Università
degli Studi
di Padova

Dipartimento
di Ingegneria
dell'Informazione

Operations Research 2

Optimization algorithms for the Traveling Salesman Problem

Riccardo Zerbinati, Alessandro Dario, Jaime Candau

Academic year 2024/2025

Contents

1 Introduction

The Traveling Salesman Problem (TSP) is a foundational problem in combinatorial optimization. It asks for the shortest possible route that visits each node in a given graph exactly once and returns to the starting position. Despite its simple formulation, the TSP is NP-hard, meaning that solving large instances optimally is computationally challenging. Historically, the origins of the TSP can be traced back to the 19th century, notably to W.R. Hamilton's work, but it became a formal topic of study in the 20th century. One of the first major breakthroughs came in the 1950s, when Dantzig, Fulkerson, and Johnson used linear programming to solve a 49-node instance.

Decades later, William Cook and collaborators developed the Concorde TSP Solver, capable of solving instances with tens of thousands of nodes. Today the TSP continues to serve as a benchmark problem due to its computational complexity and relevance in areas such as logistics, bioinformatics and electronics.

[1] [2] [3] [4] [5] [3] [6]

2 Tools and Implementation Environment

2.1 C Programming Language

The C programming language is a general-purpose, low-level language that offers precise control over memory and system resources. It was developed in the early 1970s by Dennis Ritchie at Bell Labs as part of the development of the UNIX operating system. C was designed to provide an efficient and portable way to implement system software, including operating systems and compilers, while retaining enough abstraction to make it easier than assembly.

C has remained highly influential and is still widely used today in systems programming, embedded systems, and performance-critical applications. Many modern languages, including C++, Java, and Rust, are either based on or strongly influenced by C. For this project, C was chosen due to its speed, low-level control, and suitability for implementing custom optimization routines and memory-intensive algorithms.

[7].

2.2 CMake

CMake is a powerful cross-platform build system generator developed by Kitware in the year 2000, primarily by Bill Hoffman and Ken Martin. It simplifies the process of compiling large-scale C and C++ projects by allowing developers to describe build requirements in platform-independent configuration files CMakeLists.txt. From this input, CMake generates native build files (e.g., Makefiles or Visual Studio projects) tailored to the user's system.

CMake is particularly valuable when working with external libraries, such as Cplex or Concorde, because it offers fine-grained control over linking, dependencies, and build targets. In this project, CMake was used to automate the build process, manage compiler settings, and ensure portability across systems. Its flexibility and wide adoption make it the standard tool for C/C++ projects in both academia and industry.

[7]

2.3 Gnuplot for plotting

Gnuplot is a command-line graphing utility developed in 1986 by Thomas Williams and Colin Kelley. Although its name suggests a connection to the GNU Project, it was created independently. Its original purpose was to provide a portable, open-source tool for plotting mathematical functions and data across various platforms, which made it popular in scientific and engineering communities from the start.

Over time, Gnuplot gained relevance in programming due to its simplicity, lightweight nature, and the ability to integrate with many programming languages such as Python, C/C++, Fortran and Perl. It is particularly appreciated for producing high-quality publication ready plots and being scriptable, which makes it ideal for automated workflows.

Gnuplot is often used to visualize data from simulations, benchmarks or optimization problems in research environments. For instance, after computing a solution, the list of nodes (represented by their coordinates) can be plotted to visually inspect the path. We also use it to plot the evolution of solution costs across steps for iterative algorithms.

[8]

2.4 Cplex

Cplex is a high-performance commercial solver developed by IBM for solving linear programming (LP), mixed-integer programming (MIP) and quadratic programming (QP) problems. It is widely used in operations research and optimization and offers APIs in C, C++, Python, Java and other languages. In this work, we use the C interface of Cplex to implement and benchmark exact algorithms for the Traveling Salesman Problem (TSP).

Installation. Installing Cplex is straightforward: the user must obtain an academic or commercial license from the official IBM website and download the Cplex Optimization Studio. No further installation steps are typically required beyond unpacking and registering the license. The installed directory serves as the base for compilation.

CMake integration. Our project uses a custom `FindCplex.cmake` module placed in the `cmake/` directory. The build system is configured with CMake (version 3.20 or higher) and the solver is linked as follows:

```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/")
find_package(\textsc{Cplex} REQUIRED)
include_directories(${\textsc{Cplex}_INCLUDE_DIRS})
...
target_link_libraries(tsp PRIVATE ${\textsc{Cplex}_LIBRARIES})
```

The full CMake configuration also includes automatic source discovery, support for address sanitizers in debug mode and linking with other components such as Concorde. This modular setup ensures portability and straightforward integration with the Cplex library across platforms.

[9]

2.5 Concorde TSP Solver

Concorde is a state-of-the-art exact solver for the Traveling Salesman Problem (TSP), developed by David Applegate, Robert Bixby, Vasek Chvátal and William Cook. It implements one of the most effective branch-and-cut algorithms for solving TSP instances to proven optimality and is widely regarded as the benchmark solver for symmetric TSPs.

The Concorde software is written in `C` and integrates multiple algorithmic components, including:

- Linear programming relaxations of the TSP formulation,
- Cutting plane generation (e.g., subtour elimination, comb inequalities, clique cuts),
- Branch-and-bound search,
- Heuristics for initial tour construction (e.g., chained Lin-Kernighan).

It is capable of solving instances with several thousand nodes and includes both a command-line interface and callable library routines.

To integrate Concorde into our own codebase, we compiled it as a static library and linked it via `CMake`. We added the necessary paths and flags in our project's `CMakeLists.txt`, allowing us to call selected Concorde functions from our C code.

Use in this work. In this project we did not use Concorde as a full solver. Instead, we extracted and reused its internal routines for **separating fractional cuts**, which is a key step in our own branch-and-cut implementation. Specifically, we integrated:

- Finding connected components of the residual graph.
- Flow algorithm to identify violating flows in the residual graph.

By isolating the cut separation logic, we were able to take advantage of Concorde's efficiency in identifying meaningful cuts, without relying on its complete solving pipeline.

[10] [11] [12]

2.6 Performance Profiles

The concept of performance profiles was introduced by Elizabeth Dolan and Jorge Moré in 2002 in their influential paper *Benchmarking Optimization Software with Performance Profiles* [13]. Their goal was to create a fair, normalized methodology to compare the performance of multiple optimization algorithms across a common set of test problems. Since then, performance profiles have become a standard benchmarking tool in both numerical and combinatorial optimization. They provide an intuitive and visual comparison of how consistently and efficiently each method performs over a benchmark suite.

Formal Definition

A performance profile is a cumulative distribution function that shows, for each algorithm, the proportion of problems for which its performance is within a factor $\tau \geq 1$ of the best algorithm's performance.

1. Let P be a set of test problems and S a set of solvers (algorithms).
2. Let $t_{p,s}$ be the performance of solver $s \in S$ on problem $p \in P$ (e.g., time, cost, iterations, or solution quality).

3. Define the **performance ratio**:

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

   so that the best solver for each problem has $r = 1$.

4. The **performance profile** of solver s is:

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

   which gives the fraction of problems that solver s can solve within a factor τ of the best solver.

Application in our Project

In this thesis, we employ performance profiles to systematically compare both **heuristic** and **exact** algorithms for the Traveling Salesman Problem (TSP), as well as to tune hyperparameters and select the most effective variants. Testing is done on a randomly generated set of instances.

Heuristic algorithms. Heuristic algorithms typically operate under a fixed time budget and may return suboptimal solutions. To benchmark them, we:

- Fix problem size $n = 1000$,
- Fix time limit $T = 1$min for constructive heursitics, $T = 3$min for solver-aided matheuristics.
- Run multiple algorithms (or configurations) on a shared instance set,
- Record the quality of the solutions each algorithm returns,
- Construct performance profiles based on these quality metrics.

Exact algorithms. Exact methods always return the optimal solution, so solution quality is not a useful metric. Instead, we:

- Fix problem size $n = 300$,
- Fix time limit $T = 3$min, though we expect algorithms to terminate before hitting the time limit,
- Run each exact algorithm on the same instance set,
- Record the time required to reach the optimal solution,
- Use performance profiles based on time to determine which solver is most efficient.

All tests were conducted on the same hardware (Apple M3 chip, 16GB RAM). Relevant plots were produced with a script created by Domenico Salvagnin, with slight modifications to the original code.

3 Heuristics

Heuristic algorithms offer efficient ways to tackle the Traveling Salesman Problem by finding good-enough solutions without guaranteeing optimality. Unlike exact global search, which is often too slow for large instances, heuristics focus on speed and practicality. They generally fall into two types: constructive methods, which build a solution from scratch, and refinement methods, which improve an existing one.

3.1 Nearest Neighbor

Nearest Neighbor (NN) is a constructive heuristic for the Traveling Salesman Problem. It likely appeared informally in the 1950s during the early development of combinatorial optimization, but was explicitly studied and used in algorithmic form from the 1960s onward. It became one of the first algorithms implemented in early computational experiments for routing problems, due to its simplicity and feasibility even with limited computing power. The method gained traction as a baseline algorithm and is still used today for educational, benchmarking, and initialization purposes.

The algorithm builds a tour incrementally by making a greedy choice at every step: from the current node, it selects the nearest unvisited neighbor and adds it to the tour. This process repeats until all nodes have been visited, and finally, the tour is closed by returning to the starting node.

Despite its simplicity, the algorithm does not guarantee optimality. In fact, its worst behavior typically occurs in the last iteration, where the only remaining move is to close the tour, often resulting in a long edge. This edge can disproportionately affect the total cost.

The time complexity is $\mathcal{O}(n^2)$ in a generic graph, since at each of "n" steps the algorithm must scan the remaining unvisited nodes to find the closest one.

Nearest Neighbor has very few **Hyperparameters**, making it attractive for quick prototyping or comparisons:

- **Start node:** the node from which the tour begins. This greatly affects the result. In our implementation, a random starting node is chosen.
- **Tie-breaking rule:** in case multiple nearest neighbors have equal distance, we simply choose the lowest-indexed neighbor.

Pseudocode

```
procedure NearestNeighbor(dist, $s$)
    Input:
    dist$[i, j]$: symmetric $n \times n$ distance matrix
    $s$: starting node index
    Output: a tour as a permutation of $0, \dots, n-1$
    mark all nodes unvisited
    tour ← [ ]
    current ← $s$
    mark $s$ visited; append $s$ to tour
    while $\exists$ an unvisited node do
        $v \leftarrow \arg\min\{\text{dist}[\text{current}, u] : u \text{ unvisited}\}$        ▷ break ties arbitrarily
        mark $v$ visited; append $v$ to tour
        current ← $v$
    end while
    append $s$ to tour        ▷ close the cycle
    return tour
end procedure
```

[14] [6]

3.1.1 All Starts variant

The All Starts variant of Nearest Neighbor (NN) is a natural extension of the basic greedy heuristic for the Traveling Salesman Problem. This variant emerged from the well-known observation that the quality of the NN tour is heavily dependent on the choice of starting node. To mitigate this variability, the All Starts strategy runs the NN algorithm once from every possible starting node, and returns the shortest resulting tour. This approach eliminates the need to treat the starting node as a hyperparameter, and produces a more robust result. Formally, if the input graph has "n" nodes, the All Starts algorithm:

1. Executes Nearest Neighbor starting from each node $v_1, v_2, \dots, v_n$.
2. After each run, it records the total cost of the tour.
3. Finally, it selects the tour with minimum cost among all n candidates.

The computational complexity becomes $O(n^3)$ since for each of the "n" start nodes, the $\mathcal{O}(n^2)$ NN procedure must be executed. This makes the algorithm significantly slower than basic NN, especially for large instances.

In our implementation, to maintain practical runtimes, a timeout condition is enforced. After each NN run, the algorithm checks whether enough time remains to execute another iteration, and terminates early if necessary.

We observed that combining All Starts with 2-opt results in significantly better tours. For this reason, it's often beneficial to treat All Starts not as a complete algorithm but as a first phase of a broader heuristic pipeline.

3.2 Extra mileage

Extra Mileage is a greedy constructive heuristic for the Traveling Salesman Problem (TSP). It belongs to the family of insertion-based methods studied since the 1970s, and is related in principle to the Farthest Insertion heuristic described in early TSP literature. It shares structure with methods proposed in Frieze, Clarke-Wright, and Reinelt's computational analyses. While not attributed to a specific author, the method follows a rigorous insertion strategy based on minimizing extra tour cost. The algorithm starts with the two farthest nodes in the graph, forming an initial tour of the form:

$$\text{tour} = [i, j, i]$$

At each iteration:

1. For each unvisited node i and each edge (i, j) in the tour we compute the extra mileage
$$\Delta_{ihj} = c(i, h) + c(h, j) - c(i, j)$$
2. Insert h between i and j where Δ_{ihj} is minimal.

The process repeats until all nodes are inserted or a timeout is reached; if time expires, the tour is completed by randomly inserting remaining nodes. The algorithm checks all node-edge pairs, leading to a complexity of $\mathcal{O}(n^3)$ Since the initial path is fixed, no all-starts variant is used.

Although Extra Mileage builds better raw tours than Nearest Neighbor, we observed that after applying 2-opt "NN + 2-opt" often yields better solutions then "EM + 2-opt".

Pseudocode

```
procedure ExtraMileage(dist, $T_{\max}$)
    Input:
  dist$[i, j]$: symmetric $n \times n$ distance matrix
  $T_{\max}$: time limit (seconds)
    Output: a tour permutation of $0, \dots, n-1$
    find pair $(i, j)$ with maximum dist$[i, j]$
    tour $\leftarrow$ $[\, i,\ j,\ i \,]$
    mark all nodes except $i, j$ as unvisited
    while $\exists$ unvisited node do
        $(h, i, j) \leftarrow \arg\min_{\substack{w \in \text{Unvisited} \\ (u,v) \in \text{Edges}(tour)}} \big(\text{dist}(u, w) + \text{dist}(w, v) - \text{dist}(u, v)\big)$
        insert node $h$ between $i, j$ in tour
        mark $h$ visited
    end while
    return tour
end procedure
```

[15] [11]

3.3 Optimality moves

2-opt is a local search algorithm and a classic refinement heuristic for the Traveling Salesman Problem. Originally proposed by G. A. Croes (1958), it remains one of the most widely used improvement procedures in combinatorial optimization due to its simplicity and effectiveness. The algorithm begins from a feasible tour and iteratively improves it by performing edge swaps that reduce total cost.

The core idea of 2-opt is to define a neighborhood around a given tour by considering all pairs of edges and evaluating the result of removing those edges and reconnecting the tour by swapping their ends. At each iteration:

- For all valid pairs of edges $(i, i+1)$, $(j, j+1)$, compute the cost delta between the current configuration and the swapped one:

$$\Delta = c(i, j) + c(i+1, j+1) - c(i, i+1) - c(j, j+1)$$

- Select the best pair (i, j) that yields the maximum reduction in cost (best improvement).
- Apply the move by reversing the subtour between $i + 1$ and j.

This process continues until no further improving swap exists, resulting in a local minimum in the 2-opt neighborhood.

The complexity of each iteration is $\mathcal{O}(n^2)$, since all edge pairs must be checked. However, there is no guarantee on the number of iterations required, which depends on the tour and instance size. For this reason, a timeout condition is included in our implementation.

Our version uses the best-improvement strategy (checking all pairs and selecting the best), rather than the first-improvement alternative, which immediately accepts the first improving move. While slightly more expensive, best-improvement tends to yield better final solutions.

Pseudocode

```
procedure TwoOptImprove(tour, dist, $T_{\max}$)
    Input:
    tour: initial permutation of $0, \ldots, n-1$
    dist$[i, j]$: symmetric distance matrix
    Output: improved tour
    $t_0 \leftarrow$ current time
    repeat
        $(i, j, \Delta_{\min}) \leftarrow \arg\min_{0 \le i < j-1 < n} \big(\text{dist}(t_i, t_j) + \text{dist}(t_{i+1}, t_{j+1}) - \text{dist}(t_i, t_{i+1}) - \text{dist}(t_j, t_{j+1})\big)$
        if $\Delta_{\min} < 0$ then
            reverse the segment of tour from $i + 1$ to $j$
        else
            break                                        ▷ local optimum reached
        end if
    until timeout
    return tour
end procedure
```

[14]

3.4 Grasp

GRASP (Greedy Randomized Adaptive Search Procedure) is a multi-start metaheuristic that combines greedy construction with randomization to build diverse initial solutions, which are typically refined later via local search. It was introduced by Feo and Resende in the late 1980s, and has since been applied successfully to a wide range of combinatorial problems, including the Traveling Salesman Problem (TSP).

Our implementation of GRASP can be viewed as a randomized version of Nearest Neighbor, where at each construction step the algorithm chooses from a restricted candidate list (RCL) instead of deterministically picking the closest node.

The tour is built incrementally, starting from a random node. At each iteration:

1. Candidate selection: A subset of promising candidate nodes is created based on lowest insertion cost.

2. A node is selected from the RCL randomly, with probabilities inversely proportional to the insertion cost.

3. The selected node is appended to the tour.

To efficiently manage selection, we store potential successors in a heap, which allows for quick updates and cost-based sorting. We balance exploration and exploitation by using greedy selection on some iterations, randomized selection on others.

Hyperparameters;

- **RCL size:** Determines how many of the lowest-cost nodes are eligible for random selection at each step.
- **Random selection probability:** Probability of using randomized selection instead of the greedy NN selection.

[16]

Pseudocode

```
procedure GRASP(dist, $r$, $p$)
    Input:
    dist$[i, j]$: symmetric $n \times n$ distance matrix
    $r$: size of the Restricted Candidate List (RCL)
    $p$: probability of random selections
    Output: best-found tour
    best_tour ← NIL, best_cost ← $+\infty$
    repeat
        tour ← [ ]
        $s$ ← random start node
        mark $s$ visited; append $s$ to tour
        while there exists unvisited node do
            for each unvisited $u$, compute insertion cost $\Delta_u$
            if random with probability $p$ then
                form RCL of the $r$ nodes with smallest $\Delta_u$          ▷ e.g. via heap
                select $v \in$ RCL at random, weighted by $1/\Delta_u$
            else
                $v \leftarrow \arg\min_u \Delta_u$
            end if
            mark $v$ visited; append $v$ to tour
        end while
        append start node $s$ to tour                                       ▷ close the loop
        $c$ ← cost(tour)
        if $c <$ best_cost then
            best_tour ← tour; best_cost ← $c$
        end if
    until timeout
    return best_tour
end procedure
```

Tuning

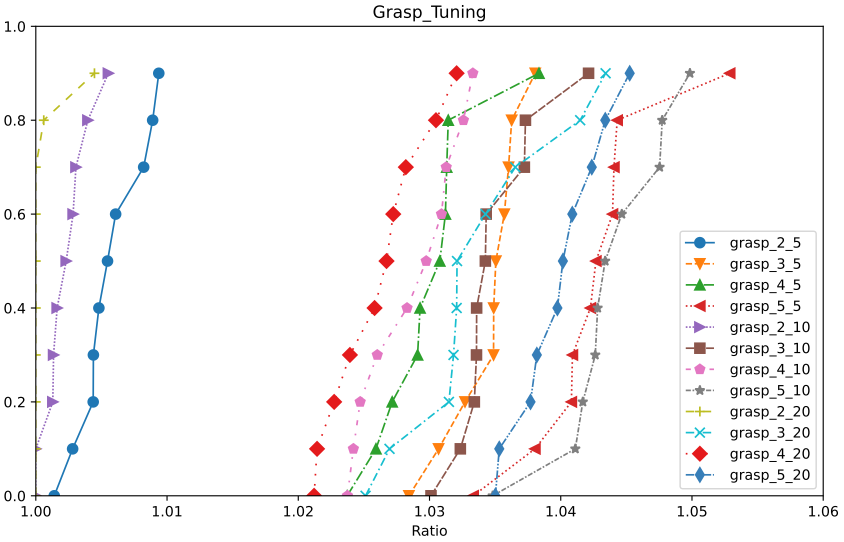

FIGURE 1: Parameters: (1) RCL size (2) every how many iterations do a randomized choice.

Tuning definitely favors a small RCL size, and fewer randomized choices.

3.5 Variable neighborhood search

Variable Neighborhood Search (VNS) was first proposed by Nenad Mladenović and Pierre Hansen in 1997 in their influential paper “Variable Neighborhood Search”, published in “Computers & Operations Research”. Their goal was to design a unified framework that would overcome the limitations of traditional local search methods, particularly their tendency to get trapped in local optima. It is now widely used in operations research, logistics, and scheduling.

VNS is a metaheuristic designed to escape poor local minima by dynamically changing the neighborhood structure during the search. Unlike basic local search algorithms like 2-opt, which can become trapped in local optima, VNS introduces randomized perturbations called kicks to explore other regions of the solution space.

VNS is initialized with a solution found by the NN heuristic, then operates by alternating between two phases:

1. **Kick phase** (diversification): The current tour is modified using a sequence of random 3-opt moves designed to escape the 2-opt neighborhood, that cannot be easily reversed by 2-opt steps. The number of 3-opt moves is controlled by a parameter k, which defines the intensity of the perturbation.

2. **Local search phase** (intensification): After the kick, the modified solution is refined with 2-opt, searching for a new local minimum.

This loop continues until a time limit is reached. The purpose of the kick is to restart local search in a new basin of attraction, increasing the chance of finding better optima.

Hyperparameters:

- **k (kick strength):**
  - Determines how many random 3-opt moves are applied.
  - A low k introduces small perturbations but may return to the same minimum.
  - A high k ensures diversity but may degrade solution quality excessively, effectively restarting the algorithm from scratch.

- **Incremental adaptation of k:** We also attempted a dynamic schedule for k:
  - If cost improves we found a better local minima: k → reset to the minimum for exploitation.

- If cost worsens we found a worse local minima: $k \to$ decrease slightly to exploit the new neighborhood.
- If cost stagnates we are back to the same local minima: $k \to$ increase up to a cap since we need more exploration.

[17] [18] [19]

Pseudocode

```
procedure VNS(tour, dist, $k$)
    Input:
      tour: initial permutation of $0, \dots, n-1$
      dist$[i, j]$: symmetric distance matrix
      $k$: kick strength
    Output: best-found tour
    best_tour ← tour
    repeat
        candidate ← Kick(candidate, $k$)
        candidate ← TwoOptImprove(candidate, dist)
        if cost(candidate) < cost(best_tour) then
            best_tour ← candidate
        end if
        optionally change the value of $k$
    until timeout
    return best_tour
end procedure
```

Tuning

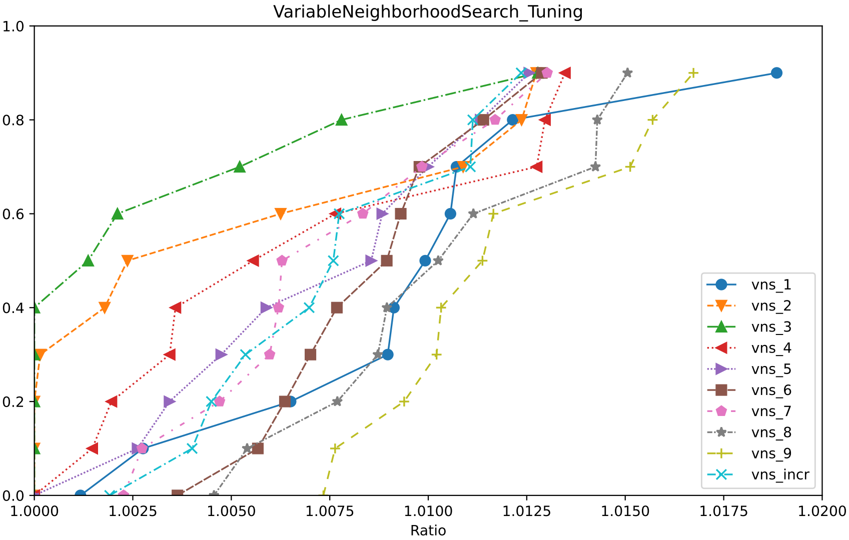

Figure 2: Parameter: k, with `incr` denoting the dynamic schedule approach.

Clearly k is a sweet spot for this instance size. We may expect the dynamic schedule work better work on more diverse settings.

3.6 Tabu Search

Tabu Search is a metaheuristic designed to overcome the limitations of local search algorithms like 2-opt, which often gets trapped in poor-quality local minima. Originally introduced by Fred Glover in 1986, it enables the exploration of larger regions of the solution space by allowing non-improving moves and systematically avoiding cycling.

The core idea is to perform the best available move at each step, even if it worsens the objective while maintaining a memory structure called the tabu list, which temporarily forbids the reversing of recent non-improving moves. This mechanism prevents immediate backtracking and promotes diversification.

Our implementation proceeds as follows:

1. Start from a feasible solution (e.g., after running Nearest Neighbor).
2. At each iteration, apply the best 2-opt move, even if the solution cost worsens.
3. If the move is non-improving, mark it as tabu by storing it in the tabu list.
4. Forbid any future move involving the same attributes until it is dropped from the tabu list.

To avoid cycling, we use a precise move encoding:

- Each 2-opt move is identified by the four vertices it affects: (a, b, c, d), representing edges (a, b) and (c, d).
- Once a move is tabu, any future move involving the same four vertices is blocked until it expires.
- A hash table is used for fast detection of whether a move is currently tabu.

Hyperparameters:

- **Move encoding:** We use a 4-vertex identifier per 2-opt move. Alternatives include storing edge indices or positions in the tour.
- **Tabu tenure:** We use a dynamic tabu list whose size oscillates sinusoidally to balance exploration and intensification phases.
    - Minimum tenure $T_{\min}$ (minimum list size)
    - Maximum tenure $T_{\max}$ (maximum list size)
    - Oscillation frequency

[20] [21]

Pseudocode

```
procedure TabuSearch(tour, dist, $T_{\min}$, $T_{\max}$ $f$)
    Input:
    tour: initial tour permutation of $0, \ldots, n-1$
    dist$[i, j]$: symmetric distance matrix
    $T_{\min}, T_{\max}, f$: parameters of the tenure size sinewave
    Output: best-found tour
    tabu_list ← {}
    best_tour ← tour
    repeat
        $(i, j, \Delta) \leftarrow$ BestNonTabu(tour, dist, tabu_list)
        apply the 2-opt swap at $(i, j)$ on tour
        if cost(tour) < cost(best_tour) then
            best_tour ← tour
        end if
        if $\Delta < 0$ then                                   ▷ Non-improving move
            add move $(i, j)$ to tabu_list with current tenure
        end if
        $T \leftarrow T_{\min} + (T_{\max} - T_{\min})(\frac{1}{2} + \frac{1}{2}\sin(\frac{2\pi}{f} \cdot \text{iteration\_count}))$   ▷ Current tenure size
        while $|\text{tabu\_list}| > T$ do
            expire oldest entry in tabu_list
        end while
    until timeout
    return best_tour
end procedure
```

Tuning

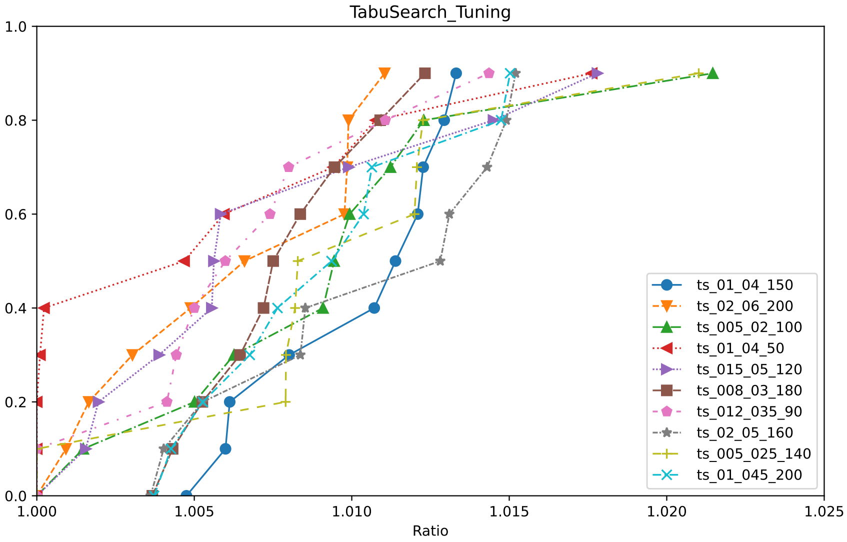

FIGURE 3: Parameters: (1) min. tenure, (2) max. tenure, (3) sinewave frequency.

Results are less clear cut compared to other tunings, with best competing algorithms differing by less then 1%.

3.7 Comparison of Heuristic Methods

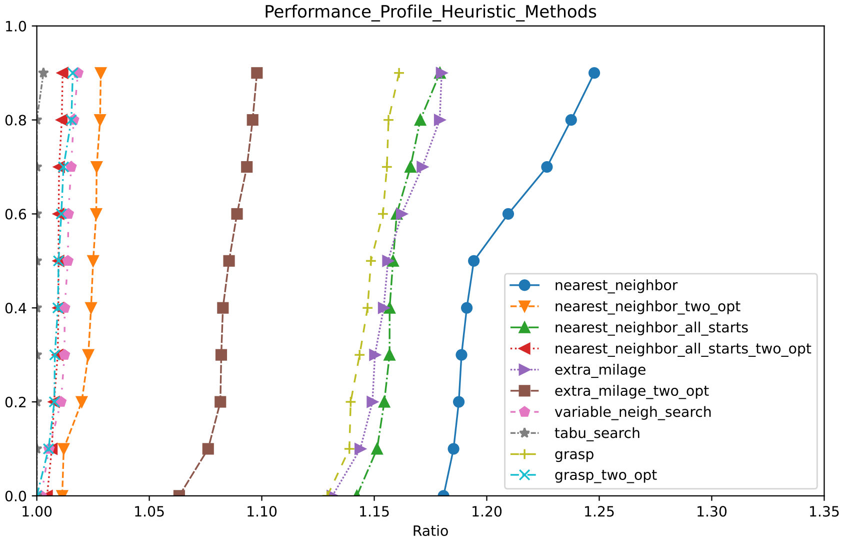

FIGURE 4: Performance profile comparison of all heuristic algorithms and their variants.

The best tuning of each algorithm was compared. Tabu search is clearly the best performing heuristic, as such it will be used to seed an initial solution for algorithms of the next chapters.

4 Global Search

The Traveling Salesman Problem is known to be NP-Hard, and solving it optimally and exactly involves a global search of the whole solution space. Techniques such as branch and cut allow pruning part of the search tree when it can be proved to not contain any optimal solution, making medium sized instances more tractable. Modern solvers such as Cplex provide fast, robust and numerically stable routines for many problems of the like.

4.1 Mathematical model

The mathematical formulation of the Traveling Salesman Problem (TSP) as an Integer Linear Program (ILP) dates back to the 1950s and was first formally studied by researchers at the RAND Corporation. The classical model is attributed to Dantzig, Fulkerson, and Johnson (1954), who introduced the use of subtour elimination constraints to model the TSP exactly. Over time, this formulation became a benchmark for testing exact optimization algorithms. With the development of commercial solvers like IBM ILOG Cplex, exact methods for solving the TSP using ILP have become more accessible. Cplex uses advanced techniques like branch and cut, cutting planes and presolving to solve such models efficiently for moderate-sized instances.

Cplex exposes several solver-level configuration switches that significantly impact performance:

- MIP Emphasis: prioritize either feasibility or optimality.
- Cut generation level: control how aggressively cuts are added to the model.
- Search strategy: traditional branch-and-bound or dynamic search.
- Parallelism: number of threads used during solving.
- Tolerance parameters: optimality gap tolerance, integrality tolerance, etc.
- Node and time limits: restrict the search space or runtime.

[2] [22]

Problem formulation

The global search approach to the TSP involves modeling the problem as a Mixed Integer Linear Program (MILP). The classical DFJ (Dantzig–Fulkerson–Johnson) formulation on a weighted undirected graph $G = (V, E)$ includes:

- Binary decision variables x_{ij} indicating whether the tour includes a direct trip from node i to node j,
- Objective function minimizing the total travel cost,
- Degree constraints ensuring each node is entered and exited exactly once,
- Subtour elimination constraints (SECs) to ensure a single connected tour.

We define binary decision variables $x_{ij} \in \{0, 1\}$, where $x_{ij} = 1$ means the salesman travels directly from node i to node j. The objective is to minimize the total cost or distance:

$$\min \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} c_{ij} x_{ij}$$

Degree constraints

They ensure that every node has exactly one incoming and one outgoing edge.

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} x_{ij} = 2 \quad \forall i \in \{1, \ldots, n\}$$

However, without SECs, this model allows for disconnected subtours like multiple smaller cycles that cover all nodes but don‘t form a single tour. As a result, the optimal solution to this relaxed model is often infeasible for the TSP.

Subtour elimination

SECs are added to the model to ensure that any subset of nodes cannot form a closed loop unless it includes all nodes, preventing subtours.

$$\sum_{i\in S}\sum_{\substack{j\in S\\ j\neq i}} x_{ij} \leq |S|-1 \quad \forall S\subset V,\ |S|>1$$

A complete model would require an exponential number of constraints, untractable to even enumerate. For this reason solving starts with just degree constraints, then employing different techniques to dynamically add SECs to the model as needed. This proves useful as usually only a reduced number of SECs (less then n) is needed to solve the problem.

[2] [22]

4.2 Bender's method

The Benders Loop approach for solving the Traveling Salesman Problem (TSP) is inspired by Bender's decomposition, originally introduced by Jacques F. Benders in 1962 to tackle large-scale mixed-integer programs by separating complicating constraints. While Benders decomposition was not initially designed for TSP, a similar iterative constraint generation philosophy has been successfully adapted to it, particularly for handling Subtour Elimination Constraints (SECs) efficiently.

The benders algorithm starts by solving the TSP model in Cplex without SECs, using degree-2 constraints and binary edge variables but no connectivity requirements. Once an integral solution is obtained, the set of connected components (subtours) is extracted by performing a depth-first search on the undirected graph defined by the selected edges. Each subtour S with $|S| < n$ violates the global tour constraint, so for each such S the SEC

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

is added explicitly to the model. The model is then re-solved in Cplex, and this process repeats: detect subtours, add their SECs, and solve again, until the solution forms a single connected tour.

Benders in practice usually converges in a small number od iterations, much less then n. However, we observe some drawbacks:

- Each model is solved from scratch, losing all progress of previous iterations.
- If we run out of time we have no feasible solution we can output to the user.

Implementation

We chose to add all violated SECs at each iteration. An alternative would be to add only the most violated ones or a fixed number per iteration. No removal or deactivation of SECs is ever performed: we always keep all SECs add through all past iterations.

```
procedure BendersLoop($dist$)
    Input:
    $dist$: distance matrix defining the problem
    Output: optimal tour as integer solution of MIP
    model ← InitModel                    ▷ binary variables and degree-2 constraints only
    repeat
        $x^* \leftarrow$ MipSolve(model)
        components ← FindCycles($x^*$)
        if |components| = 1 then
            return $x^*$                 ▷ feasible tour found
        end if
        for all $S \in$ components do
            add to model constraint $\sum_{i\in S}\sum_{j\in S,\ j\neq i} x_{ij} \leq |S|-1$
        end for
    until timeout
    return timeout failure
end procedure
```

[23] [24] [25]

4.2.1 Patching heuristic improvement

In the context of the Benders Loop, we often deal with incomplete TSP solutions composed of multiple disconnected components. These are not valid solutions to the original TSP and cannot be returned to the user. Although they offer a valid lower bound, they are not feasible. The Patching Heuristic addresses this issue by merging subtours into a single feasible tour using a greedy edge-swapping strategy.

This approach allows us to construct an approximate solution from any disconnected tour, making it possible to return a feasible solution even when the algorithm runs out of time.

The algorithm is iterative:

1. Scan all possible pairs of edges $(i_1, j_1) \in T_1$ and $(i_2, j_2) \in T_2$ on different subtours $T_1 \neq T_2$.
2. Evaluate the cost of swapping these edges, i.e. replacing them with (i_1, i_2) and (j_1, j_2) similarly to a 2-opt move.
3. Choose the pair of edges that minimizes the increase in cost. The swap merges components T_1 and T_2.
4. After each merge, the number of components is reduced by 1. Repeat until there is only one connected component remaining.

Each merging iteration requires checking all edge pairs residing in different components, resulting in a worst-case computational cost of $O(n^2)$ per iteration. The total number of iterations is $k - 1$, where k is the number of subtours present in the initial infeasible solution. Throughout the Benders loop we now have a lowerbound given by the cost of the relaxed problem, an upperbound given by the cost of the patched tour; the algorithm terminates when these two bounds cross. Improved results are obtained by running a 2-opt search on the resulting tour.

Pseudocode

```
procedure FindCycles($x^*$)
    Input: $x^*$ — binary edge values for a complete graph on $n$ nodes
    Output: arrays succ[0..$n-1$], comp[0..$n-1$], and component count $k$
    initialize succ[$i$] ← NIL and comp[$i$] ← 0 for all $i$
    $k \leftarrow 0$
    for $u = 0$ to $n - 1$ do
        if comp[$u$] $\neq 0$ then
            continue
        end if
        $k \leftarrow k + 1$, comp[$u$] ← $k$, start ← $u$, current ← $u$
        repeat
            choose any $v$ with comp[$v$] $= 0$ and $x^*$[current, $v$] $= 1$
            if such $v$ exists then
                succ[current] ← $v$
                comp[$v$] ← $k$
                current ← $v$
            else
                succ[current] ← start
            end if
        until succ[current] = start
    end for
    return (succ, comp, $k$)
end procedure
```

When considering edge swaps it is important to check both possible final configurations, as in 2-opt.

4.3 Branch and Cut

Branch and Cut was first applied to the TSP by Padberg and Rinaldi in the early 1990s, forming the foundation of the renowned Concorde TSP Solver. The Branch and Cut method is an advanced and more efficient alternative to the Benders Loop for solving the Traveling Salesman Problem (TSP) via integer programming.

One major drawback of the Benders Loop is that every time a set of violated Subtour Elimination Constraints (SECs) is added, the solver must restart the entire branch-and-bound process from scratch. This is computationally expensive, especially for large instances.

To avoid this inefficiency, Branch and Cut embeds the generation of SECs directly within the search process.

The model is initially defined without any SECs, just like in Benders. We install a callback function that is triggered each time Cplex finds an integer feasible solution (an incumbent).

This callback checks whether the incumbent forms a single connected tour. If the solution is valid (no subtours), it is accepted. Otherwise, the callback rejects the solution by providing the violated SEC to the solver. This prevents the same infeasible solution from being reconsidered and allows Cplex to continue the search efficiently without restarting.

This integration of subtour elimination within the branch-and-bound tree results in significantly faster convergence compared to Benders, as the tree is maintained throughout the solving process, and violated constraints are handled incrementally and locally.

While Branch and Cut methods can theoretically involve numerous hyperparameters, like cut selection rules, branching strategies, node selection policies). In our implementation we delegate these choices to Cplex's internal heuristics and strategies, using its default configuration. This is justified because Cplex's developers have already heavily optimized these settings across a wide range of problems, including the TSP.

The implementation is based con Cplex's C APIs: the callback is installed with `CPXcallbacksetfunc` on the context of `CPX_CALLBACKCONTEXT_CANDIDATE`. Our callback receives a `CPXCALLBACKCONTEXTptr` from which it can access the candidate incumbent with `CPXcallbackgetcandidatepoint`, eventually rejecting it by providing one or more SECs with `CPXcallbackrejectcandidate`.

[10] [5] [19]

4.3.1 Solution posting

An effective enhancement to our Branch and Cut implementation is the solution posting mechanism. The idea is that whenever Cplex returns an infeasible integer solution (i.e., one containing multiple subtours), we do not limit ourselves to rejecting the solution by injecting the corresponding SEC cuts. Instead, we also run the patching heuristic to quickly convert the infeasible solution into a valid TSP tour.

This patched tour, while heuristic, is provided back to Cplex as a candidate solution. Cplex recognizes it as non-optimal, but it may still use it to improve its current incumbent, that is, the best solution found so far. This is extremely valuable, especially in the early stages of optimization when good incumbents are rare.

Providing high-quality incumbents through posting allows Cplex to prune more nodes in the branch-and-bound tree (via bound comparisons), and it also facilitates variable fixing during presolve and restarts. The callback is only triggered a few hundred times during the whole search, so the overhead of computing and posting a patched solution is little, while the gain in convergence speed can be substantial.

4.3.2 Warm Start

The warm start is another effective strategy for accelerating the convergence of the Branch and Cut procedure. The goal is to provide Cplex with a good initial incumbent solution before the optimization process begins. To do this, we execute one of our internal constructive heuristics prior to launching the branch-and-cut tree. The solution obtained is then passed to Cplex as a starting incumbent.

This helps Cplex from the outset by giving it a tight upper bound, which in turn improves pruning efficiency and variable fixing during preprocessing and search. In our implementation, we use the Nearest Neighbor heuristic followed by 2-opt as the warm start routine. This choice is guided by two considerations: (1) it consistently produces a reasonable-quality solution, and (2) it runs in deterministic and bounded number of iterations, not subject to a time limit.

Although other heuristics like VNS or Tabu Search can produce better solutions, they come with unbounded runtimes and introduce difficulty in tuning their execution time. In contrast, Nearest Neighbor $O(n^2)$ followed by 2-opt (few iterations in practice) is fast, predictable, and generally sufficient for providing a valuable initial incumbent.

Empirically, we observed that either warm start or solution posting alone significantly improves Cplex's performance in Branch and Cut mode, and combining both offers diminishing returns. Thus, warm start serves as a clean, pre-optimization enhancement that leverages our internal heuristics without adding runtime uncertainty.

4.3.3 Fractional cuts separation

Another enhancement we implement in the Branch and Cut framework is the use of fractional cuts. Unlike traditional callback separation at integer incumbent solutions, here we also act on fractional solutions obtained after solving LP relaxations, especially at the root node. These cuts aim to improve the lower bounds in the branch-and-cut tree, making the solver more effective at pruning.

Technically, after each LP relaxation (most importantly at the root), we analyze the fractional solution and check whether it violates Subtour Elimination Constraints (SECs). If the solution already induces a disconnected residual graph, we immediately derive SECs from the components, similar to the integer case. However, in most cases, the residual graph is connected in the fractional solution, and we need to detect more subtle violations.

To do this, we run a maximum flow algorithm over the residual graph to detect minimum cuts that violate SECs. This is computationally expensive and delicate to implement. As suggested by our professor, instead of developing the separation routines from scratch, we integrated Concorde's open-source separation routines, which are highly optimized for this task.

Because this separation is expensive, we do not perform it at every node in the branch-and-bound tree. Instead, we introduce a **hyperparameter** $\theta \in [0, 1]$ that determines the probability of performing fractional cut separation at a given node:

- $\theta = 0$: no separation at fractional nodes.
- $\theta = 1$: always separate.
- $\theta = 0.5$: separate at half of the nodes (on average).

This trade-off allows us to balance the overhead of flow-based separation and additional simplex iterations against the benefits of tighter bounds. In our experiments, separating fractional cuts at the root node is always worth the effort and is always done regardless of θ.

It is important to stress that separating cuts at fractional solutions is optional, as these solutions are never returned to the user. By contrast, separation at integer solutions (as in classical Benders or SEC rejection) is mandatory, since skipping it would mean returning infeasible tours.

Tuning

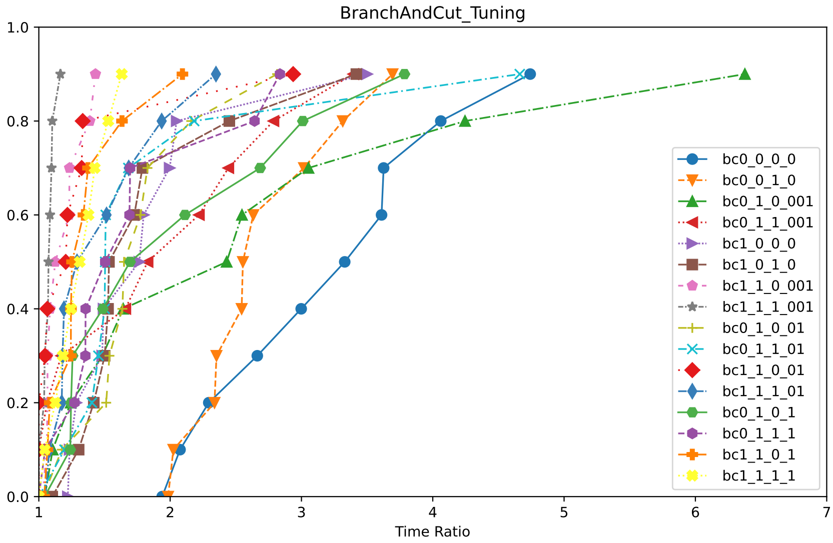

FIGURE 5: Parameters: (1) posting, (2) fractional cuts at the root, (3) warm start, (4) $\theta \in \{0.01, 0.1, 1\}$.

Tuning shows that proposed improvements are useful, and turning them all on yields the best results. Without them the base algorithm takes from 2 to 4 times longer. The best algorithm separates fractional cuts at internal nodes only once every 100 nodes.

4.4 Comparison of exact methods

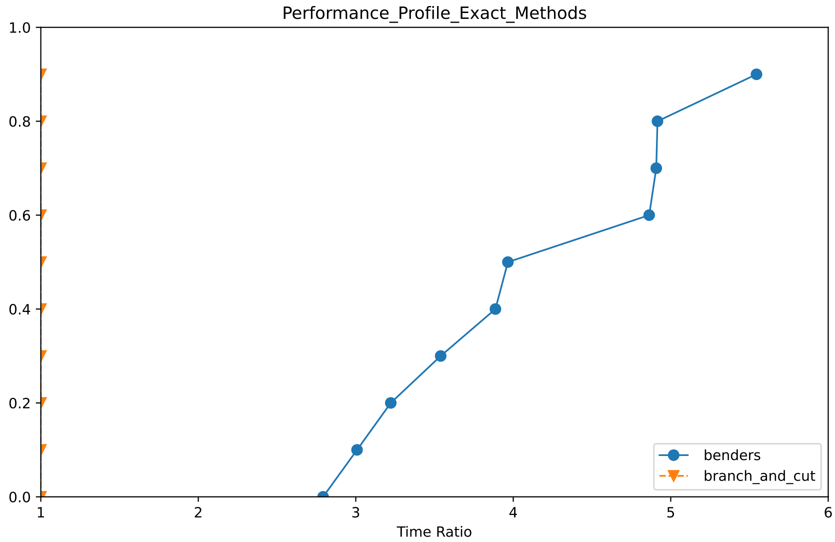

Figure 6: Comparing Benders with the best branch and cut.

Clearly branch and cut is the superior exact algorithm, with Benders taking up to 5 times as long to solve instances of 300 nodes.

5 Matheuristics

Matheuristics are hybrid approaches that integrate mathematical programming (typically MILP) with heuristic techniques to improve solution quality or performance. In the context of the Traveling Salesman Problem (TSP), matheuristics allow combining the power of exact solvers like Cplex with domain-specific heuristic strategies. We implemented two notable matheuristics: Hard Fixing and Soft Fixing.

Both are expensive refinement techniques, able to keep improving solutions even where VNS or Tabu would get stuck on a mediocre local minima. It is crucial to seed matheuristics with a good initial solution, since classical heuristics are much faster at finding easy improvements which would be wasteful to compute using mathematical solvers.

5.1 Hard Fixing

The technique of Hard Fixing in combinatorial optimization was popularized in the late 1990s and early 2000s, particularly by researchers like Michele Monaci and Marco Fischetti, who applied it successfully to large-scale integer programming problems, including the TSP and variants. It is often used as a practical compromise between exact methods and local search, blending heuristic construction with partial exact optimization.

Hard Fixing is a metaheuristic strategy used to simplify large and difficult integer programs by partially fixing decision variables, thus reducing the problem size and making each subproblem significantly easier to solve. In the context of the TSP, we start from a high-quality heuristic solution, obtained by running Tabu search for a fraction of the available time. We then fix a subset of its edges to 1, effectively constraining the search space. The rest of the variables remain free, and we run a branch and cut optimization to try to improve the current solution. After solving the reduced problem, we obtain a new complete tour, which may be better than the previous one. This process is iterated, each time fixing a new subset of edges, always starting from the best solution found so far.

The main hyperparameters of the algorithm govern the fraction of fixed edges:

- Initial fraction of fixed edges P_0: it determines the fraction of tour edges to be fixed in the first iteration. For instance, $P_0 = 0.8$ means 80% of the edges from the heuristic solution are fixed.
- Decay rate P_D: This controls how fast the fixing probability P decreases across iterations.

If during an iteration the fraction of fixed edges P is too big, we will search a small neighborhood and will fail to improve. Thus, if an iteration fails to improve the current best solution, we quickly reduce P. If an iteration succeeds, we slightly increase it: P may currently be smaller then needed, fixing more edges leads to faster iterations.

To select the subset of edges, we implemented three alternative strategies which we alternate across iterations:

1. Random fixing: Each edge is fixed independently with probability P, giving an expected $P \cdot n$ edges fixed. This is the simplest and most commonly used strategy in literature.
2. Segment fixing: A contiguous block of $P \cdot n$ edges in the tour are fixed.
3. Wall-centered fixing: A “free zone” is created around a randomly selected central edge, fixing the rest. This effectively focuses the optimization around a local region.

The stopping criteria for each fixing iteration are also hyperparameters. We use:

- Time-based limit: each fixing iteration is allowed to run for up to a percentage of the total runtime (e.g., 10%). Imposing a hard timeout is risky: if it turns out to be too low, no optimization will take place and the whole procedure will be pointless.
- Node-based limit: alternatively, the solver stops after exploring a fixed number of nodes (e.g., 1,000). We are more interested in improving the solution than proving optimality, and good solutions are often found at the beginning of optimization.

Because the solution obtained in each iteration is not guaranteed to be optimal (due to fixed edges) the focus is purely on iterative improvement, not exactness. We observed that hard fixing often improves the solution substantially within a fraction of the time required for a full branch and cut run on the full model.

[26] [27]

Pseudocode

```
procedure HardFixing(tour, dist, $P_0$, decay)
    Input:
      tour: current best tour permutation of $0, \ldots, n-1$
      dist$[i, j]$: symmetric distance matrix
      $P_0$: initial fraction of edges to fix
      decay: decay/adaptation rule for $P$
    Output: improved tour
    $P \leftarrow P_0$
    best_tour $\leftarrow$ tour
    repeat
        $\mathcal{F} \leftarrow$ SelectFixEdges(best_tour, $n$, $P$)                ▷ using one of the three strategies
        model $\leftarrow$ TspModelWithFixings(dist, $\mathcal{F}$)
        new_tour $\leftarrow$ SolveMIP(model, time_limit = fraction_of($T_{\max}$))        ▷ branch-and-cut
        if cost(new_tour) < cost(best_tour) then
            best_tour $\leftarrow$ new_tour
            $P \leftarrow$ increase($P$, decay)                                    ▷ grow on improvements
        else
            $P \leftarrow$ decrease($P$, decay)                                    ▷ decay on no-improve
        end if
    until timeout
    return best_tour
end procedure
```

Tuning

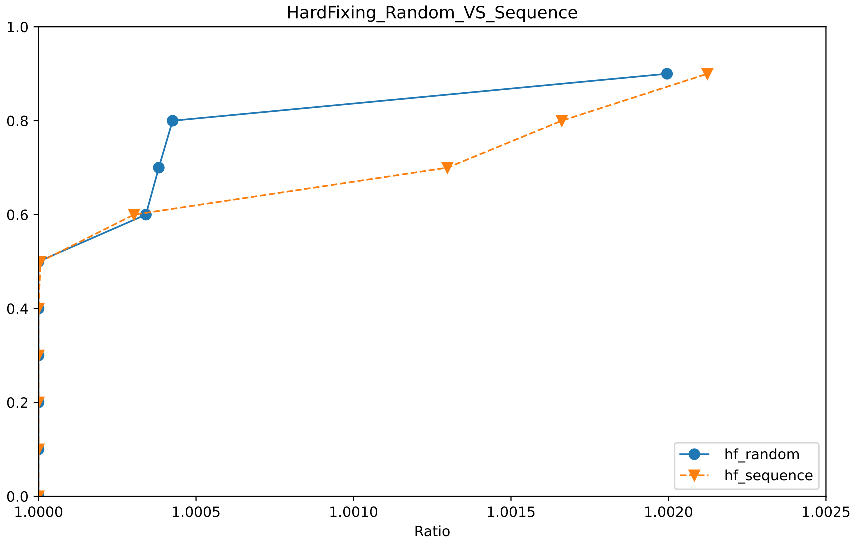

Figure 7: Comparing using only fixing strategy (1) and alternating it with sequence strategies (1+2+3).

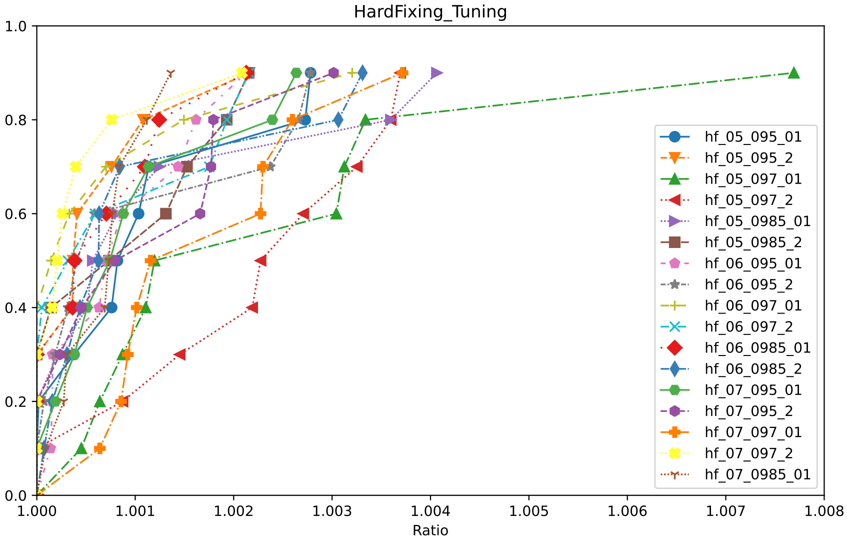

FIGURE 8: Parameters: (1) $P_0 \in \{.5, .6, .7\}$, (2) $P_D \in \{.95, .97, .985\}$, (3) node limit as a fraction of n, either 0.1 or 2.

Tuning results are not clear cut, with algorithms differing by less then by 0.4% on most cases. Figure 7 shows little to no difference between our creative fixing strategies and simple random fixing.

5.2 Soft Fixing

Local Branching, also known as soft fixing, was introduced by Fischetti and Lodi (2003) as a general technique for MIP-based neighborhood search. Its flexibility and solver compatibility have made it a key component in many hybrid metaheuristics and large-scale problem solvers.

The Local Branching technique provides a flexible way to explore the neighborhood of a high-quality solution without overly constraining the search. Unlike hard fixing, which rigidly locks in a subset of edges, local branching works by gently nudging the solver to stay ”close” to a given solution while still allowing controlled changes.

In our implementation, we begin by generating a strong incumbent solution using Tabu Search, which serves as the initial reference tour. We then pass this tour to CPLEX as a warm start. After that, we impose a local branching constraint: we require that any new solution found must share at least $n - k$ edges with the current best solution, meaning that at most k edges can differ. This constraint dynamically defines a local neighborhood around the incumbent, much larger than the $k = 2$ neighborhood of 2-opt.

Here, k is the main **hyperparameter**, representing the size of the neighborhood we are willing to explore. A smaller k means tighter control (more edges must match), while a larger k allows more exploration. We also impose a node limit and a time cap for the subproblem solved under this constraint, ensuring that each round of local optimization remains computationally tractable.

If the restricted problem yields an improved tour, we reset k to its initial value. Otherwise, we increase k: since the algorithm is deterministic, running another iteration with the same k would yield the same result again; we must instead explore a larger neighborhood. After each iteration, we drop the previous constraint and add a new one with the updated value of k, and the process repeats until the global time limit expires. Throughout the loop, we always retain the best feasible tour discovered so far.

[28]

Pseudocode

```
procedure SoftFixing(tour, dist, $k_0$, $\alpha$)
    Input:
      tour: incumbent tour permutation of $0, \ldots, n-1$
      dist$[i, j]$: symmetric distance matrix
      $k_0$: initial neighborhood size (max edges that may differ)
      $\alpha$: growth factor for $k$
    Output: best-found tour
    $k \leftarrow k_0$
    best_tour ← tour
    repeat
        model ← InitModel
        WarmStart(model, best_tour)
        AddRow(model, $\sum_{(i,j)\in \text{best\_tour}} x_{ij} \geq n - k$)        ▷ at most $k$ edges may differ
        candidate ← SolveMip(model)        ▷ branch-and-cut
        if cost(candidate) < cost(best_tour) then
            best_tour ← candidate
            $k \leftarrow k_0$        ▷ reset neighborhood
        else
            $k \leftarrow \min\big(k + \lfloor \alpha\, k_0 \rceil,\ n\big)$        ▷ expand neighborhood
        end if
    until timeout
    return best_tour
end procedure
```

Tuning

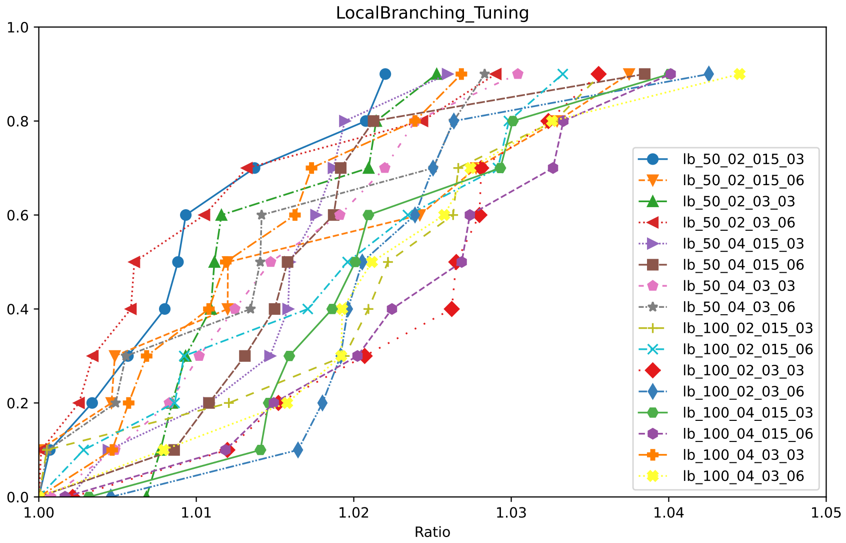

Figure 9: Parameters: (1) k_0, (2) k growth factor, (3) time limit fraction, (4) node limit as fraction of n

Tuning seems to favor small k values, both in term of k_0 and its growth.

5.3 Comparison of Matheuristics

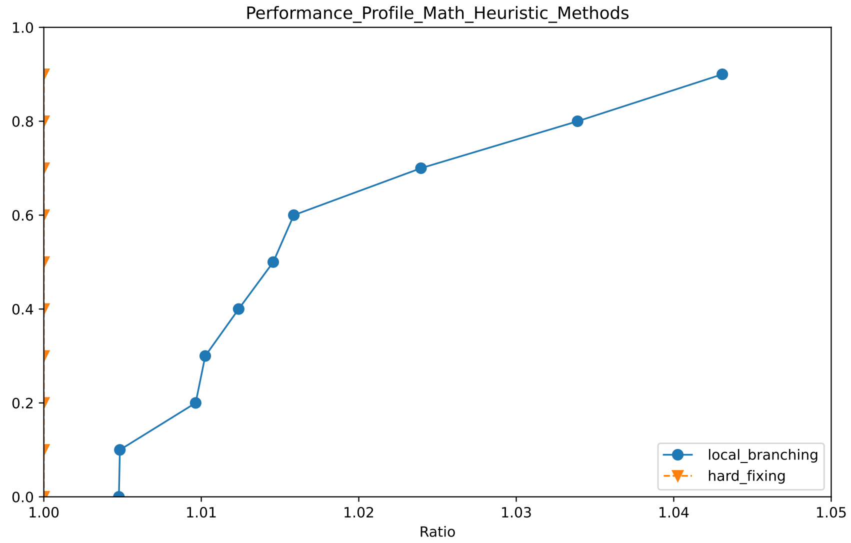

Figure 10: Comparison of hard vs soft fixing.

Clearly Hard Fixing outperforms Soft Fixing (local branching) in all instances, always producing better solutions, by around 2%.

References

[1] William J. Cook. In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation. Princeton University Press, 2011.

[2] George B. Dantzig, Ray Fulkerson, and Selmer M. Johnson. “Solution of a Large-Scale Traveling-Salesman Problem”. In: Operations Research 2.4 (1954), pp. 393–410. DOI: `10.1287/opre.2.4.393`.

[3] Merrill M. Flood. “The traveling-salesman problem”. In: Operations Research 4.1 (1956), pp. 61–75. DOI: `10.1287/opre.4.1.61`.

[4] David L. Applegate et al. The Traveling Salesman Problem: A Computational Study. Princeton University Press, 2007.

[5] Gregory Gutin and Abraham P. Punnen, eds. The Traveling Salesman Problem and Its Variations. Springer, 2002.

[6] Eugene L. Lawler et al. The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, 1985.

[7] William H. Press et al. Numerical Recipes in C: The Art of Scientific Computing. 2nd. Cambridge University Press, 1992, pp. 960–965.

[8] Thomas Williams and Colin Kelley. Gnuplot: An Interactive Plotting Program. Available at `https://gnuplot.sourceforge.net/documentation.html`. 1986.

[9] IBM. IBM ILOG Cplex Optimization Studio User’s Manual. Version 22.1. 2023.

[10] David L. Applegate et al. “Implementing the Dantzig–Fulkerson–Johnson Algorithm for Large Traveling Salesman Problems”. In: Mathematical Programming 97.1 (2003), pp. 91–153. DOI: `10.1007/s10107-003-0395-0`.

[11] Gerhard Reinelt. The Traveling Salesman: Computational Solutions for TSP Applications. Springer, 1994.

[12] William J. Cook. Concorde TSP Solver Documentation. Available at `https://www.math.uwaterloo.ca/tsp/concorde.html`. 2006.

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

[14] G. A. Croes. “A Method for Solving Traveling Salesman Problems”. In: Operations Research 6.6 (1958), pp. 791–812. DOI: `10.1287/opre.6.6.791`.

[15] G. Clarke and J. W. Wright. “Scheduling of Vehicles from a Central Depot to a Number of Delivery Points”. In: Operations Research 12.4 (1964), pp. 568–581. DOI: `10.1287/opre.12.4.568`.

[16] Ibrahim H. Osman and Gilbert Laporte. “Metaheuristics: A Bibliography”. In: Annals of Operations Research 63.5 (1996), pp. 513–628.

[17] Nenad Mladenović and Pierre Hansen. “Variable Neighborhood Search”. In: Computers & Operations Research 24.11 (1997), pp. 1097–1100. DOI: `10.1016/S0305-0548(97)00031-2`.

[18] Pierre Hansen, Nenad Mladenović, and José A. Moreno Pérez. “Variable Neighborhood Search: Methods and Applications”. In: Annals of Operations Research 175.1 (2010), pp. 367–407. DOI: `10.1007/s10479-009-0657-6`.

[19] Gregory Gutin and Abraham P. Punnen. The Traveling Salesman Problem and Its Variations. See section 7.2.1, pp. 263–265. Springer, 2002. ISBN: 978-1-4020-0586-7.

[20] Fred Glover. “Future Paths for Integer Programming and Links to Artificial Intelligence”. In: Computers & Operations 13.5 (1986), pp. 533–549. DOI: `10.1016/0305-0548(86)90048-1`.

[21] Fred Glover. “Tabu Search: A Tutorial”. In: Interfaces 20.4 (1990), pp. 74–94. DOI: `10.1287/inte.20.4.74`.

[22] George L. Nemhauser and Laurence A. Wolsey. Integer and Combinatorial Optimization. Wiley, 1988.

[23] Jacques F. Benders. “Partitioning Procedures for Solving Mixed-Variables Programming Problems”. In: Numerische Mathematik 4.1 (1962), pp. 238–252. DOI: `10.1007/BF01386316`.

[24] George L. Nemhauser and Laurence A. Wolsey. Integer and Combinatorial Optimization. Chapter 12, pp. 538–540. Wiley, 1988.

[25] Giovanni Codato and Matteo Fischetti. “Combinatorial Benders’ Cuts for Mixed-Integer Linear Programming”. In: Operations Research 54.4 (2006), pp. 756–766. DOI: `10.1287/opre.1060.0320`.

[26] Marco Fischetti and Andrea Lodi. “Hard Fixing for the Set Covering Problem”. In: 6th International Workshop on Inte
Springer, 2002, pp. 319–332. DOI: 10.1007/3-540-45578-7_22.

[27] Michele Monaci and Paolo Toth. “Partial enumeration and hard fixing: A computational study on the set covering problem”. In: Operations Research Proceedings 2001. Springer, 2002, pp. 370–375. DOI: 10.1007/978-3-662-12650-1_56.

[28] Marco Fischetti and Andrea Lodi. “Local branching”. In: Mathematical Programming 98.1 (2003), pp. 23–47. DOI: 10.1007/s10107-003-0382-0.