← j;
4  Mark i and j as inserted, nInserted ← 2;
5  while nInserted < n do
6      bestDelta ← ∞, node_a, node_b, node_h ← −1;
7      foreach pair (a, b) of consecutive nodes in s.path do
8          foreach h ∈ V not yet inserted do
9              Δ ← cost(a, h) + cost(h, b) − cost(a, b);
10             if Δ < bestDelta then
11                 bestDelta ← Δ;
12                 node_a ← a, node_b ← b, node_h ← h;
13             end
14         end
15     end
16     Insert node_h between node_a and node_b in s.path;
17     Mark node_h as inserted;
18     nInserted ← nInserted + 1;
19 end
20 Set s.path[n] ← s.path[0] to complete cycle;
21 Compute cost of s;
22 return s;
```

5.4 Heuristic Algorithms Comparison

We compared the algorithms using the performance profiler, on 10 instances of 1000 nodes each, with 60 seconds as time limit. We have first run both the algorithm without the two opt refinement, and then both with the refinement method. The best algorithm in this case is clearly the Nearest Neighbor with the 2-OPT, as we can see in figure 5.1.

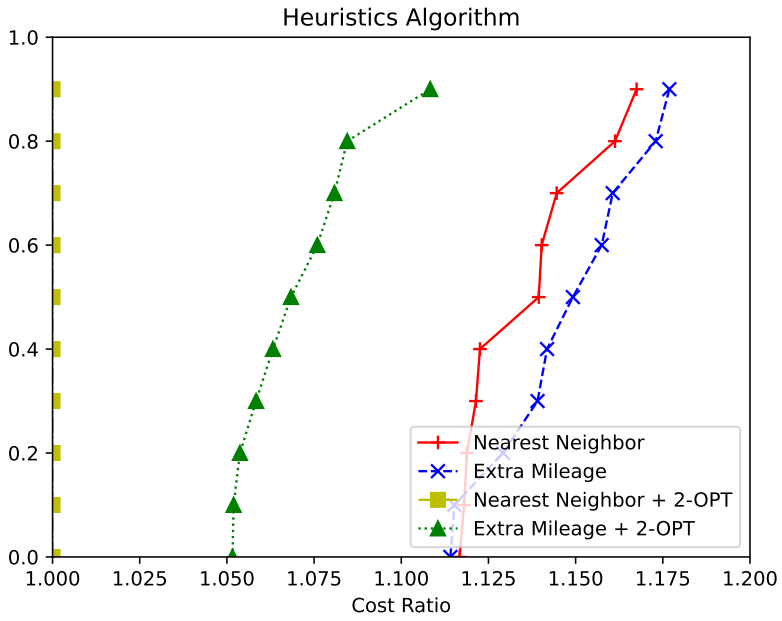


Figure 5.1: Heuristic algorithms comparison

Chapter 6

Metaheuristic

Metaheuristics are high-level search frameworks that extend simple local-search by incorporating adaptive mechanisms to balance intensification (deepening the search around good solutions) and diversification (exploring new regions of the solution space). Although they don't guarantee optimality, by combining randomized decisions with memory structures (such as tabu lists) metaheuristics algorithms can escape local optima and adapt their behavior based on feedback from the ongoing search.

In this chapter we focus on four classic metaheuristic schemes for the TSP: **GRASP (Greedy Randomized Adaptive Search Procedure)**, **Variable Neighborhood Search (VNS)**, **Tabu Search** and **Genetic Algorithm**.

6.1 GRASP

GRASP (Greedy Randomized Adaptive Search Procedure)[9] is an effective multi-start metaheuristic algorithm designed to solve combinatorial optimization problems, comprised of a construction phase that utilizes a greedy randomized method to generate feasible solutions, followed by a local search phase to refine these solutions to local optima. This process is repeated across multiple iterations, allowing GRASP to balance intensification through greedy selection and diversification via randomness, ultimately retaining the best solution discovered throughout the iterations to enhance the exploration of the solution space.

Our implementation of the GRASP algorithm for TSP follows a two-phase approach: a randomized greedy construction phase followed by a local search refinement.

In the construction phase, the algorithm begins from a selected starting node and iteratively builds a complete tour. At each iteration, it identifies the `MIN_COSTS` nearest unvisited nodes based on edge cost. With probability `ALPHA`%, the algorithm selects one of these candidate nodes at random (excluding the absolute best). Otherwise, the nearest unvisited node is selected

deterministically.

In our multi-start GRASP scheme, we systematically launch the GRASP procedure once from each possible starting node, then we apply a 2-OPT refinement. We record the best tour found so far and continue with the next start until we have tried every city or the overall time limit is reached.

Algorithm 5: GRASP Algorithm

Input: TSP instance `inst`, starting node `start_node`

Output: Constructs a feasible TSP tour with a greedy randomized strategy

```
1  visited ← array of nodes zeros;
2  k = inst.params[MIN_COSTS];
3  nearest_node ← array of size k, min_cost ← array of size k;
4  s.path[0] ← start_node, s.path[nodes] ← start_node, visited[start_node] ← 1;
5  for i ← 1 to n − 1 do
6  |  last ← s.path[i − 1];
7  |  Initialize min_cost with INF_COST, nearest_node with −1;
8  |  Find the MIN_COSTS nearest unvisited nodes and put them inside an array
   |   nearest_nodes
9  |  Draw random r ∈ [0, 1];
10 |  if r ≤ α = inst.params[ALPHA]/100.0 then
11 |  |  Select a random node among candidates in nearest_nodes;
12 |  end
13 |  else
14 |  |  Select the best candidate nearest_nodes[0];
15 |  end
16 |  Add selected node to s.path[i] and mark it as visited;
17 end
18 Compute total cost of s and check feasibility;
19 Free memory;
```

6.1.1 Hyperparameters

In our implementation of the GRASP algorithm, we selected specific values for two key hyperparameters, based on empirical testing across various TSP instances. These values were chosen because they consistently led to high-quality solutions within the allocated time constraints.

- **ALPHA = 2:** This parameter controls the level of randomness in the selection of the next node during the construction phase. Specifically, it defines the probability (expressed as a

percentage) of choosing a node that is not the best, but among the next best candidates. An ALPHA value of 2 means that there is a 2% chance at each step to select one of the top candidates (excluding the best), allowing for diversification while still favouring greedy choices.

- **MIN_COSTS = 3:** This parameter determines how many of the nearest candidate nodes (based on cost) are stored at each construction step. A value of 3 means that for every node selection, the algorithm considers the 3 nodes with the lowest costs as potential candidates. This value offers a good compromise between exploration (diversification) and exploitation (intensification) in the search space.

These values were selected through iterative experimentation and thanks to the performance profiler. Figure 7.1 shows the performance of the different GRASP configurations, where each legend entry is formatted as `G_alpha_minCost`, denoting the parameter α and *min_costs*, respectively.

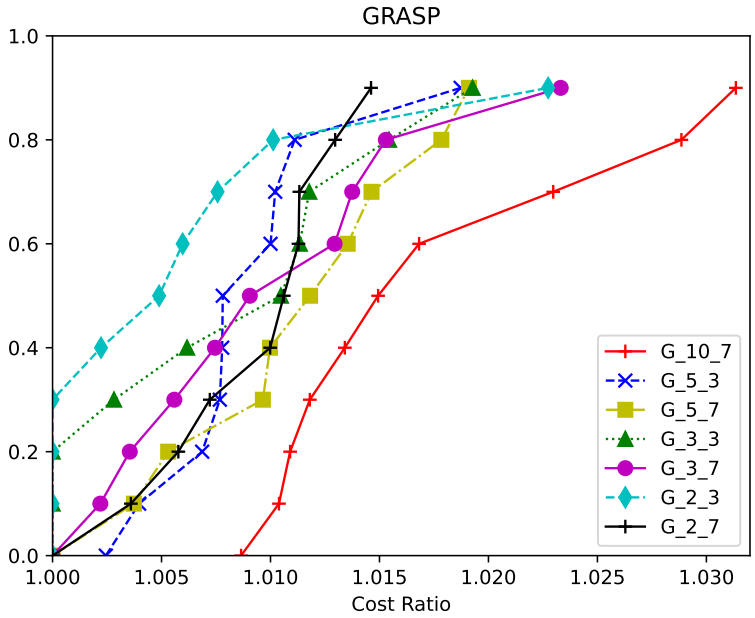


Figure 6.1: GRASP tuning

6.2 Variable Neighborhood Search (VNS)

Variable Neighborhood Search (VNS) is a metaheuristic framework that systematically changes neighborhood structures within the search process to escape local minima and explore the solution space more effectively. In the context of the symmetric Traveling Salesman Problem, VNS iteratively applies perturbations and local search methods to improve an initial solution.

VNS was introduced by Mladenović and Hansen in their seminal 1997 paper, where they showed that systematically changing the neighbourhood structure can greatly enhance diversification while preserving intensification [16].

The VNS algorithm starts with an initial solution found with the Nearest Neighbor algorithm in its best configuration (with 2-opt); then VNS operates by alternating between two primary phases:

1. **Shaking Phase**: A perturbation is applied to the current solution to move to a different region in the solution space. This is achieved by performing a series of k-opt moves, where k is a parameter defining the extent of the perturbation.

2. **Local Search Phase**: A local optimization method, such as 2-opt, is applied to the perturbed solution to find a local minimum in the new neighborhood.

This process is repeated until a stopping criterion is met, such as a time limit or a maximum number of iterations without improvement.

Algorithm 6: Variable Neighborhood Search

Input: TSP instance `inst` with initial solution, time limit `timelimit`

Output: Updates the best solution found

```
1  Initialize s ← copy of inst.best_solution;
2  iterationsWithoutImprovement ← 0;
3  while elapsed time < timelimit and
     iterationsWithoutImprovement < MAX_NO_IMPROVEMENT do
4      oldCost ← cost of inst.best_solution;
5      Copy path from inst.best_solution to s;
6      for i ← 1 to inst.params[KICK] do
7          if inst.params[K_OPT] == 3 then
8              Apply 3-opt to s;
9          else
10             if inst.params[K_OPT] == 5 then
11                 Apply 5-opt to s;
12             else
13                 Apply random k-opt with k = inst.params[K_OPT];
14             end
15         end
16     end
17     Compute cost of s;
18     Apply 2-opt local search to s;
19     if cost of s < cost of inst.best_solution then
20         Update inst.best_solution with s;
21         iterationsWithoutImprovement ← 0;
22     else
23         iterationsWithoutImprovement ← iterationsWithoutImprovement + 1;
24     end
25 end
26 Free memory of s;
```

In our implementation, the "shaking" phase supports three different kick operators—3-OPT, 5-OPT and a general k-OPT move—whose details are described in the following sections. In figure 6.2 we can see the evolution of the solution cost (in blue) and the best solution cost (in red) during the execution of the method. Looking at the blue line, we can see how the "shaking" operation is useful to diversify the search.

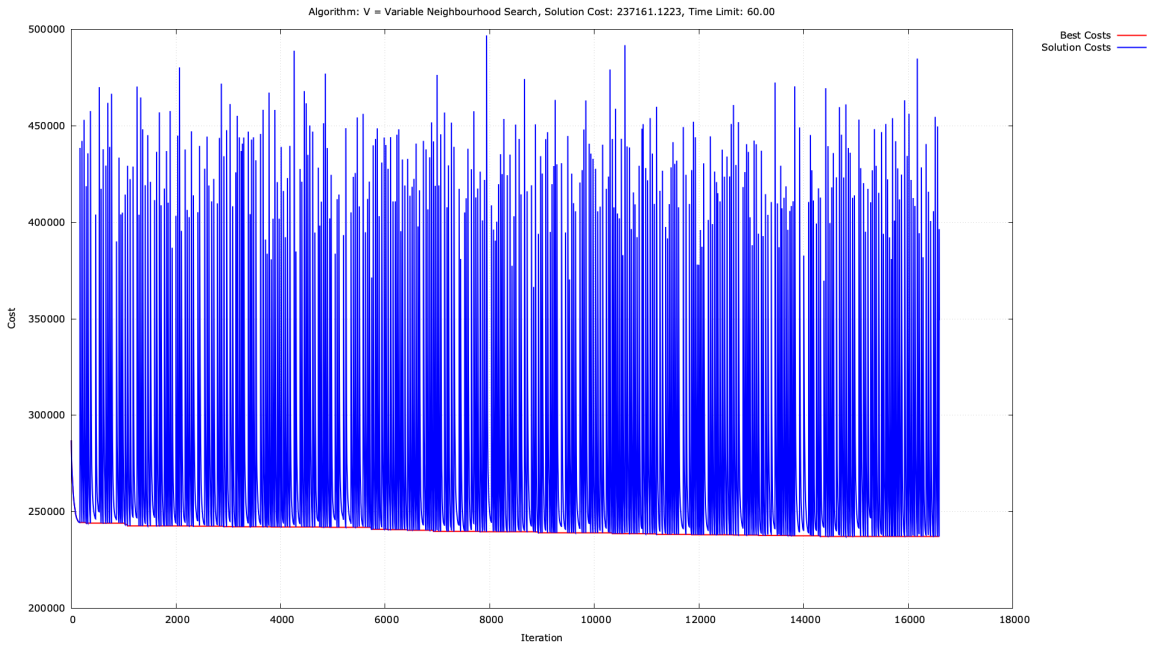


Figure 6.2: Variable Neighborhood history plot

6.2.1 Hyperparameters

In our implementation of the Variable Neighborhood Search (VNS) algorithm, we focused on tuning three key hyperparameters that directly influence the balance between intensification and diversification of the search. These values were determined empirically through iterative testing and performance profiling on a variety of TSP instances.

- **MAX_NO_IMPROVEMENT = 5000:** This parameter defines the maximum number of consecutive iterations without improvement allowed before the algorithm terminates. A value of 5000 allows the algorithm to thoroughly explore the neighborhood space while avoiding premature convergence to suboptimal solutions.
- **KICK = 7:** The KICK parameter controls the number of perturbations (or random changes) applied to the current solution when the algorithm is stuck in a local optimum. A value of 7 implies that seven modifications are introduced sequentially to escape the local optimum and reinitialize the search from a new region of the solution space.
- **K_OPT = 5:** This parameter determines the size of the neighborhood structure used during the perturbation phase. Specifically, a 5-opt move is used to generate significant alterations to the current tour.

The comparative results are summarized in Figure 6.3, where each legend entry is formatted as `V_kick_k-opt`, denoting the parameter *kick* and $k - opt$, respectively.

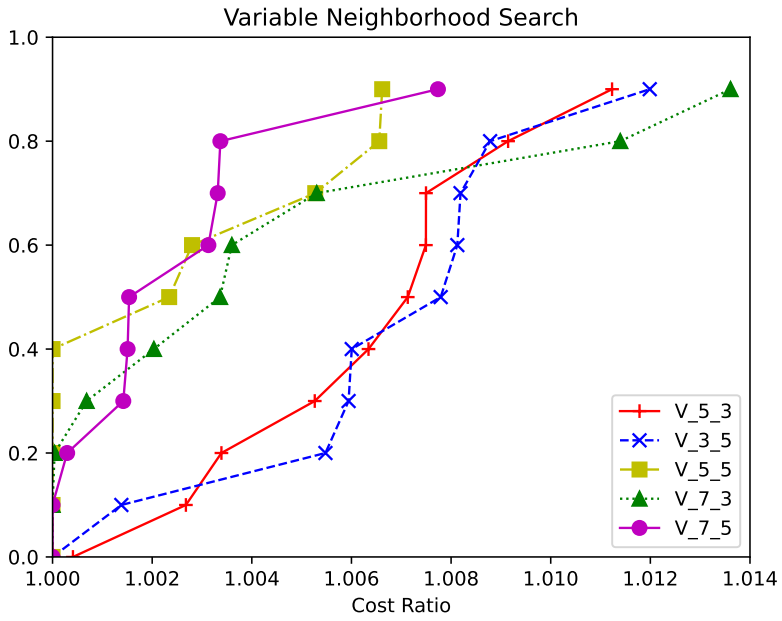


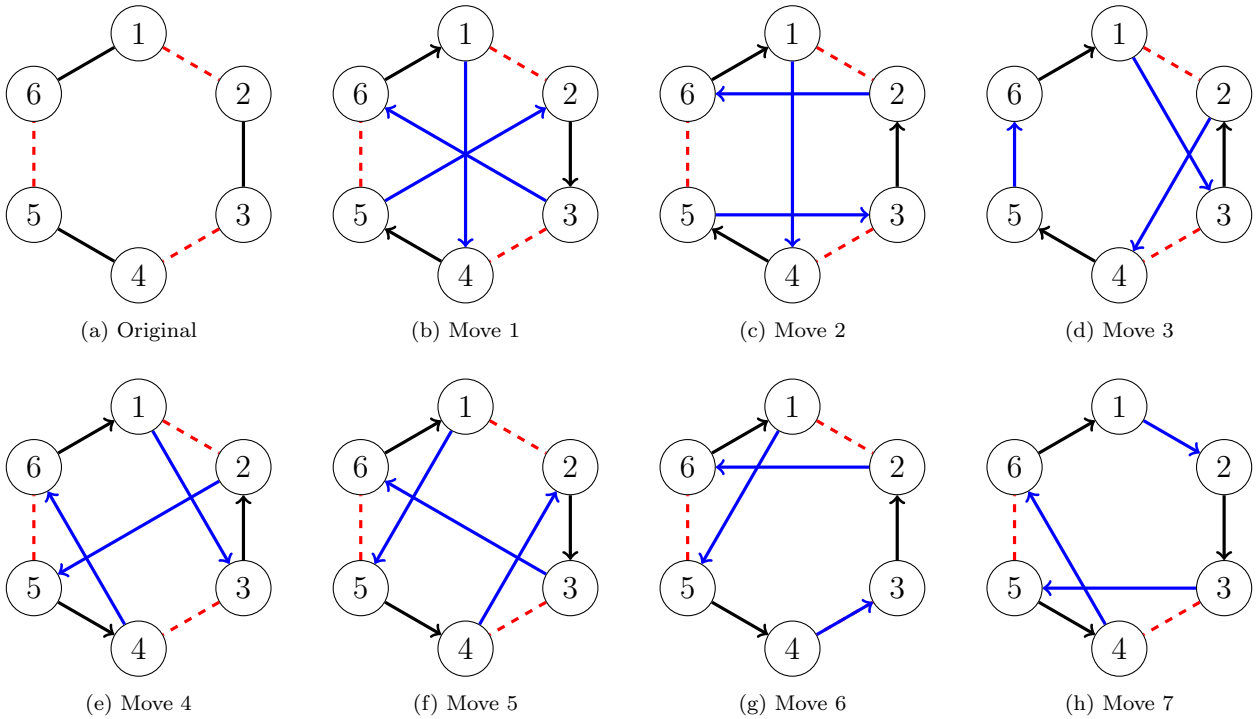
Figure 6.3: Variable Neighborhood tuning

6.3 3-OPT

The *3-OPT* algorithm is a higher-order local search heuristic that generalizes 2-OPT by removing three edges from a current tour and reconnecting the resulting segments in alternative ways [14].

In our implementation, at each iteration, three non-adjacent edges are selected at random with a minimum index separation to partition the tour into three paths. For each of the four canonical reconnection schemes, the change in total tour cost is computed, and the scheme that yields the greatest cost reduction is applied by reversing the appropriate segments. All the possible 3-opt moves are visualized in figure 6.4

By considering triples of edges rather than pairs, 3-OPT explores a larger neighborhood, which not only allows for deeper refinement of a given solution but also provides a mechanism to escape local minima that cannot be resolved by 2-OPT alone. This move-based procedure is applied a fixed number of times, based on the tuned `KICK` parameter.



(a) Original (b) Move 1 (c) Move 2 (d) Move 3

(e) Move 4 (f) Move 5 (g) Move 6 (h) Move 7

Figure 6.4: All Possible 3-OPT moves.
Removed (dashed); Added.

Implementation details and role of the "3". Each 3-OPT move cuts exactly three edges $(\pi_i, \pi_{i+1}), (\pi_j, \pi_{j+1}), (\pi_k, \pi_{k+1})$ with $i < j < k$ and $|i-j|, |i-k|, |j-k| > 1$. Cutting three edges partitions the tour into three chains, which in principle admit seven ways to reconnect. However, three of those reconnections simply replicate what a pair of 2-OPT moves could achieve; the remaining **four true 3-OPT schemes** are the only ones that cannot be reduced to any combination of 2-OPT exchanges.

In our C routine `three_opt` we pick i, j, k uniformly at random under the non-consecutiveness constraint, then call `find_best_move()` to evaluate in constant time the cost change Δ for each of these four irreducible reconnections, and finally `apply_best_move()` reverses the affected segments. Thus the "3" parameter controls the size of the neighborhood: it explores strictly more possibilities than 2-OPT (two cuts) but checks only four alternatives per move instead of the full $O(n^3)$ enumeration.

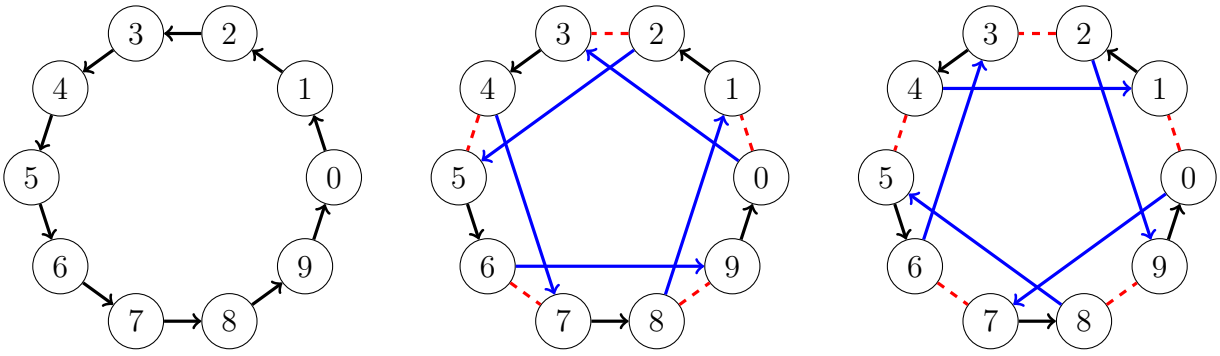
6.4 5-OPT

The *5-OPT* algorithm extends the idea of a 2-OPT local search by simultaneously removing five edges from the current tour and reconnecting the resulting five path segments in a new configuration. We implement the algorithm so that, at each application, five non-adjacent edges

are selected at random, subject to a minimum index separation to define five "cut" positions. The indices are then reordered to ensure a consistent traversal. A reconnection pattern is chosen and the corresponding segments are reversed to form a new tour. In our implementation, we defined just two possible moves that are chosen at random; these are the two moves that change the higher number of edges.

By exploring modifications that involve five edges at once, 5OPT accesses a larger neighborhood than lower-order moves, allowing the search to overcome deeper local optima and potentially discover more substantial improvements. Like the 3-OPT, this move-based procedure is applied a fixed number of times, based on the `KICK` parameters.

Implementation details and role of the "5". Here the value *5* means that five cut points $A < B < C < D < E$ are selected (array `edges[5]`). The auxiliary check `check_valid_kopt_nodes` guarantees that no two of them are adjacent, preserving tour feasibility. Removing five edges creates five paths; in theory there are many possible reconnections, but we adopt only the *two* patterns that flip the largest number of edges (lines marked "pattern 1" and "pattern 2" in `five_opt`). This design choice keeps the cost of each move constant, yet the perturbation is much stronger than with 2- or 3-OPT, allowing the search to escape deep local minima.



(a) Original tour (b) After 5-OPT Pattern 1 (c) After 5-OPT Pattern 2

Figure 6.5: Tour before and after the two maximally-disruptive 5-OPT patterns.
Red dashed edges are removed; blue arrows are newly added.

6.5 K-OPT

The *k-OPT* method generalizes the 2-OPT local search considering the removal of k non-consecutive edges from a given tour and reconnecting the resulting k segments in a different order. In our implementation, k distinct vertices are selected uniformly at random with indices that differ by at least two, ensuring non-adjacency. These indices are then sorted in ascending order to impose a canonical ordering of the affected segments. Subsequently, for each pair among

the selected k positions, the segment between them is reversed, each reversal constituting a 2-OPT operation on that subpath. This randomized sequence of k segment reversals can be repeated iteratively or until a prescribed time limit is reached. By varying k, the heuristic interpolates between the light computational cost of 2-OPT ($k = 2$) and the stronger but more expensive improvements attainable with larger k. The overall strategy is inspired by the k-exchange heuristics introduced in the work of Lin and Kernighan [15].

Implementation details and role of the parameter k. The integer $k \geq 2$ is a *user-tunable knob* that sets the number of breakpoints (and therefore edges) removed at once. Function `random_k_opt(inst,s,k)` first checks that $2 \leq k \leq n$; it then draws k distinct, non-adjacent indices and sorts them. Instead of exploring the $\mathcal{O}(k!)$ possible permutations, the procedure performs exactly k random `reverse_segment` operations between pairs of chosen indices, for a total cost of $O(k)$. Consequently:

- Small k (e.g. 3) yields cheap, intensification-oriented moves.
- Larger k (e.g. 5 or 7, as in our VNS experiments) delivers powerful shakes that foster diversification.

By tuning k, the practitioner finds the right balance between the computational cost and the neighbourhood width. This results in a smooth interpolation between 2-OPT and complete Lin-Kernighan exchanges.

6.6 Tabu Search

The idea behind *Tabu Search* is to explore the solution space by performing local moves and allowing non-improving moves to escape local optima, while maintaining a memory structure, called *tabu list*, to avoid cycling back to recently visited configurations. In the context of TSP, the algorithm iteratively applies edge swaps (e.g., 2-opt style) and forbids reversing recent moves for a number of iterations determined by the tenure.

Tabu Search was pioneered by Glover, who formalised the use of adaptive memory and strategic oscillation to guide local search beyond local optima [11].

The Tabu Search algorithm starts from an initial feasible solution and explores its neighborhood through local modifications, specifically using 2-opt edge swaps. At each iteration, the algorithm evaluates all admissible moves and selects the best one in terms of cost improvement (*delta*), even if it results in a temporary worsening of the solution, as long as it is not prohibited by the memory structure.

To prevent revisiting recently explored configurations, the algorithm employs a **tabu list**, implemented as a symmetric matrix. Each entry `tabuList`$[i][j]$ tracks the iteration until which the move involving nodes i and j remains forbidden. Rather than maintaining an explicit FIFO queue, the algorithm simulates FIFO behavior using an iteration counter: a move is considered tabu if the current iteration is less than the value stored in the matrix.

A distinguishing feature of our implementation is the **adaptive management of the tabu tenure**. Instead of a fixed duration, the tenure dynamically varies throughout the execution of the algorithm. This variation follows a cyclical pattern that alternates phases of intensification and diversification, helping the search escape local optima and explore new regions of the solution space.

Figure 6.6 illustrates how the current solution cost (blue curve) and the all-time best cost (red curve) evolve over time. Notice that the blue line, after a sharp initial decline, begins to oscillate as the algorithm continues to explore its neighborhood, occasionally accepting higher-cost moves in search of better long-term improvements, while the red line only decreases when a new better tour is found.

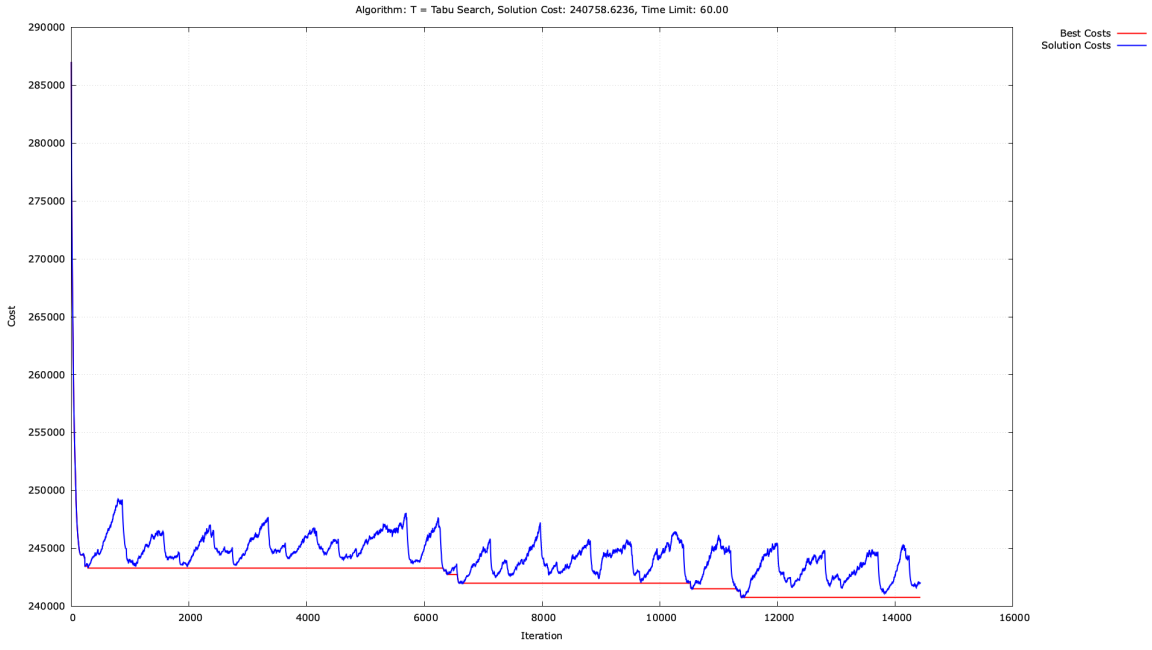


Figure 6.6: Tabu history plot

Algorithm 7: Tabu Search

Input: TSP instance `inst` with initial solution, time limit `timelimit`
Output: Updates the best solution found

Initialize solution $s \leftarrow$ copy of `inst.best_solution`;
Initialize tabu list `tabuList`$[i][j] \leftarrow 0$ for all node pairs (i, j);
$currentTenure \leftarrow$ `inst.params[MIN_TENURE]`;
$iter \leftarrow 0$;
while *elapsed time* $<$ *timelimit* **do**
  $minDelta \leftarrow \infty$;
  $swap_i \leftarrow -1$, $swap_j \leftarrow -1$;
  for $i \leftarrow 0$ **to** $nodes - 1$ **do**
    for $j \leftarrow i + 2$ **to** $nodes - 1$ **do**
      if $iter <$ *tabuList*$[s.path[i]][s.path[j]]$ **then**
        continue;
      end
      $delta \leftarrow$ cost difference from swapping segment (i, j);
      if $delta < minDelta$ **then**
        $minDelta \leftarrow delta$;
        $(swap_i, swap_j) \leftarrow (i, j)$;
      end
    end
  end
  if $swap_i == -1$ ***or*** $swap_j == -1$ **then**
    Exit: No admissible move found;
  end
  Update tabu list:
    `tabuList`$[s.path[swap_i]][s.path[swap_j]] \leftarrow iter + currentTenure$;
    `tabuList`$[s.path[swap_j]][s.path[swap_i]] \leftarrow iter + currentTenure$;
  Reverse segment $(swap_i + 1, swap_j)$ in s;
  Compute cost of s, update best if improved;
  $iter \leftarrow iter + 1$;
  $currentTenure \leftarrow currentTenure +$ `inst.params[TENURE_STEP]`;
  if $currentTenure >$ *inst.params[MAX_TENURE]* **then**
    $currentTenure \leftarrow$ `inst.params[MIN_TENURE]`;
  end
end
Free memory used by s and `tabuList`;

6.6.1 Hyperparameters

The performance of the Tabu Search algorithm is strongly influenced by the way the *tabu tenure* is managed. In our implementation, the tenure is not constant, but dynamically adjusted through three key hyperparameters:

- **MIN_TENURE = 700:** This parameter sets the initial value for the tenure, determining how many iterations a move remains tabu at the beginning of the search. A lower value promotes intensification by allowing the algorithm to revisit configurations more frequently.
- **MAX_TENURE = 900:** This is the upper bound for the tenure. Once the tenure reaches this value, it is reset to `MIN_TENURE`. A high value increases diversification, helping the algorithm escape local optima by preventing recently visited moves from being revisited for longer periods.
- **TENURE_STEP = 50:** At every iteration, the tenure is incremented by this value. This causes the tenure to grow gradually over time, cycling between `MIN_TENURE` and `MAX_TENURE`. This cyclical behavior enables a controlled alternation between exploration and exploitation phases.

The values for these parameters were selected through empirical tuning. The comparative results are summarized in Figure 7.1, where each legend entry is formatted as `G_minTenure_maxTenure_tenureStep`, denoting the parameter *min_tenure*, *max_tenure* and *tenure_step*, respectively. Several configurations were evaluated, and the triplet `MIN_TENURE = 700`, `MAX_TENURE = 900`, and `TENURE_STEP = 50` provided the most consistent results across different TSP instances.

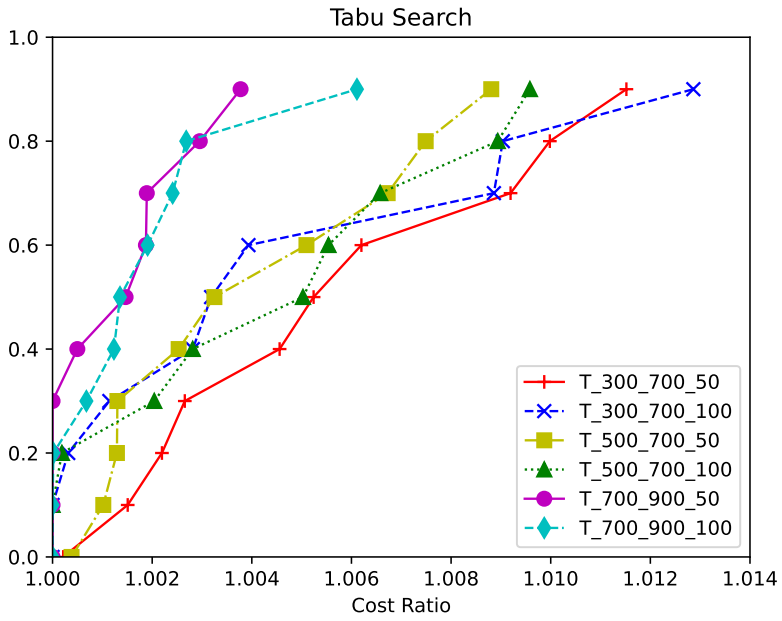


Figure 6.7: Tabu Search tuning

6.7 Genetic Algorithm

The *Genetic Algorithm (GA)* is a population-based metaheuristic inspired by evolutionary biology. It maintains a population of feasible solutions that evolve over time through genetic operations such as crossover, repair, and selection. Our design choices are consistent with the study of GA variants for the TSP by Potvin, which highlights effective crossover and repair strategies for permutation problems [17].

In our implementation, the algorithm begins by generating an initial population of random tours. At each generation, pairs of parent solutions are selected to produce new offspring via a one-point crossover operator. Since crossover can generate infeasible TSP tours (with duplicate or missing nodes), a two-step repair process is applied. First, a *shortcut* operation removes duplicates by retaining only the first occurrence of each node. Then, the *extra mileage* heuristic inserts the missing nodes by identifying the best insertion point that minimizes the cost increase.

After repair, solutions that are significantly worse than the current best (specifically, 80% worse) are further refined using a 2-opt local search. Once all children are generated, a subset of the population is removed using a stochastic selection mechanism, where individuals are eliminated based on their fitness probabilities inversely proportional to their cost. This ensures that higher-quality solutions have a higher chance of survival, while maintaining diversity. The best individual in the population is preserved throughout the process to avoid regression. This

evolutionary process continues until the time limit is reached, at which point the best solution encountered is returned.

However, with a strict time limit and a large population (e.g. `POPULATION_SIZE = 500`), the 2-OPT refinement often cannot be applied to every child before the time expires, leading to suboptimal convergence. In contrast, when the time limit is set to one hour and 2-OPT is applied to all offspring without per-application time restriction, the genetic algorithm achieves solution qualities of the same order of magnitude with the other metaheuristics, as we can see in figure 6.8. An alternative strategy is to reduce the population size (e.g. to 10–20) so that 2-OPT can be run on every new individual; in that case the final costs remain of the same order of magnitude as the best algorithms described earlier.

Algorithm 8: Genetic Algorithm for TSP

Input: TSP instance `inst`, time limit `timelimit`

Output: Best solution found

```
Initialize population with random feasible tours;
while elapsed time < timelimit do
    Identify the best individual (champion) in the population;
    Store the best cost found so far;
    for each child to generate do
        Select two parents at random;
        Apply one-point crossover to generate a child tour;
        Remove duplicates using shortcut operation;
        Complete the tour using extra-mileage heuristic;
        if child is 80% worse than best then
            Apply 2-opt local search to refine it;
        end
    end
    Compute total population cost;
    Remove individuals from the population using cost-proportional probabilities
     (excluding best);
    Replace eliminated individuals with generated children;
end
Return best solution found in the population;
```

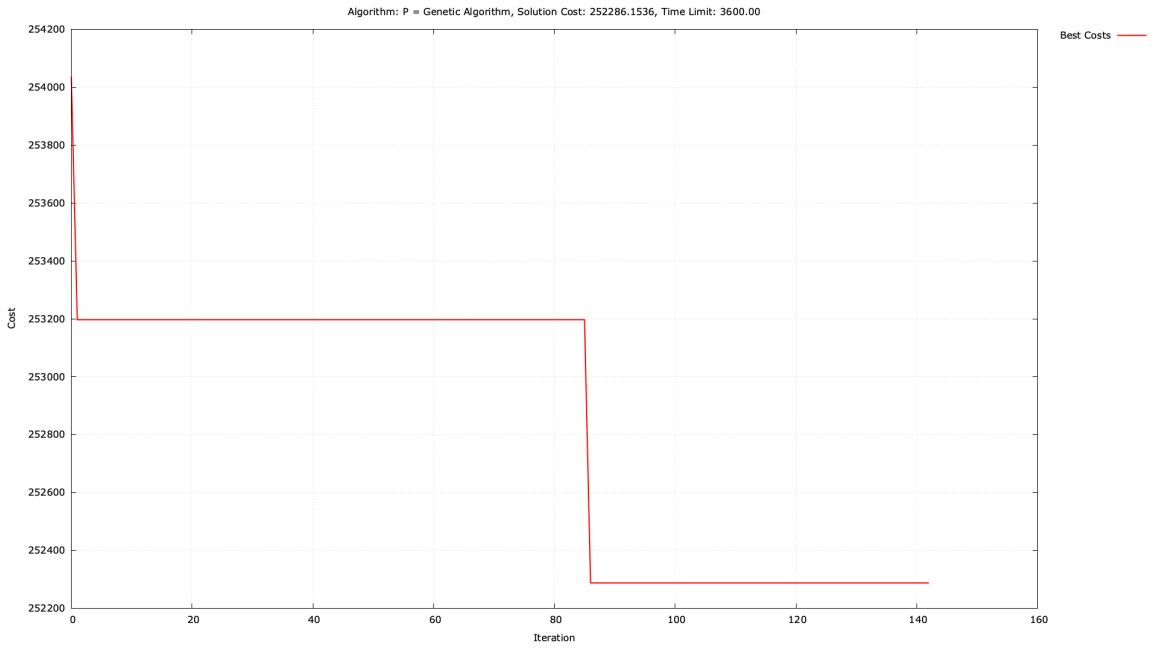


Figure 6.8: Genetic Algorithm history plot

6.7.1 Hyperparameters

The genetic algorithm relies on two hyperparameters to control the size and progression of the population over generations:

- **POPULATION_SIZE = 500:** This parameter defines the total number of solutions maintained in the population at any time. A larger population enhances genetic diversity but increases memory and computational demands.
- **GENERATION_SIZE = 20:** Specifies the number of child solutions generated in each generation. These are inserted into the population via replacement, ensuring evolution and exploration of the solution space.

The comparative results are summarized in Figure 6.9, where each legend entry is formatted as `GA_populationSize_generationSize`, denoting the parameter *population_size* and *generation_size*, respectively.

Instead, wit a low population size and applying 2-OPT at each child (removing the if of line 10 in the pseudocode) , the best tuning is with **POPULATION_SIZE = 10** and **GENERATION_SIZE = 5**, as we can see from figure 6.10.

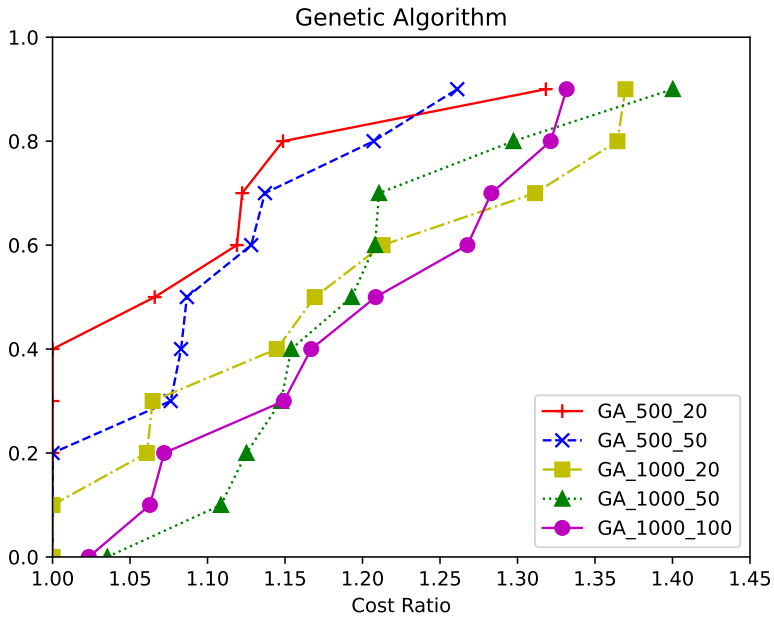


Figure 6.9: Genetic Algorithm tuning

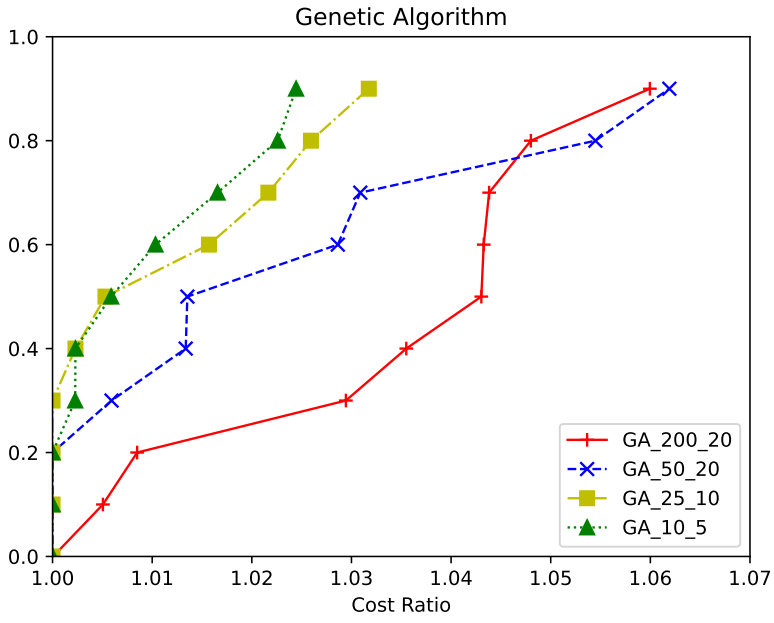


Figure 6.10: Genetic Algorithm tuning with low population sizes

6.8 Metaheuristic Algorithms Comparison

We compared the algorithms using the performance profiler, on 10 instances of 1000 nodes each, with 60 seconds as time limit. All metaheuristics were executed in their best-tuned configurations. As shown in figure 6.11, Variable Neighborhood Search clearly outperforms the others, with Tabu Search achieving similarly strong results. Our Genetic Algorithm performs worse by comparison, although this outcome is specific to our implementation, the chosen time limit, and the problem size used here. In this plot, we used the Genetic Algorithm with low population size, so in its best configuration.

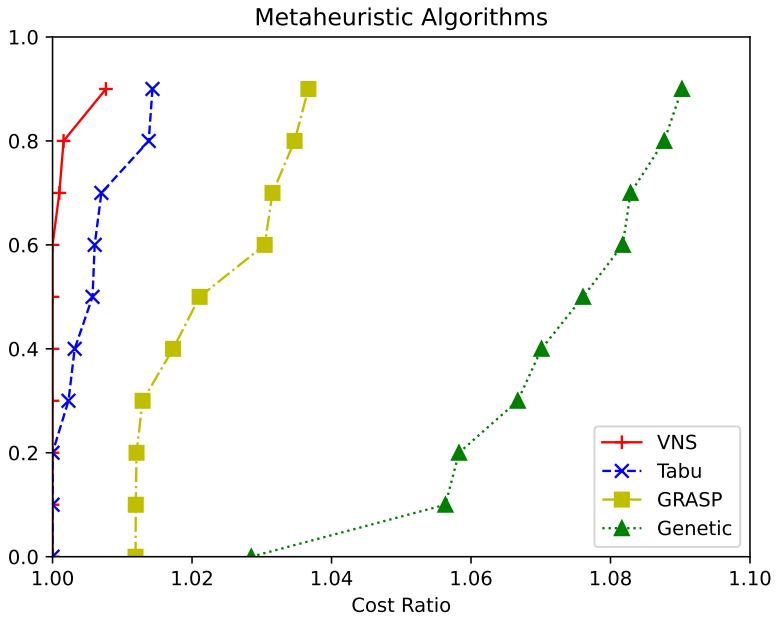


Figure 6.11: Metaheuristic algorithms comparison

Chapter 7

Exact Algorithms

7.1 Introduction

Unlike heuristic or metaheuristic methods, which provide good approximations in limited time, exact algorithms are designed to solve combinatorial problems to proven optimality. In this chapter, we focus on exact approaches for the Traveling Salesman Problem, which, despite being NP-hard, can be solved to optimality for moderate-sized instances using advanced integer programming techniques and efficient solvers like IBM ILOG CPLEX.

IBM ILOG CPLEX Optimization Studio [13] is a commercial optimization software package that includes a high-performance linear programming solver. CPLEX has been utilized as a key component in exact algorithms. Concorde, recognized as the most performing exact algorithm currently available for the TSP, relies on CPLEX as its linear programming solver. The exact algorithms proposed in this book will use the CPLEX package; a detailed guide on how to install the library is written on Appendix A. This package is available for free with an academic account on the official IBM site.

We implemented two exact algorithms to solve TSP instances: a classic Branch-and-Cut method and a Benders-like decomposition. Both rely on the compact edge-based Integer Linear Programming (ILP) formulation of the symmetric TSP, where binary variables x_e indicate whether edge $e \in E$ is selected in the tour. Subtour elimination constraints (SECs) are handled dynamically through callback mechanisms to improve scalability.

These approaches are supported by custom preprocessing, warm-start heuristics, and integration with the Concorde TSP library, which enables powerful cut separation.

7.2 CPLEX Environment and Model Initialization

All our implementations share the following steps to set up and solve the TSP with CPLEX:

1. **Create CPLEX environment and problem:**

```
int error = 0;
CPXENVptr env = CPXopenCPLEX(&error);
if (error) print_error("CPXopenCPLEX() error");
CPXLPptr lp = CPXcreateprob(env, &error, "TSP model version 1");
if (error) print_error("CPXcreateprob() error");
```

   This allocates the CPLEX environment `env` and the problem object `lp`.

2. **Build the model:**

```
build_model(inst, env, lp);
```

   The `build_model` function (see Appendix B) adds binary edge variables x_{ij} and degree constraints $\sum_{j \neq i} x_{ij} = 2$ for all i.

3. **Set solver parameters:**

```
// turn off screen output by default
CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);
// require a very tight relative MIP gap
CPXsetdblparam(env, CPX_PARAM_EPGAP, 1e-9);
if(VERBOSE >= DEBUG) CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_ON
   );
// later, inside the main loop, enforce the remaining time limit
CPXsetdblparam(env, CPX_PARAM_TILIM, residual_time);
```

   We silence console output unless in verbose mode, set an optimality tolerance of 10^{-9}, and dynamically adjust the time limit.

4. **Solve and retrieve solution:**

```
error = CPXmipopt(env, lp);
if (error) print_error("CPXmipopt() error");
double objval;
CPXgetobjval(env, lp, &objval);
CPXgetx(env, lp, xstar, 0, inst->ncols-1);
build_sol(xstar, inst, succ, comp, &ncomp);
```

   You can find the description of the `build_sol` function in appendix B.

5. **Write model (debug mode) and cleanup:**

```
if (VERBOSE >= DEBUG)
    CPXwriteprob(env, lp, "model.lp", NULL);
CPXfreeprob(env, &lp);
CPXcloseCPLEX(&env);
```

This sequence—environment setup, model construction, parameter tuning, (optional) callbacks, optimization, and cleanup—is common to every implementation of our exact algorithms variants.

In the following sections, we will call the function to transform a solution into CPLEX format `xpos`, which is described in appendix B.

7.3 Benders' Decomposition

The *Benders' decomposition* approach [3] exploits the structure of the TSP by separating the problem into:

- **Master problem:** Contains the degree constraints

$$\sum_{e \in \delta(v)} x_e = 2 \quad \forall i \in V, \quad x_e \in \{0, 1\},$$

  and a current set of Benders cuts (subtour elimination constraints).

- **Subproblem (Cut generation):** Given a candidate solution x^* from the master, check for subtours. If a subtour $S \subset V$ is found, generate the cut

$$\sum_{e \in E(S)} x_e \ \leq \ |S| - 1,$$

  and add it to the master.

We iterate between solving the master and generating cuts until no violated subtour is found or the time limit is reached.

The Benders-like decomposition used in our implementation is inspired by classical logic-based Benders decomposition, although we do not fully decouple the problem into master and subproblem layers with duality reasoning. Instead, we implement a simplified cut-generation loop: at each iteration, the model is solved with the current set of constraints, and subtour elimination constraints are dynamically added when needed.

This variant is easier to integrate into the CPLEX environment using standard MIP facilities while retaining the core benefit of delayed constraint generation, thereby improving tractability on larger graphs.

Algorithm 9: Benders Decomposition for the TSP

Input: TSP instance inst, time limit t

Output: Optimal tour (if found) or best feasible tour

Initialize: master problem $\mathcal{M}$ with degree constraints only;
Optional: add warm-start solution to CPLEX // Described in the next section;
while $ncomp \geq 2$ **do**
    solve $\mathcal{M}$ as MIP to obtain solution x^*;
    build_sol(x^*, inst, succ, comp, ncomp);
    if $ncomp \geq 2$ **then**
        foreach *subtour* $S \in \mathcal{S}$ **do**
            add constraint $\sum_{e \in E(S)} x_e \leq |S| - 1$ to $\mathcal{M}$ `// Through add_sec();`
        end
    end
    check residual time
end
if $ncomp \geq 2$ **then**
    patching_heuristic(inst, succ, comp, ncomp) //Described in the next sections;
end
Transform solution from CPLEX to our representation;
Update best solution;

7.4 Branch-and-Cut

Branch-and-Cut is one of the most effective general-purpose methods for solving the TSP to optimality. It combines branch-and-bound with a dynamic cut-generation scheme, where violated constraints (in our case, subtour elimination constraints) are added on-the-fly as they are discovered, to accelerate convergence.

In our implementation, CPLEX handles branching and node selection, while we inject violated SECs during the candidate callback. Optionally, we extend this with Concorde-based separation at the root node through the relaxation callback, which identifies deep fractional violations using minimum cut routines.

This combination of lazy constraints and user cuts enables strong relaxations while avoiding

the exponential blowup of modeling all SECs explicitly.

The main steps are: The classic Branch-and-Cut approach dynamically adds Subtour Elimination Constraints (SECs) during the optimization process using CPLEX's callback mechanism. Depending on the configuration, this mechanism supports both:

- lazy cuts via the `candidate callback`, and
- user cuts (e.g., via Concorde) via the `relaxation callback`.

Callback registration: At the beginning of the solution process, we register a unified callback function using `CPXcallbacksetfunc`. The type of callback depends on the configuration:

- If parameter `inst->params[CONCORDE]` is enabled, the callback is registered for both `CPX_CALLBACKCONTEXT_CANDIDATE` and `CPX_CALLBACKCONTEXT_RELAXATION`.
- Otherwise, it is only used in the `CANDIDATE` context.

```
if (inst->algorithm == 'C') {
    CPXLONG contextid;
    if(inst->params[CONCORDE]){
        contextid = CPX_CALLBACKCONTEXT_RELAXATION |
           CPX_CALLBACKCONTEXT_CANDIDATE;
    } else{
        contextid = CPX_CALLBACKCONTEXT_CANDIDATE;
    }
    if(CPXcallbacksetfunc(env, lp, contextid, cpx_callback, inst)){
        print_error("CPXcallbacksetfunc() error");
    }
}
```

The unified function `cpx_callback` dispatches the logic based on the context:

```
int CPXPUBLIC cpx_callback(CPXCALLBACKCONTEXTptr context, CPXLONG
   contextid, void *userhandle){
instance* inst = (instance*) userhandle;
if(contextid == CPX_CALLBACKCONTEXT_CANDIDATE)
return sec_callback(context, contextid, inst);
if(contextid == CPX_CALLBACKCONTEXT_RELAXATION)
return concorde_callback(context, contextid, inst);
print_error("Unknown contextid in cpx_callback()");
return 1;
}
```

This function acts as a context-sensitive dispatcher. If the callback is triggered during an integer-feasible solution, it invokes `sec_callback`, which lazily adds SECs. If instead the callback is triggered at an LP-relaxation point (e.g., root node), it delegates to `concorde_callback` to separate fractional cuts using Concorde's routines.

7.4.1 Classic Branch-and-Cut

The core of the Branch-and-Cut algorithm is structured as follows:

1. **Model Construction and Parameter Configuration:** As described in section 7.2.
2. **Warm-Start Solution:** Optionally provide an initial feasible tour using the heuristic described in section 7.5.
3. **Optimization:** Call `CPXmipopt`. CPLEX will automatically invoke the callback to check for violated constraints.
4. **Post-Processing:** Extract the best solution found using `CPXgetx`, rebuild the tour using `build_sol`, and apply the `patching_heuristic` if any subtours remain, described in section 7.6.
5. **SEC Injection:** If multiple components are detected, inject violated SECs with `add_sec()` and apply the patching heuristic to merge subtours.

Algorithm 10: Classic Branch-and-Cut for the TSP

Input: TSP instance with vertex set V, time limit t

Output: Optimal tour (if found) or best feasible tour

```
Initialize CPLEX model;
Register sec_callback for candidate context via CPXcallbacksetfunc;
CPXmipopt(env, lp);
CPXgetx(env, lp, xstar, 0, inst → ncols-1);
build_sol(xstar, inst, succ, comp, &ncomp);
if ncomp ≥ 2 then
    call add_sec();
    patching_heuristic(inst, succ, comp, ncomp)
end
Transform CPLEX solution into our tour format and update best solution;
```

7.5 Warm-Start Heuristic

Providing an initial feasible tour allows CPLEX to begin the Benders' method or the Branch-and-Cut process with a valid incumbent, which can drastically reduce the size of the search tree. This technique is especially beneficial for MIP-based TSP formulations, where a poor initial bound often leads to unnecessary branching.

Our method leverages the nearest-neighbor heuristic for fast construction and 2-OPT local optimization for refinement. The final tour is transformed into CPLEX's variable space using the function `solution_to_CPX` and passed via `CPXaddmipstarts`.

Algorithm 11: Warm-Start Heuristic for the TSP

Input: TSP instance *inst, CPXENVptr env, CPXLPptr lp

Output: Feasible tour π^* used as initial incumbent

1 Nearest-Neighbor Construction;
2 2-OPT Refinement;
3 Transform solution into CPLEX format through xpos;
4 Call to CPXaddmipstarts;

By providing CPLEX with a high-quality incumbent, the solver can prune large portions of the tree early, often yielding significant reductions in total runtime.

7.6 Patching Heuristic

Since solving an instance of 1000 nodes with exact methods (Benders and Branch-and-Cut) can be difficult, we apply a heuristic method to return a feasible solution also when CPLEX solver does not find an optimal solution. When the solution returned by CPLEX contains more than one subtour (i.e. $ncomp > 1$), we apply a greedy patching heuristic. This heuristic merges the "main" subtour (`comp` $= 1$) with another subtour k by selecting the pair of edges—one in each cycle—whose swap yields the smallest increase in tour length, optionally invoking `reverse_cycle` on subtour k to maintain orientation, rewiring the four endpoints accordingly, and then relabeling all nodes of k to 1.

Algorithm 12: Patching Heuristic to Merge Subtours from CPLEX

Input: TSP instance *inst*, successor array `succ`, component array `comp`, number of components n_{comp}

Output: Updated successor array `succ` forming a single tour if possible

if $ncomp < 2$ **then**
  return
end
for $c1 \leftarrow 1$ **to** $ncomp$ **do**
  for $c2 \leftarrow c1 + 1$ **to** $ncomp$ **do**
    if *no nodes with comp*$==c2$ **then**
      Continue;
    end
    $C1 \leftarrow \{\, i \mid comp[i] = c1\}$, $C2 \leftarrow \{\, j \mid comp[j] = c2\}$;
    $\Delta_{\min} \leftarrow +\infty$, *best_swap* $\leftarrow$ none;
    foreach $(i \in C1,\ j \in C2)$ **do**
      $i' \leftarrow succ[i]$, $j' \leftarrow succ[j]$;
      $\Delta_1, \Delta_2 \leftarrow$ compute two deltas(i, j, i', j');
      update ($\Delta_{\min}$, *best_swap*) if smaller
    end
    //check if the orientation of the second tour is different from the orientation of the first tour;
    if *orientation* $=$ *true* **then**
      reverse_cycle$\big($*best_swap*.$j)$;
    make the best swap;
    relabel all nodes in $C2 \rightarrow c1$;
  end
end

7.7 SEC Injection Function `add_sec()`

The `add_sec()` function is responsible for injecting subtour elimination constraints in all phases of our exact algorithms, including Benders, Branch-and-Cut, and Concorde integration.

Given a component array `comp[]` labeling each node by its connected component and the number of components n_{comp}, the function scans each component $S \subset V$ and constructs the constraint:

$$\sum_{e \in E(S)} x_e \leq |S| - 1$$

This constraint is valid for the TSP and prevents the reoccurrence of disconnected subtours.

Depending on context, the function behaves as follows:

- In the `CANDIDATE` context: SECs are added as lazy constraints via `CPXcallbackrejectcandidate()`.
- In the `RELAXATION` context: SECs are added as user cuts via `CPXcallbackaddusercuts()`.
- Outside of callbacks: SECs are added directly to the model via `CPXaddrows()`.

This flexibility allows `add_sec()` to be reused consistently across all components of our exact framework.

7.8 Concorde Integration

The Concorde TSP solver provides sophisticated routines to detect violated subtour elimination constraints (SECs) in fractional LP solutions via minimum cut separation.

In our implementation, we partially integrate Concorde's logic within the Branch-and-Cut framework, using it as a user-cut generator inside the relaxation callback of CPLEX. Specifically, we use Concorde's `CCcut_violated_cuts` function to separate fractional subtours from LP solutions. The procedure works as follows:

1. During the CPLEX relaxation callback, we extract the current fractional solution x^*
2. We call Concorde's `CCcut_connect_components` function to check whether the graph defined by x^* is connected. If the graph is not connected, we add SECs via the same mechanism used in the candidate callback.
3. If the graph is connected, we invoke Concorde's `CCcut_violated_cuts` routine, passing a callback function `violated_cuts_callback`.
4. All detected SECs are then added to the model via CPLEX's `CPXcallbackaddusercuts` function, with the `CPX_USECUT_FILTER` flag.

To avoid overhead, the Concorde-based cut generation is only performed at the root node.

Algorithm 13: Concorde-Based SEC Separation

Input: Fractional solution x^*, instance `inst`, context `context`

Output: Add violated SECs to LP relaxation

Get current node index from `context`;
if *node index* $\neq 0$ **then**
  return; `// Run only at root node`
end
Build edge list `elist` for complete graph $G = (V, E)$;
Call `CCcut_connect_components` on (V, E, x^*);
if *graph is disconnected* **then**
  build_sol(x^*, inst, succ, comp, ncomp);
  call `add_sec()` with subtour components;
else
  Define violation threshold $\theta = 1.9$;
  Call `CCcut_violated_cuts` with callback `violated_cuts_callback`;
end

Algorithm 14: Violated SEC Callback (Concorde)

Input: Subset of nodes $S \subset V$ violating SEC, size $|S| = n$, violation value `cutval`, instance `inst`

Output: Post user cut $\sum_{e \in E(S)} x_e \leq |S| - 1$

Set right-hand side: `rhs` $\leftarrow n - 1$;
Set cut sense: `sense` $\leftarrow$ 'L';
Set number of non-zero: `nnz` $\leftarrow 0$;
Set flag: `purgeable` $\leftarrow$ CPX_USECUT_FILTER;
Initialize `index[]` and `value[]` arrays;
foreach *pair* $(u, v) \in S \times S,\ u < v$ **do**
  index[k] $\leftarrow$ xpos(u, v);
  value[k] $\leftarrow 1.0$;
  nnz++;
end
Post cut via `CPXcallbackaddusercuts(context, 1, nnz, &rhs, &sense, &matbeg, index, value, &purgeable, &local)`;

7.9 Posting Solution

One of the advanced features provided by CPLEX is the ability to post heuristic solutions during the Branch-and-Cut process via the callback mechanism. This functionality is especially useful when good-quality solutions are discovered at intermediate nodes of the search tree but are not automatically accepted by the solver due to issues such as subtours or missing subtour elimination constraints (SECs) in the current model.

Our implementation leverages this mechanism by applying a solution posting strategy in the candidate callback. When a candidate integer solution is found and it contains multiple connected components (i.e., subtours), we attempt to reconstruct a feasible tour by applying a patching heuristic followed by a 2-OPT local refinement. If the resulting tour improves upon the current incumbent, it is then submitted to CPLEX using the `CPXcallbackpostheursoln` function. To avoid overhead and ensure high-quality contributions, we only post solutions at shallow nodes of the branch-and-bound tree (i.e., below a certain depth threshold).

Algorithm 15: Solution Posting in Branch-and-Cut

Input: TSP instance `inst`, callback context `context`

Output: Post refined feasible tour (if better than incumbent)

During candidate callback: extract current integer solution x^*;
build_sol(x^*, inst, succ, comp, ncomp);
if *ncomp* ≥ 2 **then** `// If multiple components`
    add violated SECs via lazy constraints;
else
end `// Node is not too deep`
depth $\leq d$ patching_heuristic(inst, succ, comp, ncomp);
extract tour s from succ array;
two_opt refinement on s;
if *cost(s)* $<$ *incumbent* **then**
    post s as heuristic solution to CPLEX via CPXcallbackpostheursoln;
end

7.10 Benders Hyperparameters

The Benders-like decomposition requires minimal hyperparameter tuning. The only relevant setting in our implementation is:

- **WARMUP = 1:** Enables the use of an initial heuristic solution (via nearest-neighbor

followed by 2-OPT and optionally also VNS) to warm-start the master problem. This significantly reduces the number of iterations required to eliminate all subtours, especially on larger instances.

Warm-starting the master problem was empirically shown to speed up convergence and provide better initial upper bounds, resulting in fewer total subtour iterations, as we can see in figure 7.1

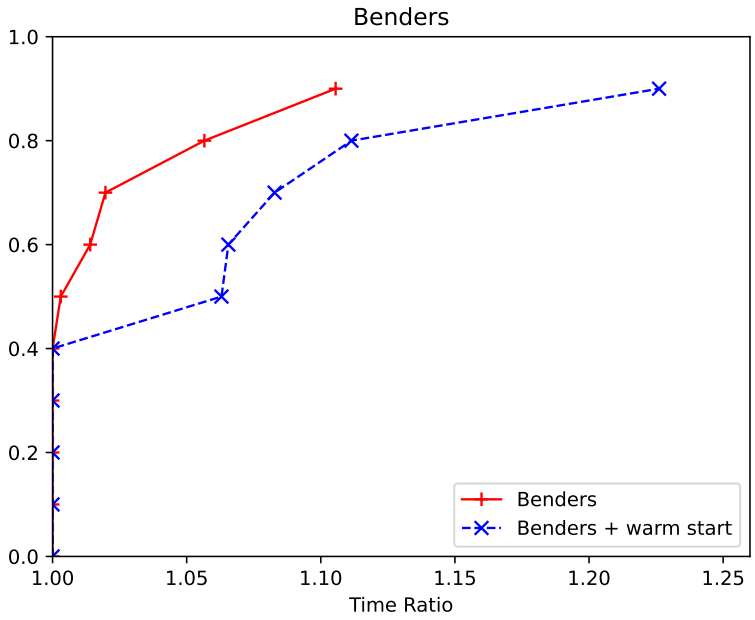


Figure 7.1: Benders tuning

7.11 Branch-and-Cut Hyperparameters

The classic Branch-and-Cut method includes several tunable parameters that control its cut management and optional enhancements:

- **WARMUP = 1:** Enables warm-start initialization with a heuristic tour. Helps CPLEX find early incumbents and improves the lower bound pruning process.
- **POSTING = 1:** Activates the posting of improved heuristic solutions during the callback execution. When triggered at shallow nodes (controlled via `DEPTH`), it patches and refines candidate solutions and submits them to CPLEX.
- **DEPTH = 100:** Sets the maximum allowed node depth for posting candidate heuristic solutions. Solutions at deeper nodes are not posted to avoid excessive overhead or low-quality updates, as we can see from the plot of figure 7.2.

- **CONCORDE = 1:** Enables the use of Concorde's cut separation routines at the root node. This allows the detection of fractional subtour violations via mincut procedures, and can greatly strengthen the initial LP relaxation.

In our experiments, enabling all options provided the most consistent improvements across diverse instance sizes. The combination of early warm-starts, selective posting, and strong root cuts allowed the solver to prune large parts of the search tree efficiently. The results are visible in figure 7.3

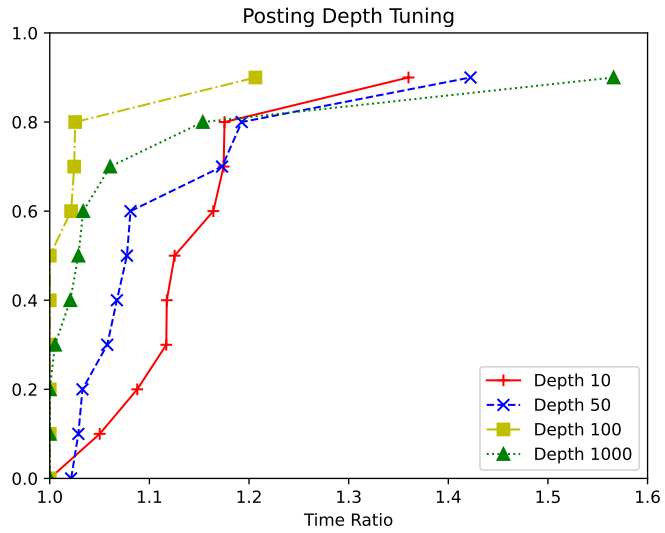


Figure 7.2: Posting Depth Tuning

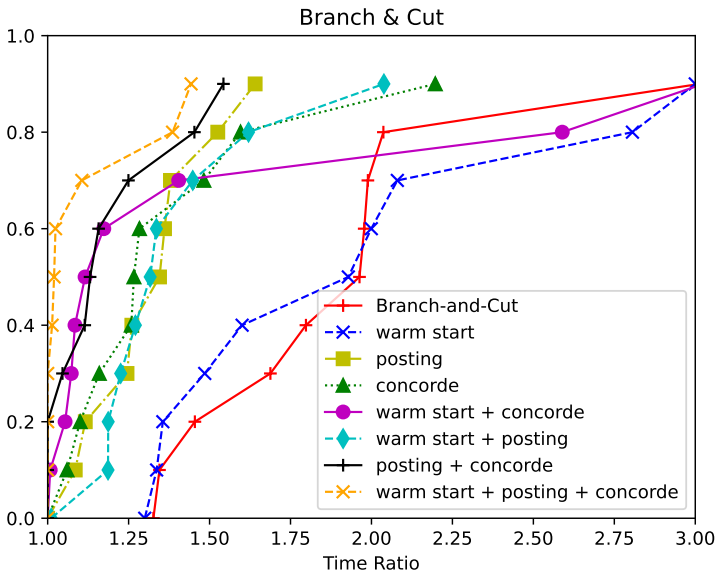


Figure 7.3: Branch and Cut tuning

7.12 Exact Algorithms Comparison

Finally, we have compared the two exact algorithms in their best configurations. We can see in figure 7.4 that, despite both algorithms are exact and arrive at the optimal solution in a certain amount of time, Branch-and-Cut outperforms Benders on all the instances.

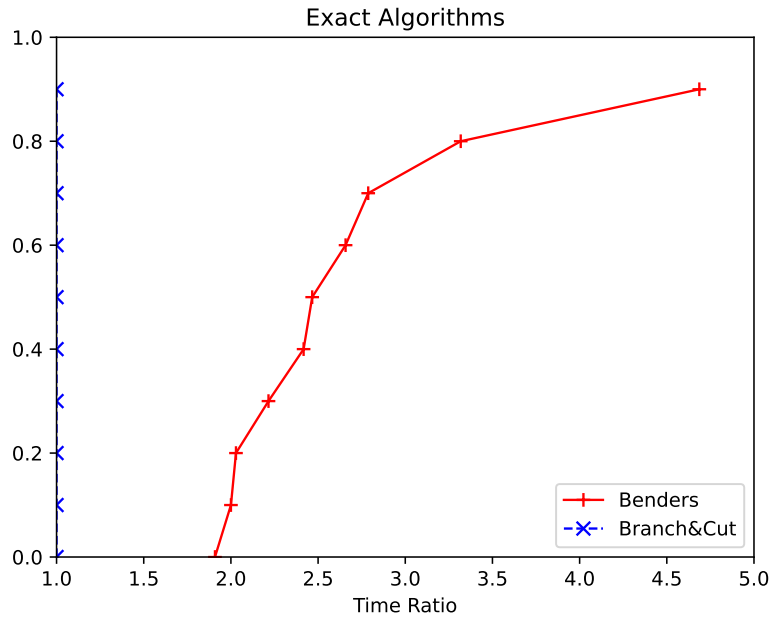


Figure 7.4: Exact Algorithms comparison

Chapter 8

Matheuristic

Exact methods can guarantee global optimality but are often computationally demanding. Matheuristics provide a powerful compromise by embedding heuristic principles within a mathematical programming framework. In this section, we describe two matheuristic strategies for the TSP that leverage the CPLEX solver while introducing problem-specific logic: Hard Fixing and Local Branching.

Both methods operate in a loop. At the beginning, they inject a high-quality solution (generated by a warm-start heuristic) into a modified TSP model. Then at each iteration the model is restricted using either variable fixing or a local neighborhood constraint, and re-optimized for a fixed time window. If a better solution is found, it becomes the new incumbent.

This iterative improvement scheme is inspired by hybrid MIP strategies and particularly by the local branching framework introduced by Matteo Fischetti and Andrea Lodi in the paper "Local Branching" [10].

8.1 Hard-Fixing

The Hard Fixing approach fixes a random subset of edges present in the current best tour. These fixed edges are enforced by setting the lower bound of corresponding binary variables x_e to 1. The subset is selected probabilistically, and the fixing probability P is controlled by two hyperparameters: `FIXEDPROB` determines whether a constant or decreasing schedule, based on the iteration number, is used, and `PROBABILITY` (or the array `PROBABILITIES`) sets the actual fixing rate. If the user choose to fix the probability at runtime the system will check that this value does not exceed the maximum value accepted for the probability, that is 90%.

The model is then re-optimized using CPLEX for a limited number of nodes, defined by the hyperparameter `CDEPTH`. This prevents excessive branching within each iteration while still allowing local improvements. After this search, the fixed variables are relaxed and the best

tour is updated if improved.

The idea of fixing a large subset of variables to anchor the solver in a promising region comes from the Relaxation-Induced Neighborhood Search scheme proposed by Danna, Rothberg and Le Pape [**danna2005**]. Their work demonstrates how hard fixing can be integrated with a MIP solver to achieve robust, high-quality improvements.

Algorithm 16: Hard Fixing Matheuristic

Input: TSP instance `inst` with incumbent tour, max time t

Output: Best feasible tour found

```
Generate initial heuristic tour s and set as warm start;
if inst → [FIXEDPROB] then
    P ← inst → [PROBABILITY];
end
while elapsed time < timelimit do
    if !inst → [FIXEDPROB] then
        Fix probability P based on current iteration;
    end
    Fix a random subset of tour edges with probability P;
    Solve modified model with CPLEX for a fixed node limit;
    newCost ← current solution cost ;
    if newCost < incumbent then
        incumbent ← newCost;
        save new tour;
    end
    Unfix all edge variables (reset lower bounds);
    Update warm start to s;
end
return best tour s;
```

8.1.1 Hyperparameters

For our implementation, fixing the probability didn't work well, so for brevity we report here just the tuning for the depth.

The Hard Fixing algorithm relies on probabilistically fixing edges from the current incumbent tour. The following parameters were used to control this mechanism:

- `FIXEDPROB = 0`: the fixing probability is gradually decreased throughout the search. If it were set to 1, a single fixed probability would be used throughout the search. Our tests

confirmed that the adaptive schedule leads to consistently better performance.

- `PROBABILITIES = {0.8, 0.5, 0.4, 0.2}`: At each restart, a different value from this array is selected in sequence, reducing the fixing intensity over time. Initially, 80% of the edges from the incumbent are fixed, strongly intensifying around the starting solution. Subsequent iterations relax this constraint, allowing more diversification and exploration.

- `CDEPTH = 7000`: This parameter sets the maximum number of nodes explored by CPLEX at each iteration. It bounds the computational effort spent inside each re-optimization loop and controls how deeply the solver is allowed to explore the solution space locally. Higher values allow for more intensive improvement, while lower values favor faster restarts.

This progressive unfixing mechanism helps the algorithm gradually widen the search space and escape local minima, while still leveraging strong incumbents found early in the process. Figure 8.2 shows the evolution of the incumbent cost under the Hard Fixing matheuristic. After a very rapid initial drop (driven by fixing 80 % of edges) the curve enters a long plateau as the fixing rate decreases (0.8→0.5→0.4→0.2). In the final iterations, with few edges fixed and full CPLEX effort, a second pronounced cost reduction occurs, yielding the best solution found.

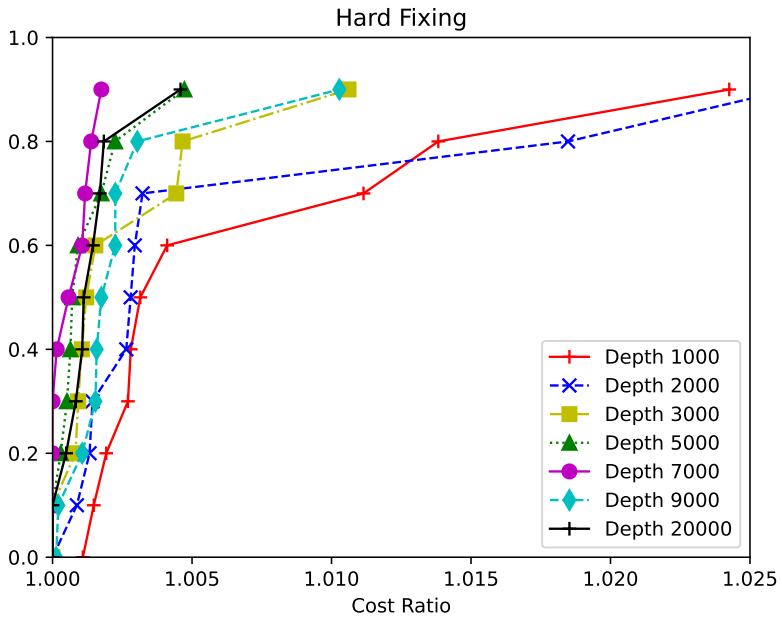


Figure 8.1: Hard Fixing tuning

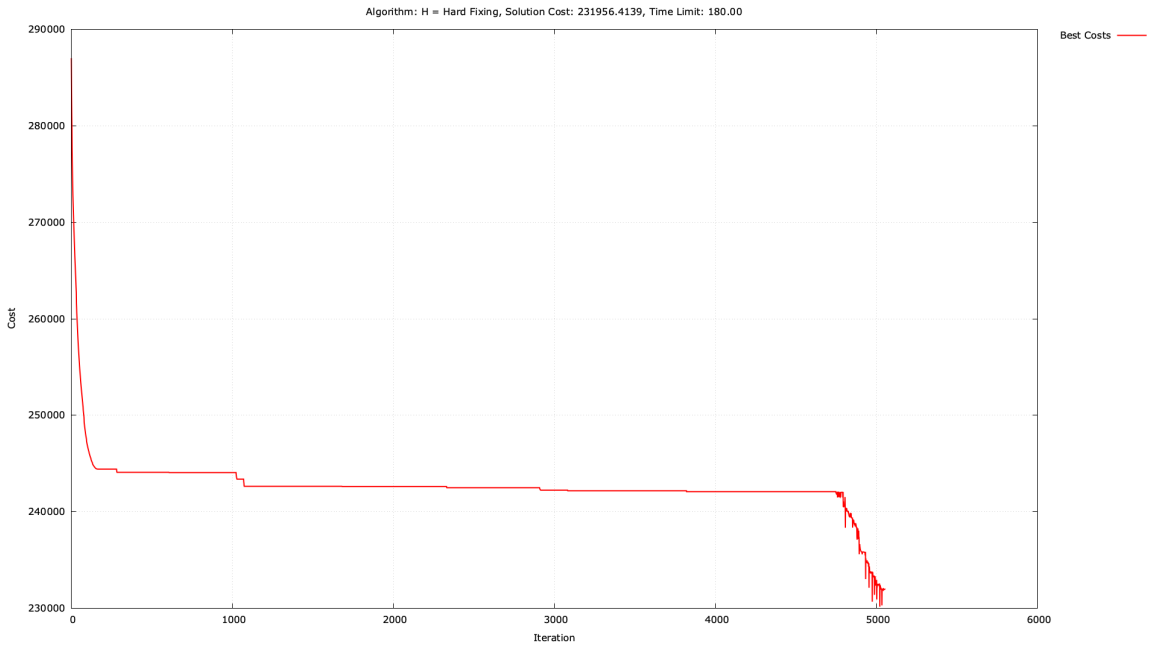


Figure 8.2: Hard Fixing history plot

8.2 Local Branching

The *Local Branching* method constrains the search to a neighborhood around the current incumbent tour. This is achieved by adding a single linear inequality—called a *local branching cut*—of the form:

$$\sum_{e \in x_e^H = 1} x_e \geq n - k$$

where n is the number of edges in the tour and k controls the size of the neighborhood.

The constraint ensures that at least $n-k$ of the edges in the incumbent are preserved in the new solution. The value of k, controlled via the hyperparameter `K_LOCAL_BRANCHING`, defines a so-called k-opt neighborhood, where the model is free to replace up to k edges in the tour.

Additionally, the hyperparameter `CDEPTH` sets the allowed depth for the solver's branch-and-bound tree during each neighborhood search. A higher value (e.g., 9000) encourages local optimization before restarting the search.

Unlike enumerative methods like 2-opt or 3-opt, which explicitly generate and evaluate candidate tours, local branching leverages the power of the MIP solver to implicitly explore neighborhoods with $k \approx 20$, which would be intractable to enumerate exhaustively.

The local branching cut is not globally valid but defines a subregion of the feasible space centered around the current solution x^H. The parameter k is a hyperparameter that controls the exploration vs. exploitation trade-off.

After solving the model with this constraint, the row must be removed from the problem before the next iteration. Since it is added as the last row, we simply delete the last row of the constraint matrix.

Algorithm 17: Local Branching Matheuristic

Input: TSP instance `inst` with incumbent tour, max time t

Output: Best feasible tour found

```
Generate initial heuristic tour s and set as warm start;
while elapsed time < timelimit do
    Add local branching constraint
     ∑_{e∈x_e^H=1} x_e ≥ n − inst → [K_LOCAL_BRANCHING];
    Solve modified model with CPLEX for a fixed node limit;
    newCost ← current solution cost ;
    if newCost < incumbent then
        incumbent ← newCost;
        save new tour;
    end
    Remove previous local branching constraint;
    Update warm start to s;
end
return best tour s;
```

8.2.1 Hyperparameters

The performance of Local Branching is primarily influenced by the radius of the neighborhood and the solver depth limit. The best results in our experiments were obtained using:

- **K_LOCAL_BRANCHING = 30:** This parameter controls the neighborhood size, defining how many edges from the incumbent solution can be changed. A value of 30 was found to offer a strong trade-off between local intensification and global exploration.
- **CDEPTH = 9000:** This value sets a soft upper bound on the depth of the CPLEX branch-and-bound tree while exploring a local branching neighborhood. Setting a high value encourages deeper search and better local exploitation before re-centering the neighborhood.

Values of $k < 20$ were too restrictive and often led to stagnation. Larger values ($k > 50$) allowed too much freedom and diminished the intensification effect of local branching. In figure

8.3 we can see the tuning of the CDEPTH parameter, while in figure 8.4 is reported the tuning of the K parameter.

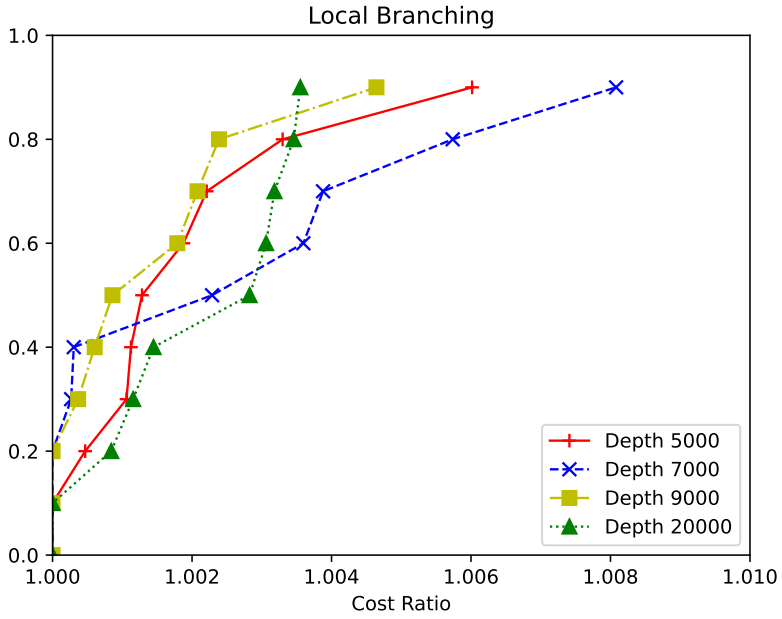


Figure 8.3: Local Branching depth tuning

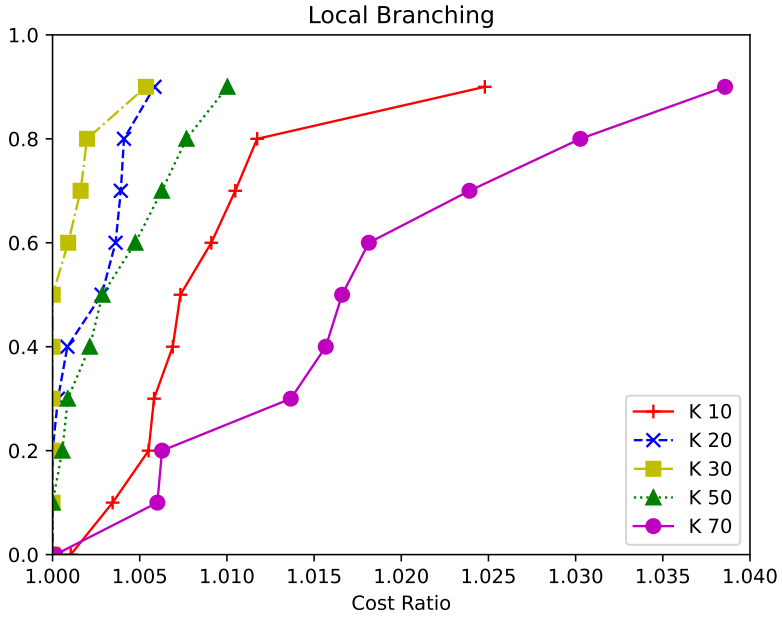


Figure 8.4: Local Branching tuning

8.3 Matheuristic Algorithms Comparison

Figure 8.5 shows the comparison between Local Branching and Hard Fixing in their best configurations. Looking at the figure, we can concluse that hard fixing is better with our implementation, with the given time limit (180 seconds) and the number of nodes (1000 nodes).

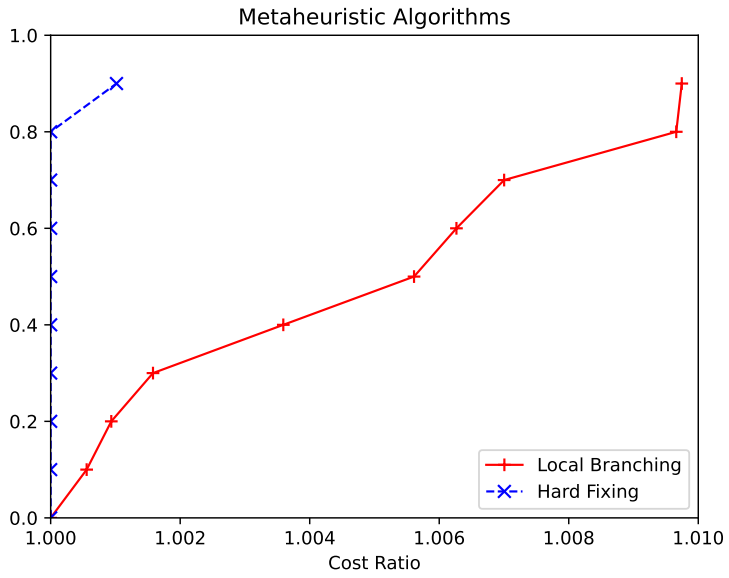


Figure 8.5: Matheuristic Algorithms Profiler plot

Chapter 9

Conclusions

In this work we have implemented, tuned, and compared a wide spectrum of algorithms for the symmetric Traveling Salesman Problem, ranging from simple construction heuristics to full-scale exact methods and matheuristic hybrids. Our main findings are:

- **Heuristics:** The Nearest Neighbour construction followed by a 2-OPT local search consistently outperformed the Extra-Mileage heuristic, achieving near-optimal tours in under one minute on 1000 node instances.
- **Metaheuristics:** Among GRASP, Genetic Algorithm, Tabu Search, and Variable Neighborhood Search, VNS emerged as the most robust and accurate within a 60s time limit, with Tabu Search a close second. GRASP offered fast initial solutions but slower convergence, while the plain Genetic Algorithm required much larger time ratios to match the other methods.
- **Exact Methods:** Both the classical Branch-and-Cut and our Benders-style cut-generation approach solved 300 node instances to proven optimality, but Branch-and-Cut, with warm start, posting heuristic and Concorde-based fractional cuts, was markedly faster and more reliable.
- **Matheuristics:** Hard Fixing and Local Branching both blend MIP re-optimization with heuristic guidance. Under a 180s time limit on 1000 node problems, Hard Fixing delivered superior tour quality by progressively relaxing a high-probability edge-fixing schedule, whereas Local Branching required careful tuning of neighborhood radius to avoid stagnation.

The common data structures, performance profiler, and CSV-based benchmarking framework developed here ensured a fair and reproducible comparison across all methods. Parameter

tuning via Dolan–Moré performance profiles was crucial to uncover each algorithm's best settings.

Among the different algorithms, we compared the top three non-exact methods on 50 instances of 1000 nodes with 180s time limit to see which performs better. We can see in figure 9.1 that Hard Fixing outperforms the other algorithms. By temporarily fixing a large portion of edges from a high-quality incumbent tour, it reduces the combinatorial complexity and then calls CPLEX on the smaller subproblem. Alternating fixation phases with global re-optimizations maintains both intensification and diversification, ensuring rapid, high-quality improvements within the 180s time limit on instances of 1000 nodes.

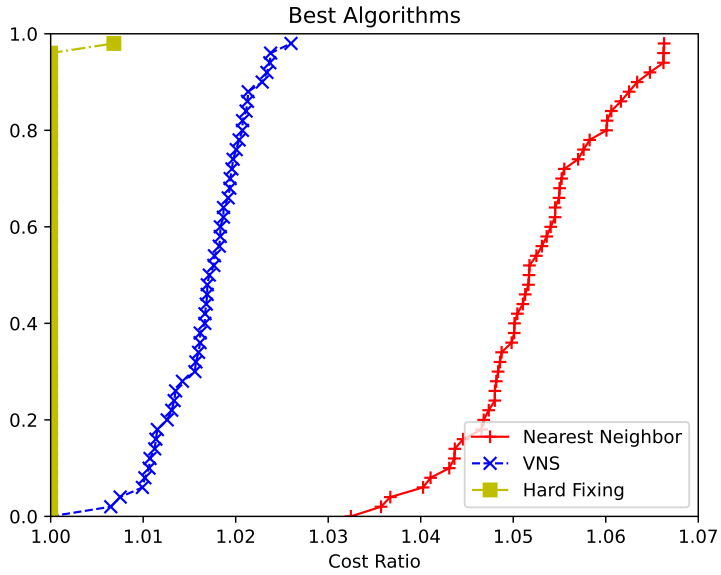


Figure 9.1: Best non-exact algorithms comparison

Overall, this comparative study highlights the trade-offs between speed, solution quality, and provable optimality, offering practical guidance for selecting and combining algorithms in TSP applications. Additional improvements could involve integrating Machine Learning techniques within an Operations Research framework to yield dynamic, data-driven decision rules. For instance, Carrizosa et al. [4] propose a counterfactual optimization model that embeds neural classifiers inside a mixed-integer programming formulation to generate real-time control strategies for complex energy systems. Incorporating similar ML-driven predictive components into TSP solvers could further enhance adaptive performance and robustness. Future work might therefore investigate hybrid ML–OR architectures to advance large-scale, high-speed TSP solution methods.

Bibliography

[1] D. Applegate et al. *Concorde TSP Solver*. http://www.math.uwaterloo.ca/tsp/concorde.html. Accessed: 2025-05-30. 2006.

[2] D. Applegate et al. "The Traveling Salesman Problem: A Computational Study." In: *The Traveling Salesman Problem: A Computational Study* (Jan. 2006).

[3] J.F. Benders. "Partitioning procedures for solving mixed-variables programming problems." In: *Numerische Mathematik* 4 (1962/63), pp. 238–252. URL: http://eudml.org/doc/131533.

[4] E. Carrizosa et al. "Counterfactual Optimization for Fault Prevention in Complex Wind Energy Systems." In: *European Journal of Operational Research (preprint)* (May 2025). Preprint.

[5] G. A. Croes. "A method for solving traveling-salesman problems." In: *Operations Research* 6.6 (1958), pp. 791–812.

[6] G. Dantzig, R. Fulkerson, and S. Johnson. "Solution of a large-scale traveling-salesman problem." In: *Journal of the Operations Research Society of America* 2.4 (1954), pp. 393–410.

[7] D. Davendra, ed. *Traveling Salesman Problem: Theory and Applications*. IntechOpen, 2010. ISBN: 978-953-307-426-9. DOI: 10.5772/38734.

[8] E. D. Dolan and J.J. Moré. “Benchmarking optimization software with performance profiles.” In: *Mathematical Programming* 91.2 (2002), pp. 201–213.

[9] T. Feo. “Greedy Randomized Adaptive Search Procedures.” In: *Journal of Global Optimization* 6 (Mar. 1995), pp. 109–133. DOI: 10.1007/BF01096763.

[10] M. Fischetti and A. Lodi. “Local branching.” In: *Mathematical Programming* 98 (Sept. 2003), pp. 23–47. DOI: 10.1007/s10107-003-0395-5.

[11] F. Glover and M Laguna. *Tabu search I*. Vol. 1. Jan. 1999. ISBN: 978-0-7923-9965-0. DOI: 10.1287/ijoc.1.3.190.

[12] W. R. Hamilton. “Account of the Icosian Calculus.” In: *Proceedings of the Royal Irish Academy* 6 (1858), pp. 415–416.

[13] IBM Corporation. *IBM ILOG CPLEX Optimization Studio*. https://www.ibm.com/products/ilog-cplex-optimization-studio. Accessed: 2025-05-30. 2025.

[14] S. Lin. “Computer solutions of the traveling salesman problem.” In: *Bell System Technical Journal* 44.10 (1965), pp. 2245–2269.

[15] S. Lin and B. W. Kernighan. “An Effective Heuristic Algorithm for the Traveling-Salesman Problem.” In: *Operations Research* 21.2 (1973), pp. 498–516.

[16] N. Mladenović and . Hansen. “Variable neighborhood search.” In: *Computers Operations Research* 24.11 (1997), pp. 1097–1100. ISSN: 0305-0548. DOI: https://doi.org/10.1016/S0305-0548(97)00031-2. URL: https://www.sciencedirect.com/science/article/pii/S0305054897000312.

[17] J.V. Potvin. “Genetic Algorithms for the Traveling Salesman Problem.” In: *Annals of Operations Research* 63 (1996), pp. 337–370. DOI: 10.1007/BF02125403. URL: https://doi.org/10.1007/BF02125403.

[18] G. Reinelt. "TSPLIB - A Traveling Salesman Problem Library." In: *INFORMS J. Comput.* 3.4 (1991), pp. 376–384. DOI: 10.1287/IJOC.3.4.376. URL: https://doi.org/10.1287/ijoc.3.4.376.

[19] D. Rosenkrantz, R. Stearns, and Philip II. "An Analysis of Several Heuristics for the Traveling Salesman Problem." In: *SIAM J. Comput.* 6 (Sept. 1977), pp. 563–581. DOI: 10.1137/0206041.

Appendix A

IBM ILOG CPLEX Installation

Both MacOS and Windows need the software IBM ILOG CPLEX to be installed on the device, so:

1. **Download** ILOG CPLEX Optimization Studio at the following link: `https://academic.ibm.com/a2mt/downloads/data_science#/`.
2. Open the executable installer and follow the instructions to **install CPLEX**.

A.1 Windows Installation

1. **Find your CPLEX installation path.**

   Typically, CPLEX is installed in:

   ```
   C:\Program Files\IBM\ILOG\CPLEX_StudioXXXX\
   ```

   Replace `XXXX` with your actual version (e.g., `2211` for version 22.1.1).

   You will need:

   - The **include directory**, e.g.:

     ```
     C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\include\
        ilcplex
     ```

   - The **library directory**, e.g.:

     ```
     C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\
        x64_windows_vs2013\stat_mda
     ```

2. **Add the paths to your system environment variables.**

(a) Press `Win + S`, search for `Environment Variables`, and open `Edit the system environment variables`.

(b) In the `System Properties` window, click on `Environment Variables...`.

(c) Under **System variables**, scroll and select the `Path` variable. Click `Edit`.

(d) Click `New` and add the following paths (adjust if your version is different):

- `C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\bin64_windows_vs2013`
- `C:\Program Files\IBM\ILOG\CPLEX_Studio2211\concert\bin64_windows_vs2013`

(e) Click `OK` to confirm all dialogs.

A.2 Mac Installation

1. **Add CPLEX to the environment path**

   Add the following lines to your shell configuration file

```
export CPLEX_HOME=/usr/local/Cellar/cplex/<version>
export PATH=$CPLEX_HOME/bin:$PATH
```

A.3 Configuring Visual Studio Code for CPLEX

To use CPLEX in Visual Studio Code, you need to create a configuration file named `c_cpp_properties.json` inside the `.vscode` folder of your project. This file tells VS Code where to find the header files.

Below is an example configuration file with separate settings for macOS and Windows.

```
{ "configurations": [
        {
            "name": "Mac",
            "includePath": [
                "${workspaceFolder}/**",
                "/Applications/CPLEX_Studio2211/cplex/include/**",
                "/Applications/CPLEX_Studio2211/concert/include/**"
            ],
            "macFrameworkPath": [
                "/System/Library/Frameworks",
                "/Library/Frameworks"
            ],
            "intelliSenseMode": "macos-clang-arm64",
            "compilerPath": "/usr/bin/clang",
            "cStandard": "c17",
            "configurationProvider": "ms-vscode.cmake-tools"
        },
        {
        "name": "Windows",
        "includePath": [
            "${workspaceFolder}/**",
            "C:/Program Files/IBM/CPLEX_Studio2211/cplex/include/**",
            "C:/Program Files/IBM/CPLEX_Studio2211/concert/include/**"
        ],
        "intelliSenseMode": "windows-msvc-x64",
        "compilerPath": "C:/Program Files (x86)/Microsoft Visual
        ↪ Studio/2019/Community/VC/Tools/MSVC/14.29.30133/bin/Hostx64/x64/cl.exe",
        "cStandard": "c17",
        "configurationProvider": "ms-vscode.cmake-tools"
        }
    ],
    "version": 4}
```

Code A.1: VSCode CPLEX configuration example

Appendix B

Utility Functions

Algorithm 18: `xpos`

Input: index i of node a, index j of node b,
number of nodes in the tour nnodes
Output: position pos in CPLEX format

```
1 if i == j then
2 |   print_error("i == j in xpos");;
3 end
4 if i > j then
5 |   return xpos(j,i,nnodes);
6 end
7 int pos = i * nnodes + j - ((i + 1) * (i + 2))/2;
8 return pos;
```

Algorithm 19: `build_model`

Input: instance *inst, CPXENVptr env, CPXLPptr lp

Output: CPLEX model with binary edge variables and degree constraints

```
1  n ← inst → nnodes allocate memory for cname, index, value;
2  for i ← 0 to n do
3  |   for j ← i + 1 to n do
4  |   |   compute distance obj ← distance(i, j);
5  |   |   create binary variable x_e for edge e = (i, j) with obj, lb=0, ub=1;
6  |   end
7  end
8  for h ← 0 to n do
9  |   nnz ← 0;
10 |   for i ← 0 to n i ≠ h do
11 |   |   index[nnz] ← xpos(i, h, n);    value[nnz] ← 1.0;    nnz ← nnz + 1;
12 |   end
13 |   add row "degree(h)" with columns index[0..nnz-1], values value[0..nnz-1],
   |    sense 'E', rhs=2;
14 end
15 free memory for cname, index, value;
```

Algorithm 20: `build_sol`

Input: const double *xstar, instance *inst

Output: arrays `succ`, `comp`, and number of components `ncomp`

```
1  if DEBUG then
2  |  verify each node has degree 2 in xstar;
3  end
4  n ← inst → nnodes ncomp ← 0;
5  initialize all succ[i] and comp[i] to −1;
6  for start ← 0 to n do
7  |  if comp[start] < 0 then
8  |  |  ncomp ← ncomp + 1;
9  |  |  i ← start;   done ← false;
10 |  |  while !done do
11 |  |  |  comp[i] ← ncomp;
12 |  |  |  if there exists j ≠ i with xstar[xpos(i, j)] > 0.5 and comp[j] == −1 then
13 |  |  |  |  succ[i] ← j;   i ← j;
14 |  |  |  end
15 |  |  |  else
16 |  |  |  |  done ← true
17 |  |  |  end
18 |  |  end
19 |  |  succ[i] ← start;
20 |  end
21 end
```