UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

University of Padua - Department of Information Engineering

Master Degree in Computer Engineering

A Comparative Study of Optimization Techniques for the Traveling Salesman Problem

Professor
Fischetti Matteo

Students
Felline Andrea - 2090597
Pietrobon Andrea - 2087639

ACCADEMIC YEAR 2023-2024

Operations Research 2

Contents

Chapter 1

Introduction

In the Field of combinatorial optimization, the Traveling Salesman Problem (TSP) stands out as a timeless and extensively researched challenge. Despite its deceptively simple formulation, finding a solution proves to be an extremely intricate task, earning it a reputation as one of the most notorious NP-complete problems [1]. The significance of TSP lies in its far-reaching practical implications, influencing fields such as logistics, transportation planning, robotics, and manufacturing, where efficient distribution and task sequencing are crucial. By cracking the TSP code, substantial gains can be made in terms of time, resources, and financial savings.
The purpose of this paper is to present, analyze and compare different approaches to solve the Traveling Salesman Problem [2] as a way to understand more deeply the various issues that arise when approaching those kind of problems.

1.1 Formulation of this thesis

In the next chapters, we are going to present all the work done, which includes mathematical formulations, implementation, and testing phases. In particular, this report is structured as follows:

Chapter 1: TSP History and formulation, providing a comprehensive overview of the Traveling Salesman Problem.

Chapter 2: Heuristics, detailing two heuristics approaches used to find approximate solutions to the TSP and discussing their efficiency and applicability.

Chapter 3: Metaheuristics, exploring advanced metaheuristic techniques and their effectiveness when applied to the TSP.

Chapter 4: Exact Models, presenting exact solution methods with a focus on their implementation and performance.

Chapter 5: Matheuristics, combining mathematical programming and heuristic methods to create hybrid approaches, and evaluating their performance on TSP instances.

Chapter 6: Conclusions, summarizing the findings, discussing the strengths and limitations of the different methods.

The code, this thesis, and further information can be found in the following GitHub repository: `https://github.com/Piero24/TSP_Optimization`

1.2 Problem history

The Traveling Salesman Problem (TSP) is a Combinatorial Optimization problem, succinctly posed as follows: ”What is the shortest route a traveling salesman can take to visit n cities and return back to his home city, only traversing each city once?”

The origins of the Traveling Salesman Problem (TSP) are shrouded in uncertainty, making it difficult to pinpoint its exact roots. However, one plausible theory dates the problem back to 1856-57 with William Hamilton. Hamilton created the ”Icosian Game,” which involved finding a path along the edges of a dodecahedron that visits every vertex exactly once, traverses no edge twice, and returns to the starting point. This concept, now recognized as a Hamiltonian Circuit, laid the groundwork for the Traveling Salesman Problem [3].

An alternative hypothesis suggests that the problem may have been formulated even earlier, potentially originating from a 1832 German handbook titled ”*The traveling salesman – how he should be and what he should do, to get the orders and assure success for his business – from an old traveling salesman.*”

The book features five routes that traverse regions of Germany and Switzerland. Notably, four of these routes involve revisiting an earlier city, which serves as a base for that leg of the journey. In contrast, the fifth route stands out as a genuine traveling salesman tour, as characterized in Alexander Schrijver’s comprehensive book on combinatorial optimization [4]. An illustration of the tour is given in Figure 1.1.
The first mathematical papers on the Traveling Salesman Problem (TSP) date back to 1940. Initially, researchers focused on finding a lower bound for the optimal tour. This interest was sparked by a practical problem encountered during an expedition in Bengal, where the transportation of men and materials between locations was a major expense.

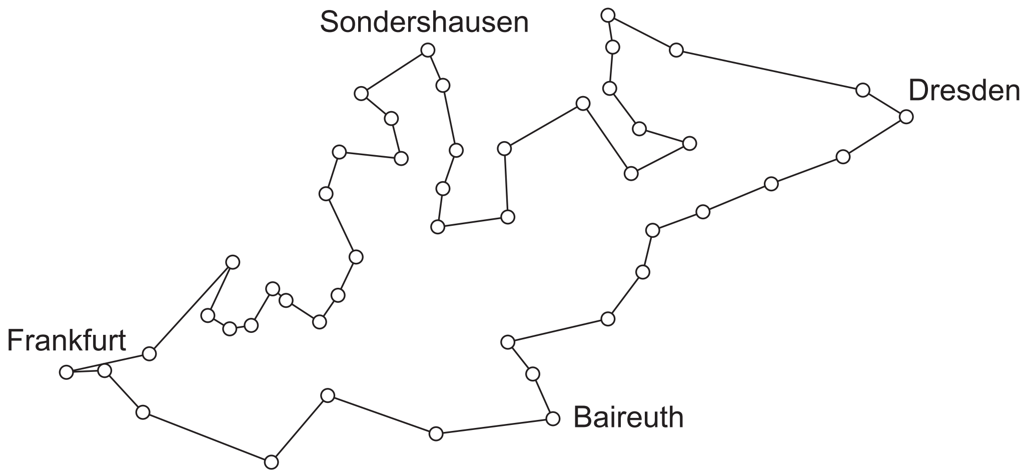

Figure 1.1: The Commis-Voyageur tour.

Mathematicians subsequently generalized the problem, first considering a finite number of random points within a unit-sided square, and later expanding to a general area. During this period, Ghosh observed that the problem proved to be extremely complex, except for very small instances, which, although solvable, held little practical significance.

The term ”Traveling Salesman Problem” (TSP) was first mentioned in a 1949 report by Julia Robinson, but it is likely that the name was already in use at Princeton in the 1930s or 1940s. In the 1940s, mathematicians at RAND Corporation, including Merrill Flood, began studying the problem in earnest. In 1954, a team of researchers at RAND, including Dantzig, Fulkerson, and Johnson, made a significant breakthrough by solving a 49-city problem in the USA using the simplex method.

Since then, improvements in solving the TSP have come mainly from developing better methods for applying the cutting plane method to linear programming problems. While there have been no major theoretical breakthroughs, researchers have made progress in understanding the TSP polytope and its properties, as well as developing heuristic algorithms to find near-optimal solutions. A notable theoretical result is the proof of the NP-hardness of the decision version of the TSP in 1972.

In the 1990s, Applegate, Bixby, Chvátal, and Cook developed the program Concorde that has been used in many recent record solutions. Gerhard Reinelt published the TSPLIB in 1991 [5], a collection of benchmark instances of varying difficulty, which has been used by many research groups for comparing results. In 2006, a team of researchers computed an optimal tour for an 85,900-city instance, currently the largest solved instance. For many other large instances, solutions can be found that are within 2-3% of an optimal tour [6].

Today, the research on the TSP is very active and its results have been successfully applied to other Operations Research problems. On one side, there is the search for better heuristics, both for speed and for optimality, using for example simulated annealing, genetic algorithms, tabu-search, neural networks or ant-colonies. On the other side, researchers have developed better and better software to solve the TSP to optimality, mainly through variants of the cutting plane method [7].

1.3 Problem formulation

The Traveling Salesman Problem (TSP) consists in finding a Hamiltonian circuit of minimum cost on a given directed graph $G = (V, E)$. In some cases, the problem can be analogously defined on a undirected graph; this happens when the cost associated with an arc does not depend on its orientation [8]. In the context of directed graphs, the goal is to find a path that traverses each arc exactly once while minimizing the total cost. Similarly, in undirected graphs, the problem is equivalent when the cost of traversing an edge is symmetric and independent of its direction.

In this document, we primarily address the Symmetric Traveling Salesman Problem (STSP), a fundamental challenge in combinatorial optimization. In STSP, we deal with an undirected weighted complete graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ represents a set of n nodes, and E comprises the set of n edges. Each edge $e = \{i, j\} \in E$ is associated with a non-negative real number, represented by the function $c : E \to \mathbb{R}^+$, denoting arbitrary distances or weights between nodes.

As mentioned before, the fundamental goal of the Traveling Salesman Problem (TSP) is to identify the most efficient sequence of edges or nodes that form a tour with the minimum total cost, also known as an optimal tour. To address this task, we need to formulate the problem as an integer linear program. Two notable formulations are the Miller–Tucker–Zemlin (MTZ) formulation and the Dantzig–Fulkerson–Johnson (DFJ) formulation. While the DFJ formulation is stronger, the MTZ formulation remains useful in specific contexts [6].

1.3.1 Dantzig–Fulkerson–Johnson (DFJ)

In this paper, we consider the Dantzig–Fulkerson–Johnson formulation. It is constructed as follows: each city is identified by a unique label from $1, \ldots, n$, and the cost (or distance) of traveling from city i to city j is represented by the positive value $c_e > 0$. The central variables in this model are $\{x_e\}_{i,j}$, where x_e represents the decision variable associated with traveling from city i to city j. They are constructed as follows:

$$x_e = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

The presence of $0/1$ variables in this formulation transforms them into integer programs, while all other constraints remain linear. Specifically, the objective of the program is to minimize the total tour length, which is represented by:

$$\sum_{e \in E} c_e x_e. \tag{1.2}$$

Without additional constraints, the variables $\{x_{ij}\}_{i,j}$ would essentially span all possible subsets of edges, which is far from the edges that form a tour. This would allow for a trivial minimum solution where all $x_{ij} = 0$. To avoid this, the formulation include the constraints that each vertex has exactly one incoming edge and one outgoing edge, which can be represented by $2n$ linear equations. The optimization problem is defined as follows:

$$\min \quad \sum_{e \in E} c_e x_e \tag{1.3}$$

$$\sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V \tag{1.4}$$

$$\sum_{e \in E(Q)} x_e \leq |Q| - 1 \quad \forall\, Q \subset V : |Q| \geq 2 \tag{1.5}$$

$$0 \leq x_e \leq 1 \text{ integer} \quad \forall e \in E \tag{1.6}$$

Constraints 1.4 ensure that each node in the graph is visited by exactly two edges of the cycle.

However, these constraints alone are not sufficient to guarantee a valid Hamiltonian Cycle, as the solution may consist of multiple isolated cycles. The final constraint in the DFJ formulation 1.5, also known as a Subtour Elimination Constraint (SEC), ensures that any solution found using this model consists of a single connected component. Specifically, this constraint prevents any proper subset Q from forming a sub-tour, ensuring that the solution is a single tour rather than a collection of smaller tours. Due to the exponential number of possible constraints, the problem can be very difficult to solve so this is typically solved using row generation in practice [6].

Chapter 2

Heuristics

As the number of nodes in a problem increases, the execution of exact algorithms becomes increasingly computationally demanding. Consequently, heuristic algorithms are employed, which do not guarantee an optimal solution but provide a satisfactory one with reasonable computational cost. We will introduce two families of heuristics:

- **Constructive heuristics**: Generate an approximate solution from scratch.
- **Refinement heuristics**: Improve an existing solution.

Constructive heuristics build a solution in a feasible amount of time, typically achieving within 15-20% of optimality. These heuristics are often used as a starting point for other heuristics, as the quality of the final solution heavily depends on the initial instance.

2.1 Nearest Neighbors

This constructive heuristic algorithm is a greedy 2-approximation algorithm, meaning that the solution is at most 100% away from the optimum. It generates a solution for the TSP instance in $O(n^2)$ time, starting from one node and selecting the next node in the tour that is closest to the current node at each iteration [9] [10]. The step of this algorithm are the following:

1. Select a random node.
2. Find the nearest unvisited node and go there.
3. If there are any unvisited nodes left, repeat step 2.
4. Return to the first node.

The pseudocode for computing a single starting solution from an arbitrary node is shown in Algorithm 1.

Algorithm 1: Nearest Neighbour

```
Input: Starting node
Output: A valid tour

path ← {};
current_node ← $starting\_node$;
visited ← {$starting\_node$};
solution ← 0;
while $|visited| \neq N$ do                    /* While there are unvisited nodes */
    closest_node ← nearest node to $current\_node \notin$ visited;
    solution ← $solution + dist(closest\_node, current\_node)$;
    current_node ← $closest\_node$;
    add $current\_node$ to $visited$;
    add ($current\_node$, $closest\_node$) edge to $path$;
end
add ($current\_node$, $starting\_node$) edge to $path$;
$solution \leftarrow solution + dist(starting\_node, current\_node)$;
```

Despite its simplicity and speed of implementation, this algorithm often find suboptimal solutions. However, these limitations can be mitigated using improvement techniques such as the 2-opt algorithm, which further enhances the quality of the obtained solution.

2.1.1 Nearest Neighbors from each node

An enhanced version of the algorithm can be developed by initiating the process from each possible node. This adjustment brings the complexity to $O(n^3)$, which is still significantly more efficient than the exponential complexity of exact algorithms. This modification mitigates the algorithm's sensitivity to initial conditions, thereby increasing its robustness. The pseudocode for this enhanced approach is as follows Algorithm 2:

Algorithm 2: Nearest Neighbour From Each Node

```
Input: TSP istance
Output: A valid tour
path ← $\{\}$;
foreach node n do
    current_path ← $NearestNeighbors(starting\_noden)$;
    if current_path better than path then
        path ← $current\_path$;
    end
end
```

2.2 Refinement Heuristics with 2-Opt

Once a tour has been generated by a construction heuristic, it can be improved using methods like the 2-opt local search. This kind of heuristic starts from a given solution and improves it by making small changes. Their performances are strongly dependent on the construction heuristic used. Other metaeuristics methods to enhance the solution include tabu search 3.1 and simulated annealing, both of which utilize the 2-opt move to find neighboring solutions.

This heuristic proposed in 1958 by G. A. Croes [11], first use some construction algorithm and then iteratively improves this tour by resolving crossing edges (2.1). This is done by selecting an edge (v_1, v_2) and searching for another edge (v_3, v_4),(the edge v_i belong to the tour) completing a move only if:

$$\text{dist}(v_1, v_2) + \text{dist}(v_3, v_4) > \text{dist}(v_2, v_3) + \text{dist}(v_1, v_4)$$

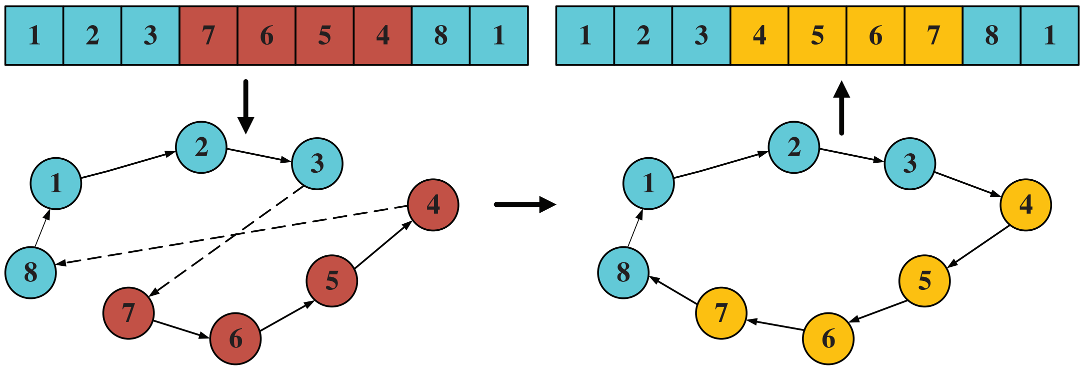

Figure 2.1: Illustration of the process for removing an intersection in a path.

This ensures a valid tour, as shown in Figure 2.2. The process continues until no more 2-opt improvements can be found, resulting in a 2-optimal tour.

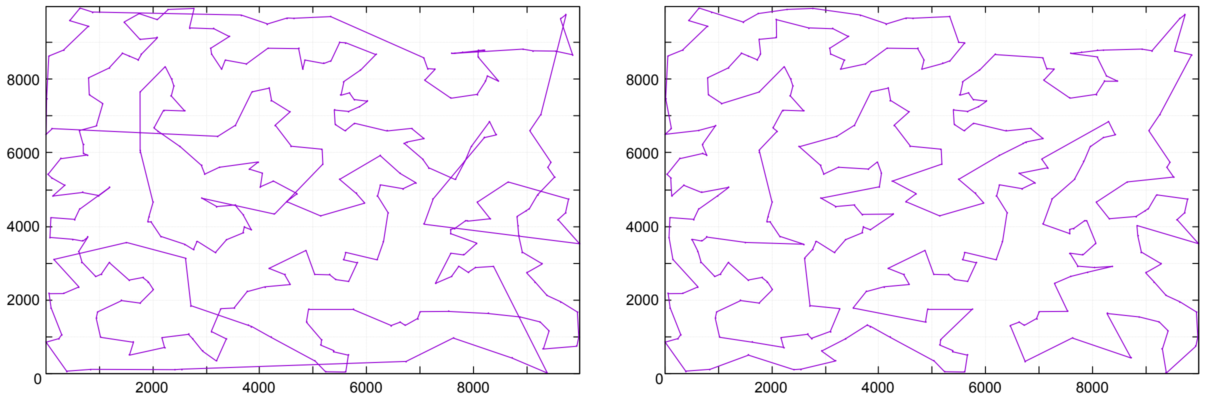

Figure 2.2: 2-opt before (left) and after (right)

The 2-OPT algorithm is shown in Algorithm 3.

```
Algorithm 3: 2-Opt
  Input: A valid tour
  Output: An optimized valid tour
  counter ←0;
  while counter < #nodes and time < time_limit do
      counter ←$counter + 1$;
      A ←$tour[0]$;
      A1 ←$tour[1]$;
      for B from $tour[2]$ to $|tour| - 2$ do
          B1 ←$B + 1$;
          $\Delta(A,B) \leftarrow (c_{A,B} + c_{A1,B1}) - (c_{A,A1} + c_{B,B1})$;
          if $\Delta(A,B) < 0$ then
              /* Reverse subvector from A1 to B                    */
              reverse subvector $tour[A1 : B]$
              tour.cost ←$tour.cost$ + $\Delta(A,B)$
              if current solution better than official solution then
                  update official solution;
              end
          end
      end
      move A to the end of the array;
  end
```

2.3 Comparison between Heuristics

As shown in Figure 2.2, the result can vary significantly in terms of route and therefore cost, as all possible intersections are eliminated thanks to the 2-opt method, as previously mentioned. However, this may in turn lead to an increase in processing time to remove the intersections and find a better route, in particular with large input files.

Figure 2.3 illustrates the performance profile with the cost comparison between the Nearest Neighbor and the Nearest Neighbor with 2-opt. We can see that the 2-opt gives better result, with an error less the 10% in every class, while the Nearest Neighbor alone reaches a 20% in most cases.

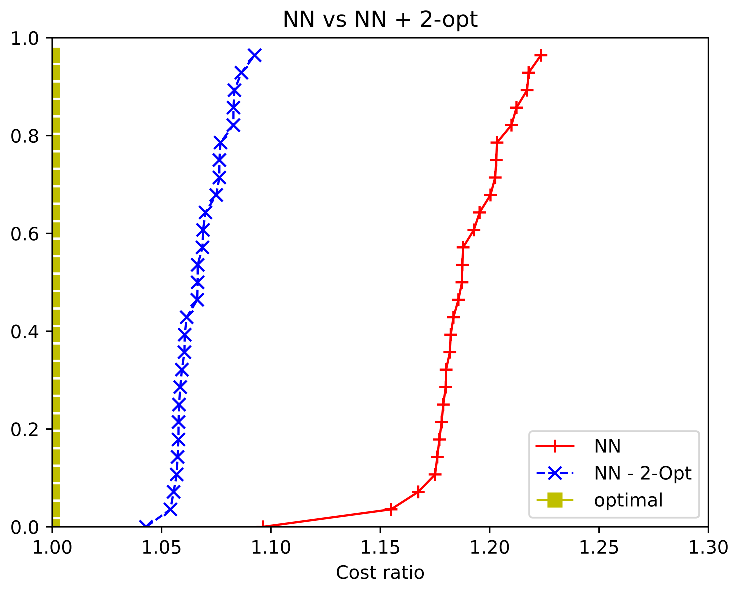

Figure 2.3: Performance Profile of Nearest Neighbor and Nearest Neighbor with 2-opt.

Chapter 3

Metaheuristics

Metaheuristics are high-level, problem-independent algorithmic that perturb the solution by moving it away from the current local optimum and attempting to get as close as possible to the global optimum. These algorithms can address various types of optimization problems with minimal adaptation. Even with a basic adaptation to a specific problem, they can often yield good solutions for certain instances. Specifically for the TSP, metaheuristics can typically find better solutions than any 2-opt variation within a reasonable time limit. These techniques essentially allow for the exploration of the solution space, avoiding stagnation in local minima or maxima with objective function values that are far from the global optimum.

In the following sections, we will present two different metaheuristic algorithms:

- **Tabu Search**: When a local minimum is reached adds certain edges to a tabu list, making them forbidden for an improving move. And force the exploration of new situations.
- **Variable Neighborhood Search (VNS)**: Randomly alters k different edges from the current solution and improves it using 2-opt.

3.1 Tabu Search

Designed by Fred W. Glover [12], the Tabu Search algorithm allows for modifying a given local optimum solution, even at the cost of worsening it, with the goal of exploring the solution space more thoroughly. At each iteration of the algorithm, the solution is modified with a new tour belonging to its 2-opt neighborhood.

To prevent these modifications from leading to the exploration of already visited local minima, a list of ”forbidden moves” is created, called $Tabu\ List$, which is a FIFO (First In First Out) queue with a specific size. The size of the tabu list is referred to as $tenure$. This way, a move won’t remain illegal forever but just for a certain number of iterations.

The performance of this algorithm can depend on the tenure of the list. If it is too small, the algorithm gets stuck in the local minimum because there is a sequence of moves that brings the solution back to the local minimum. Conversely, if the tenure is too large, the search is not effective because the neighborhood is too small.To overcome this, there are some options to dynamically adjust the tenure during execution.

There are many different policies to change the Tabu tenure dynamically. The policies considered in this case were:

- **Step policy**: Changes the Tabu List size every K iterations to a minimum or a maximum.
- **Linear policy**: The Tabu List size grows by 1 unit for each iteration until reaching the maximum size, then decreases by 1 unit for each iteration until reaching the minimum size, and so on.
- **Random policy**: Changes the Tabu List size every K iterations to a random size chosen within a range.
- **Reactive policy**: The maximum tenure is equal to $1/K$ (with K arbitrary) of the number of nodes of the instance to be solved, while the minimum is equal to half of this value.
- **Static policy**: The value of the tenure does not change during the execution.

The variant implemented in the code is the $Static\ Policy$, that is the simplest one. A dynamic policy may create better results in theory, but it also requires more computation time. In average, with our input sizes, it should make almost no difference, so we went for the simplest solution.

When, during the execution, the method encounters an edge that is tabu, it skips that edge. This ensures that other solutions are explored rather than reverting to the previous ones. When the best non-tabu couple of edges is found, the algorithm does a 2-opt step. It may happen that the best non-tabu swap is a worsening 2-opt move, it would mean that the algorithm reached a local minimum and it is trying to escape from it. When the tabu list is full and the method needs to add a new node, the oldest one is removed to make space. The termination criterion for the algorithm can be defined by either the expiration of the available time or the achievement of a maximum number of iterations. In our code, we used the time-based criterion. The solution provided by the algorithm is the best one identified up to the point of termination.

The Tabu Search pseudocode is shown in Algorithm 4.

Algorithm 4: Tabu Search

```
Input: TSP Istance with a feasible solution
Output: A valid tour

result ← best solution found so far;
cost ← cost of result;
tenure ← $\max(\text{inst.nnodes}/10, 10)$;

while time elapsed < time_limit do
    foreach node pair (A, B) do
        if A or B in TabuList then Skip;

        compute swap costs;
        if fist iteration or new cost < bestCost then
            bestSol ← updated solution;
            bestCost ← new cost;
            swappedNode ← node A;
        end
    end
    if bestCost ≥ cost then
        add swappedNode to tabu list;
    end
    result ← bestSol;
    cost ← bestCost;
end
```

3.2 Variable Neighborhood Search (VNS)

The Variable Neighborhood Search (VNS) algorithm, introduced by N. Mladenović and P. Hansen [13], aims to enhance a given local optimum by starting with an arbitrary solution and exploring neighborhoods of different sizes, as illustrated in Figure 3.1. The idea of the VNS algorithm is that Tabu Search waste too much time to escape from a local minimum.

That time should instead be used for optimization. That is why it uses a quick way of escaping: the kick. This method is also a less controllable way that does not assure good results since it may fall back to the same local minimum. VNS is founded on three key principles:

- A local minimum in one neighborhood structure may not be the same in another.
- A global minimum is a local minimum in all neighborhood structures.
- Often, a local minimum in one neighborhood structure is near a local minimum in another.

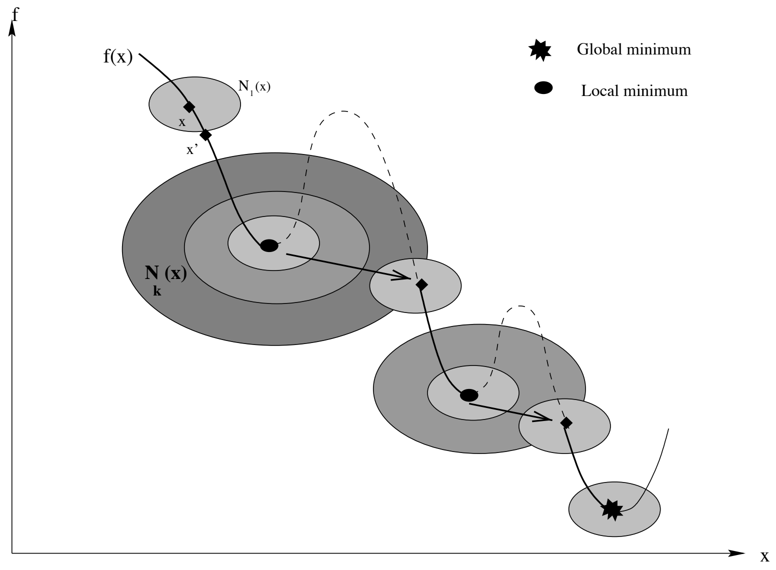

Figure 3.1: An illustration of the Variable Neighborhood Search (VNS) process [14].

If a superior solution is not discovered in the neighborhoods of size K, the algorithm randomly selects and replaces certain branches with other edges [15]. This introduces a temporary increase in cost, with the expectation that a new neighborhood will lead to a solution that diverges from the initial local optimum.

VNS repeatedly refines the current solution until a local minimum is achieved (intensification phase), using the 2-opt method. Subsequently, the algorithm picks a random solution within the neighborhood (diversification phase) as shown in the Figure 3.2. If this new solution is better than the best one found so far, it becomes the new best solution. In the next iteration, the algorithm starts with the smallest neighborhood; if no improvement is found, it moves to a larger neighborhood.

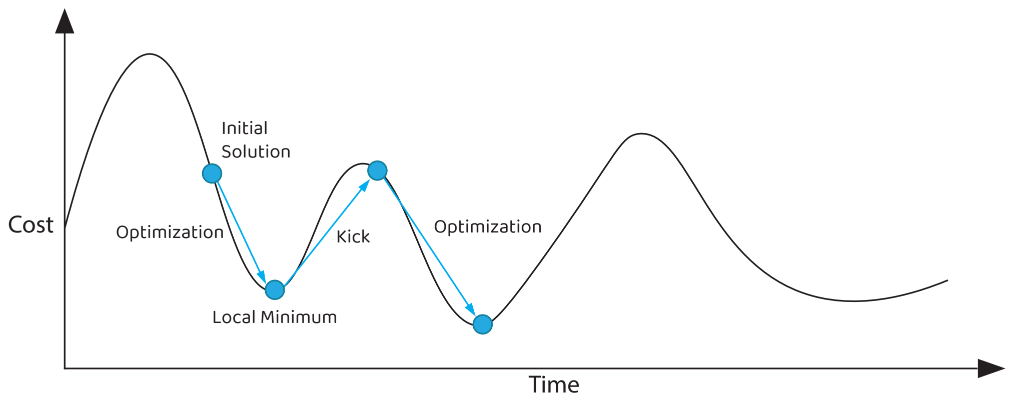

Figure 3.2: An illustration of the Variable Neighborhood Search (VNS) process.

The algorithm 5 concludes when a user-defined time limit is reached or after a set number of iterations, returning the best solution found. This method retains much of the initial solution, preventing the loss of previously gathered information.

Algorithm 5: Variable Neighborhood Search

```
Input: TSP instance
Output: A valid tour
Initialize result vector with best solution found so far;
while $time < time\_limit$ do
    performTwoOptOptimization();
    for $i \leftarrow 1$ to random number between 2 and 10 do
        applyKick();
    end
end
```

The implementation presented here considers only the 3-opt neighborhood. This means that during the kick operation 6, three edges are randomly removed and reconnected in a predefined manner.

Algorithm 6: Kick Function *applyKick()*

```
Input: Solution vector, cost
Output: Updated solution vector and cost
Randomly select indices $i$, $j$, $k$;
Update the solution vector by swapping sub-sections between $i$ and $j$ and between $j$
  and $k$;
Adjust the cost based on the new configuration;
```

3.3 Comparison between Metaheuristics

We have seen two important metaheuristic algorithms applied to the TSP. Here we compare the two algorithms in terms of costs of the solution found.

Figure 3.3 illustrates the cost comparison between our implementation of the Tabu Search algorithm and the Variable Neighborhood Search (VNS) algorithm. Both were given a time limit of 60 seconds.

As we can see they gave mostly the same results. The ability of VNS to outperform Tabu Search in some cases suggests that it can offer advantages in certain situations, though not consistently. Similarly, the inferior performance of VNS in other cases indicates that Tabu Search maintains a competitive robustness. Overall, this analysis confirms the qualitative equivalence of the two implementations in our application context.

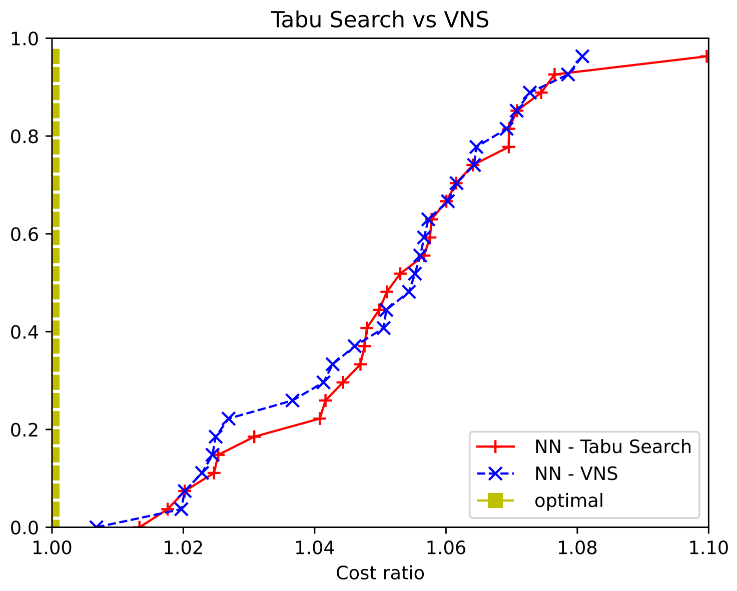

Figure 3.3: Performance Profile of Tabu Search and VNS.

Chapter 4

Exact Models

The previous sections illustrated some heuristic methods. The main property of those methods is their speed, but they provide approximate solutions. In contrast, to obtain the best possible solution, we must use exact methods that optimize a mathematical model of the problem and find the optimal solution.

This section introduces methods that solve a MIP problem to optimality using the basic CPLEX [16] MIP solver. The MIP problem model used is described in the introduction section 1.3.1.

The main challenge was adding the sub-tour elimination constraints (SEC) 1.5. The initial approach was to define and build a model without these constraints and directly use the CPLEX solver on it. Obviously, this approach yields an incorrect solution composed of various separated components instead of a unique path that visits all points.

To address this problem, the first method that we propose is Benders' Loop, which iteratively adds the SEC and solves the problem until the optimal solution without subtours is found. The second method uses a CPLEX callback function that is called every time a new solution is found. This allows us to reject solutions with subtours by adding the corresponding SECs or to accept the solution if it has only one component. We will also cover some other methods that can speed up the process, such as Posting heuristic solutions, MIP start and User-cut callbacks.

4.1 Benders' Loop method

The Dantzig-Fulkerson-Johnson (DFJ) 1.3.1 model incorporates subtour elimination constraints to the problem, but the number of such constraints is exponential.

The first method we introduce to address this issue is known as "Benders' Loop". This method is based on the idea that the majority of potential subtours consist of very unfavorable edge combinations, which correspond to parts of the solution space that the MIP solver would quickly discard while minimizing the solution cost. Therefore, the method focuses only on the subtours selected by the solver, prohibiting their selection.

Benders' implementation iteratively solves the DFJ model. It begins with the degree constraints and solves the problem using CPLEX's MIP solver. Upon finding a solution, it checks for the presence of subtours. If subtours are detected, the method adds subtour elimination constraints (SECs) for each subtour and resolves the modified model until a solution is found or the time limit is reached. Specifically, if the solution contains m subtours and S_k is a subtour, the Loop method adds the following constraints:

$$\sum_{e \in E(S_k)} x_e \leq |S_k| - 1 \quad \text{for } k = 1, \ldots, m$$

The pseudo-code of Benders' loop method is shown in Algorithm 7.

Algorithm 7: Benders implementation of the DFJ model

```
Input: TSP instance
Output: A valid tour (CPLEX solution)

model ←$naive\_model$;
solution ←$solve$ model;
while solution has subtours do
    foreach subtour in solution do
        sec_constraints ←generate SECs constraints of subtour;
        add sec_constraints to model;
    end
    solution ←$solve$ model
end
```

This procedure avoids generating an exponential number of constraints. However, the major drawback is the need to rebuild and reoptimize the model from scratch at each iteration.

4.1.1 Patching Heuristic for Benders: Gluing

The Bender's Loop method, in every iteration, only creates new SECs and then discards all previous work to start again from scratch. Instead, it would be beneficial to save as much progress as possible to make the method more efficient. A possible solution is to attempt to fix the solution provided by CPLEX by gluing together the subtours, resulting in a single comprehensive path.

The output of this technique will not have any guaranteed optimality, but it could still be valuable. In case the method is stopped before completion, we would have a feasible solution. Additionally, it can provide a good upper bound, as we know that the optimal solution will have

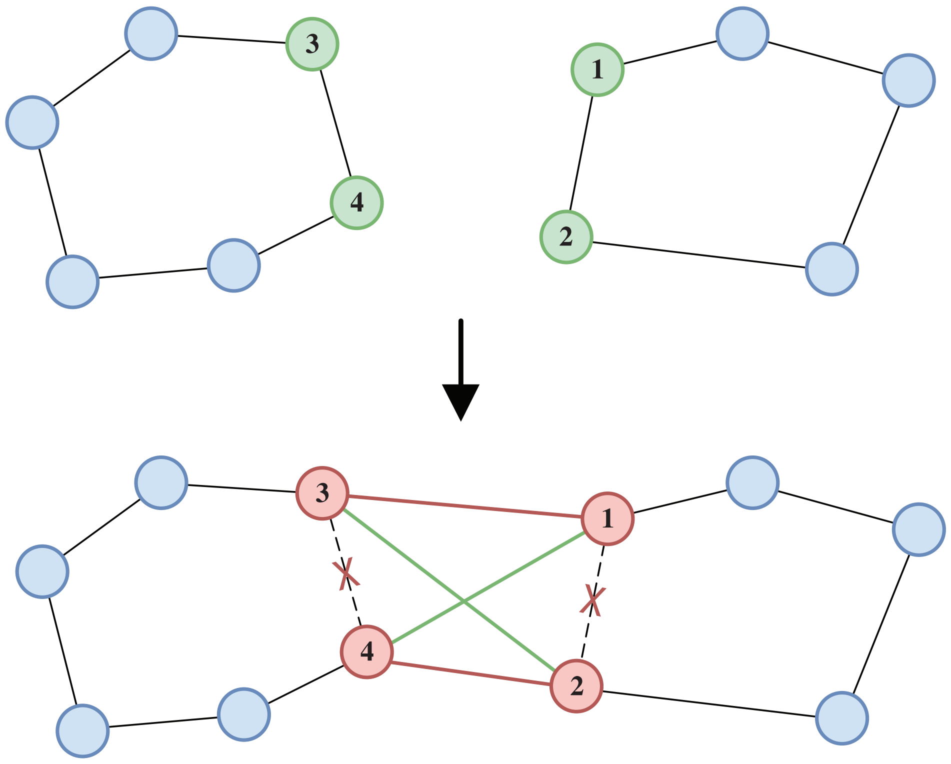

Figure 4.1: Gluing of two components.

a cost better than or equal to the glued solution. Moreover, it offers a good starting point for the next iteration, as the glued solution can be provided to CPLEX as a MIP start.

The action of gluing together two subtours is made by choosing the best couple of edges of different components that generate the lowest difference of cost when swapped (as shown in Figure 4.1). This operation is repeated until there is only one component.

The pseudo-code of the gluing operation is shown in Algorithm 8. It takes as input the multi-component solution in a particular format:

- `ncomps`: number of components
- `comp`: array with ids of components for each node
- `succ`: array with successive node of each node
- `cost`: cost of the current solution

Algorithm 8: Gluing

```
Input: ncomps, comp, succ, cost
Output: The glued solution
while ncomp > 1 do
    edge1, edge2, costDifference ← find best (with minimum cost difference) 2 edges
      to swap such that they are from different components;
    swap succ[edge1.first] with succ[edge2.first];
    update value of comp to values in range [1, ncomp-1];
    cost ← cost + costDifference;
    ncomp ← ncomp - 1;
end
```

4.2 Branch and Cut in CPLEX with candidate callback

An alternative method to incorporate SEC constraints is through the branch-and-cut technique, specifically utilizing CPLEX to address the problem.

At the root node, CPLEX's branch-and-cut algorithm performs several preprocessing steps. For each node in the branching tree (Figure 4.2), it employs multiple cut separation families, such as Gomory, Clique, and 0-1/2 cuts. Following the calculation of the relaxation, primal heuristics are applied to identify increasingly optimal incumbent solutions. These heuristics take the fractional solution as input, convert it into an integer solution, and if the resulting cost is lower than the current incumbent solution, the incumbent is updated. This integer solution may still contain subtours.

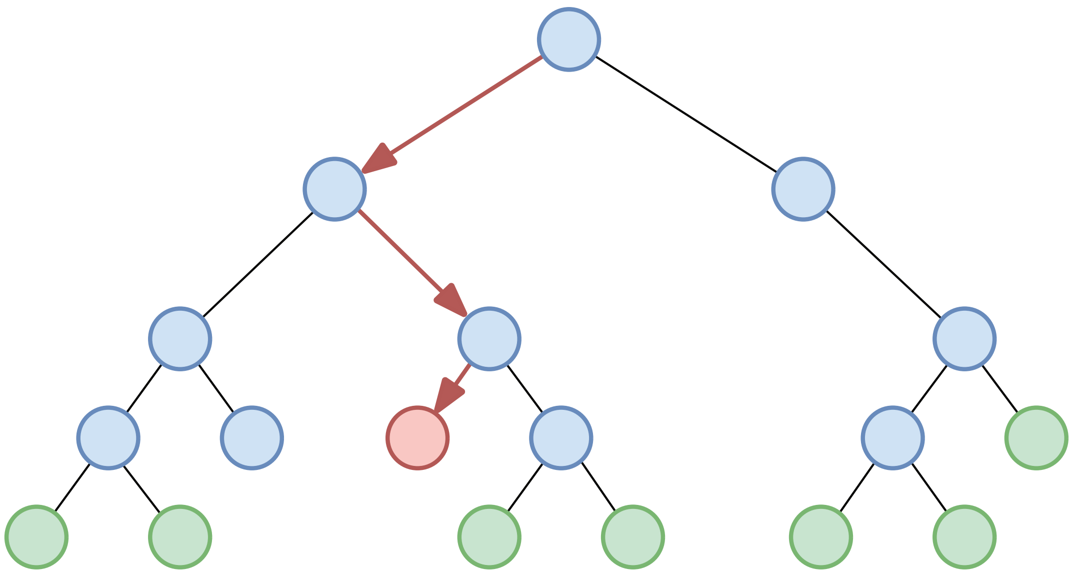

Figure 4.2: Branch & Cut tree visualization.

Between the heuristic application and the incumbent update, it is possible to instruct CPLEX to invoke a custom callback function. Exploiting this CPLEX feature we can inspect the solution and act based on its properties. If it has sub-tours the function adds the SECs and discards the solution, while if it has only one component the function accepts it.

This method let use fully use the potential of CPLEX without needing to restart the optimization multiple times.

The pseudo-code of the candidate callback is shown in Algorithm 9.

Algorithm 9: Candidate callback

```
Input: Candidate solution
if solution has subtours then
    foreach subtour in solution do
        sec_constraints ← generate SECs constraints of subtour;
        add sec_constraints to model;
        discard solution;
    end
end
else
    accept solution;
end
```

4.2.1 Posting Heuristic solutions

The incumbent callback allows for the implementation of custom heuristics to help CPLEX quickly identify an improved solution.

Starting with the integer solution provided by CPLEX in this callback, the gluing algorithm from the Bender's loop method can be applied, followed by a 2-opt optimization.

The improved solution can then be *posted* in the CPLEX environment, making it available for evaluation. CPLEX will assess this solution and determine whether to adopt or discard it.

4.2.2 MIP start

In addition to the aforementioned techniques, it is also possible to provide a starting solution to CPLEX, known as a MIP start, which prevents the creation of poor random solutions at the beginning of the Branch-and-Cut (B&C) method. By doing so, the CPLEX solver can begin its optimization process with a reasonably good feasible solution, working to improve it towards optimality rather than starting from a random solution, which may be much more challenging to optimize.

To generate the MIP start solution, any of the previously discussed heuristic or metaeuristic methods can be employed. In our case, we decided to use a simple nearest neighbor heuristic starting from a random node, combined with a 2-opt optimization: a suitable balance between a fast and robust approach.

4.3 SECs for Fractional solutions: user-cut callback

In this section, we describe an advanced use of CPLEX's callback functionality. At each node of the branching tree, CPLEX allows the invocation of a user-cut callback, a custom function applied to the relaxed solution of the problem. The relaxed solution does not account for the integer constraints of the variables. This feature facilitates the creation of custom cuts for the relaxed problem using callbacks. However, in the context of the TSP, the solution must be integer. Consequently, the methods used in previous sections to count the number of components or to merge them are not applicable.

To address this, functions from the Concorde library are utilized [17]. For example, the function `CCcut_connect_components` counts the components in a fractional solution, and `CCcut_violated_cuts` calculates the min-cut of a flow problem. The implemented callback functions, similarly to the candidate callback, use this library to compute the number of components in the relaxed solution returned by CPLEX and, for each component, an SEC is applied as a global constraint.

The distinction from the candidate callback arises when the number of components found is one. In the candidate callback, a solution with one component is considered feasible. However, in the user-cut callback, the min-cut on the graph needs to be calculated. For instance, in TSP, for each cut $(S, V \setminus S)$, the constraint

$$\sum_{(i,j)\in\delta(S)} x^*_{ij} \geq 2 \quad \forall S \subseteq V, S \neq \emptyset \tag{4.1}$$

must be satisfied, where $i \in S$, $j \in V \setminus S$, $x^*_{ij} \geq 0$ is the value of the edge (i, j) in the relaxed solution, and $\delta(S)$ is the cut-set of S. The Concorde function `CCcut_violated_cuts` identifies the cuts violating this constraint, and SECs are applied to each cut.

Unfortunately, applying this callback at every node of the branching tree is impractical due to the high time complexity of Concorde's algorithms. Thus, cuts are applied with a probability of 10%. An alternative approach could involve applying cuts when the node depth in the branching tree is below a certain threshold (e.g., 5).

It is important to note that User-Cuts callbacks are a subroutine of the candidate callback method. The candidate callback is triggered when an integer solution is found, while user cuts are called for fractional solutions. This helps CPLEX enforce critical constraints before updating the incumbent solution, potentially reducing the branching tree size and overall computation time.

The pseudo-code of the candidate callback is shown in Algorithm 10.

Algorithm 10: user-cut callback

```
Input: Fractional solution
with probability 90%: return;
call CCcut_connect_components on solution;
if number of components $\geq$1 then
    foreach subtour in solution do
        sec_constraints $\leftarrow$ generate SECs constraints of subtour;
        add sec_constraints to model;
        discard solution;
    end
end
else
    call CCcut_violated_cuts on solution;
end
```

4.4 Comparison between Exact Models

Figure 4.3 illustrates the performance profile with the time comparison between our implementation of the base Benders' loop method and Benders' loop method with gluing.

The two algorithms have similar performances, with some exception where the version with gluing is much slower. This probably happens with the biggest inputs, when the operation of gluing begins to be too heavy. In the 80% of the cases however they took the same time to find the optimal solution, and the gluing feature lets the user have a partial solution even if the algorithm is stopped before finishing.

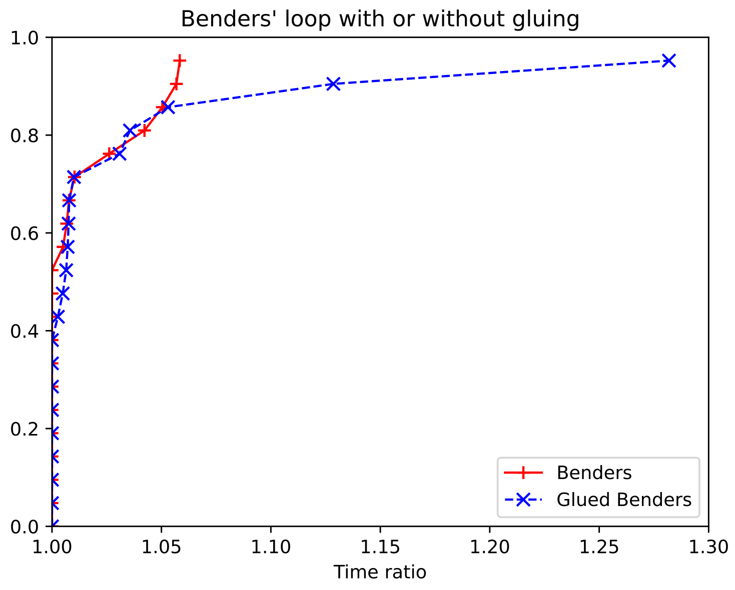

Figure 4.3: Performance profile of Benders' loop methods.

The graph in Figure 4.3 compares the performance profiles of the various exact methods described in this section.

Utilizing all available features, including MIPStart, candidate callback, user-cut callback and posting, yields the best performance, significantly improving the algorithm's efficiency and solution quality. Even when using only candidate callbacks and posting without MIPStart and user-cut callbacks, the performance remains highly effective, matching the best results, which indicates that posting is a particularly powerful technique within these methods.

The B&C approach with both callbacks is comparable with the Benders' loop method, this indicated that with the modern computational power even a more basic method is capable of performing comparably with the others.

Overall, the analysis demonstrates that while Benders' decomposition and B&C methods are both powerful, the strategic use of enhancements like callbacks, MIP start, and especially posting can significantly influence their performance, making them more efficient and adaptable to real-world scenarios.

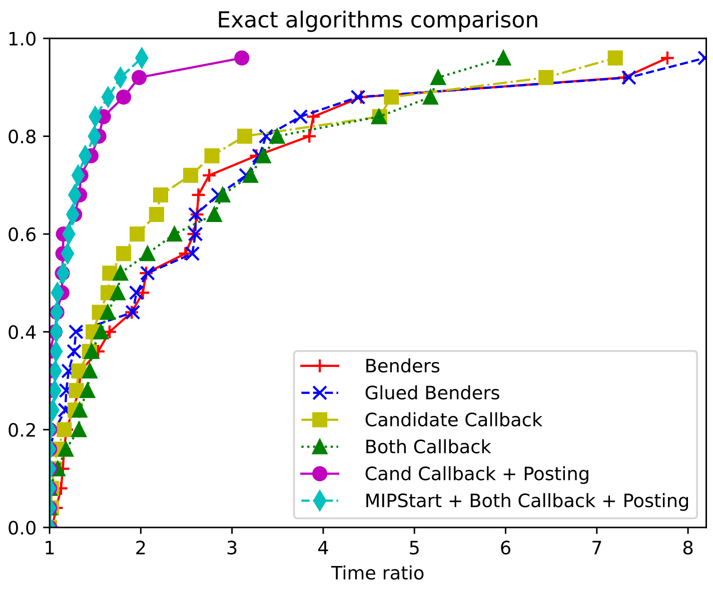

Figure 4.4: Performance profile of all exact methods.

Chapter 5

Matheuristics

A heuristic, as defined in [18], is ”any approach to problem-solving or self-discovery that employs a practical method, not guaranteed to be optimal, perfect, logical, or rational, but sufficient for reaching an immediate goal. When finding an optimal solution is impossible or impractical, heuristic methods can speed up the process of finding a satisfactory solution.” According to this definition, heuristic algorithms aim to find good solutions in a reasonable time by sacrificing optimality.

Matheuristic algorithms combine heuristic methods with mathematical programming, introducing new constraints to the model. In this way the B&C algorithm is treated as a black box, making the method independent from its implementation and from the problem. The most representative algorithm of this method is Soft Fixing (see Section 5.2).

During the solution computation, CPLEX employs various heuristic and matheuristic algorithms. By adjusting certain parameters, it is possible to change the frequency of their application or the time allocated to them.

5.1 Hard Fixing

The Hard Fixing Heuristic is an iterative approach that fixes certain variables (edges) of a reference solution computed by CPLEX and then attempts to solve the resulting simplified problem using a Mixed-Integer Programming (MIP) solver. The heuristic consists of the following steps:

1. Start from a solution
2. Fix used edges to 1 with a certain probability
3. Run MIP solver as a black box
4. Compare solutions and keep the best one

5. Remove fixed edges

6. Repeat from point 2

These steps are repeated for a fixed number of iterations. Upon finding a solution, the fixed edges are unfixed, and the process is restarted until the time limit is reached. The optimization is accelerated because the fixed edges removes a lot of feasible solutions from the domain (the nodes of a fixed edges have one degree of freedom less than before). However, optimality of the solution is not guaranteed. Importantly, each iteration's solution is at least as good as or better than the previous one.

The heuristic begins by creating the model and calling the MIP start function that generates a solution using Nearest Neighbor and the 2-opt. This initial solution is likely far from optimal, but we use it to choose some random edges and fix them to 1 by adding a costraint $x_e = 1$ to the model with a certain probability.

The percentage of edges to be fixed varies at each iteration. Generally, a high percentage is used in the initial iterations and is gradually reduced in subsequent ones, allowing CPLEX greater flexibility in computing the solution. The random selection of edges in Phase 2 ensures the algorithm terminates, as the same variables are not always fixed.

In the implemented version, the percentages used are calculated on the fly, based on how much the results improve at each iteration (a low improvement makes the percentage of free edges increase by 10%, up to 50%, allowing the solver to have a higher flexibility).

An example of the solution space exploration using this technique is shown in Figure 5.1. The pseudo-code of the Hard Fixing Heuristic is provided in Algorithm 11.

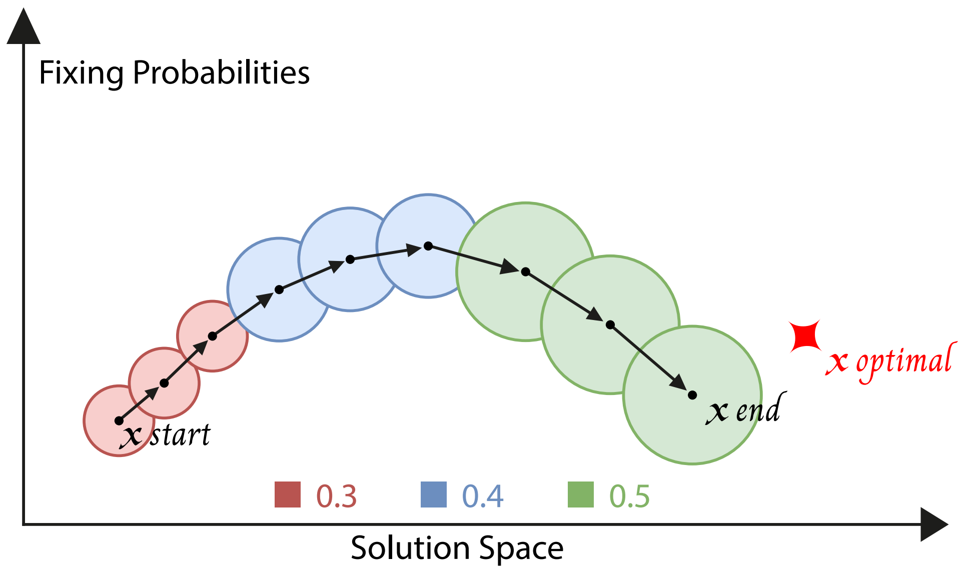

Figure 5.1: Solution space exploration using the Hard Fixing Heuristic.

Algorithm 11: Hard Fixing

```
Input: TSP Instance
Output: A valid tour
build cplex model;
current_solution ← addMipStart;
freeEdgesProb ← 0.1;
while current time < inst.time_limit do
    foreach edge of current_solution do
        if edge = 1 then
            fix edge to 1 with probability 1 - freeEdgesProb;
        end
    end
    cpx_solution ← call branch&Bound black box;
    if cpx_solution cost ≤ current_solution cost then
        current_solution ← cpx_solution;
        addMipStart(current_solution);
    end
    for each edge of current_solution do
        free edge;
    end
    if |objval - cpx_objval| < 0.1× objval then
        freeEdgesProb ← min(freeEdgesProb + 0.1, 0.5);
    end
end
```

5.2 Local Branching

The Local Branching method, also known as Soft Fixing, introduced by M. Fischetti and A. Lodi [19], adopts an approach similar to Hard Fixing. However, in this technique, the selection of variables to be set to 1 is not done randomly but is determined by CPLEX.

Starting from a feasible integer solution for the TSP, denoted as x^H, which is represented as a binary vector (e.g., [0, 1, 0, ...]) of length $|E|$, a constraint is applied to the variables with value 1:

$$\sum_{e \in E \,:\, x_e^H = 1} x_e \geq n - k \tag{5.1}$$

where the summation represents the count of variables that are set to 1 in x^H and will remain unchanged, n denotes the number of edges selected in x^H and where $k = 2, \ldots, 20$ signifies the degrees of freedom for CPLEX in finding the new solution. In each iteration of the algorithm, a new Local Branching constraint is introduced based on the current solution provided by CPLEX, while the constraint from the previous iteration is removed.

Unlike Hard Fixing, where branches are selected randomly, if there is no improvement in cost and thus no change in the solution, the branches chosen by CPLEX with the new constraint would be the same as in the previous iteration. To overcome this, the value of k starts at 2 and is incremented if the solution does not improve. Experimental results indicate that this method helps CPLEX converge more rapidly to the optimal solution, and values of k greater than 20 do not yield better outcomes. Typically, to explore the solution space, it is necessary to enumerate all the elements within it. The addition of a Local Branching constraint simplifies and accelerates this operation. Given a feasible integer solution x^H and using the Hamming distance, the k-opt solutions relative to x^H are those at a distance k from it as shown in Figure 5.2.

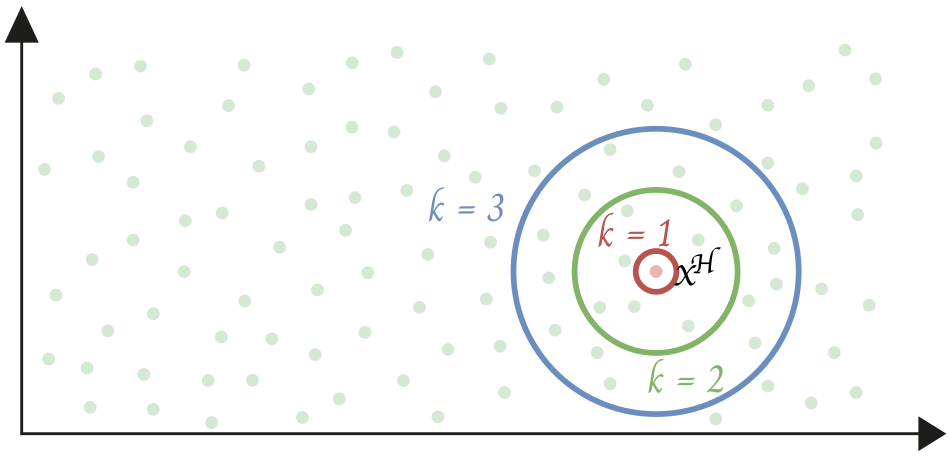

Figure 5.2: Solution space and Hamming distance.

For generic k, approximately n^k solutions should be generated at a distance k from x^H. These solutions must be analyzed to find one with a lower cost than x^H. Local Branching can also be applied to general problems, not just the TSP. Given an approximate (heuristic) solution x^H for an optimization problem, the mathematical formulation that generates solutions at a distance less than or equal to R through Local Branching is:

$$min\{c^T x \ : \ Ax = b, x \in \{0,1\}^{|E|}\} \tag{5.2}$$

$$H(x,x^h) \ = \sum_{j \in \{1...n\} \ : \ x_j^H = 0} x_j \ + \sum_{j \in E \ : \ x_j^H = 1} (1 - x_j) \ \leq R \tag{5.3}$$

Equation 5.3 represents the Hamming distance of the new solution x from x^H. The goal of Soft Fixing is to improve the solution cost by examining the nearest solutions in the space. The algorithm resembles the hard-fix approach, but instead of using the fixing-probability, we modify the neighborhood by adjusting the radius parameter k. The pseudo-code for Soft Fixing is presented in Algorithm 12.

Algorithm 12: Local Branching

```
Input: TSP Instance
Output: An improved solution
build cplex model;
current_solution ← addMipstart;
$k \leftarrow 20$;
while current time < inst.time_limit do
    Add constraint to fix $n - k$ edges;
    cpx_solution ← call Branch&Bound black box;
    if cpx_solution cost < current_solution cost then
        current_solution ← cpx_solution;
        addMipStart(current_solution);
    end
    else
        $k \leftarrow k + 10$;
    end
    Remove old constraint;
end
```

5.3 Comparison between Matheuristics

Figure 5.3 illustrates the performance profile with the cost comparison between our implementation of the Hard fixing method (Diving) and the Local Branching method. They were both given 120 seconds on a set of randomly generated inputs of various size up to 1000 nodes. It would be possible to test these methods on larger instances, but then we wouldn't be able to compare them with the optimal solution, as computing it would be too time-consuming.

We can see that both algorithms have very good performances, with an error always below 7%. The hard fixing method seems to give better results in almost all cases. Both algorithms were able to find the optimal solution in a few cases (probably with the smallest inputs).

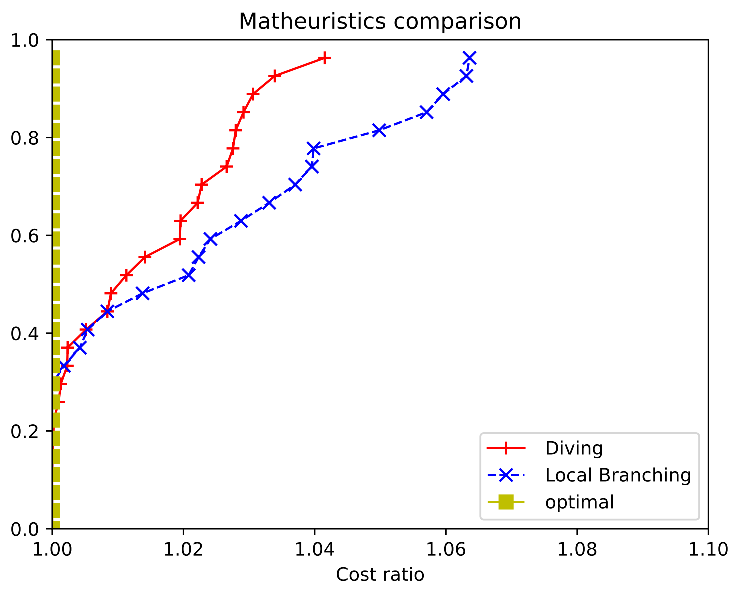

Figure 5.3: Performance profile of all Matheuristic methods.

Chapter 6

Conclusions

6.1 Performance Profile plots

The Performance Profiler [20] is a tool that facilitates the visualization of comparisons between the results of different algorithms. Depending on the type of algorithm, users can choose to compare various aspects of the results. In our case, when comparing heuristic methods, we are interested in the quality of the solutions found, so we use the solution cost as the metric. Conversely, when comparing exact algorithms, the solutions are always the same (the optimal one), but the focus shifts to the time required to find these solutions.

The performance profile classifies the execution times (or solution costs) based on the success percentage relative to a multiplication factor (ratio). The trend of the performance profile of an algorithm is monotonically increasing. The value for each ratio in the graph represents the percentage of instances that the algorithm solves with that factor compared to the optimum for that case. These graphs are often represented on a logarithmic scale to better highlight differences and achieve a clearer depiction.

The program used to create the performance profiles of the different algorithms is *perprof.py*, developed by D. Salvagnin in 2016 [21].

6.2 Heuristic and Metaheuristic methods

The diagram illustrates the performance of various heuristic and metaheuristic methods for solving the Traveling Salesman Problem (TSP): Nearest Neighbor (NN), NN with 2-opt optimization, NN with Tabu Search, and NN with Variable Neighborhood Search (VNS). The Nearest Neighbor (NN) method is the least effective, with solutions deviating 15-20% from the optimal cost, as shown by the red curve. Applying 2-opt optimization significantly improves the NN method, achieving solutions within 5% of the optimal cost, indicated by the blue curve. Both

Tabu Search and VNS perform similarly, with solutions within approximately 5% of the optimal cost, as shown by their overlapping green and purple curves. The 2-opt method alone can nearly match the performance of Tabu Search and VNS, demonstrating its effectiveness. All enhanced methods perform significantly better than the basic NN method, highlighting the importance of optimization techniques in solving the TSP.

The performance profile in Figure 6.1 shows that while NN is quick, its performance is suboptimal. Incorporating optimization techniques like 2-opt, Tabu Search, or VNS improves solution quality, reducing the error margin to within 5% of the optimal solution. The similar performance of 2-opt, Tabu Search, and VNS suggests that the choice among them can be based on computational efficiency or implementation complexity rather than solution quality. This analysis highlights the critical role of optimization in heuristic approaches to combinatorial problems like the TSP.

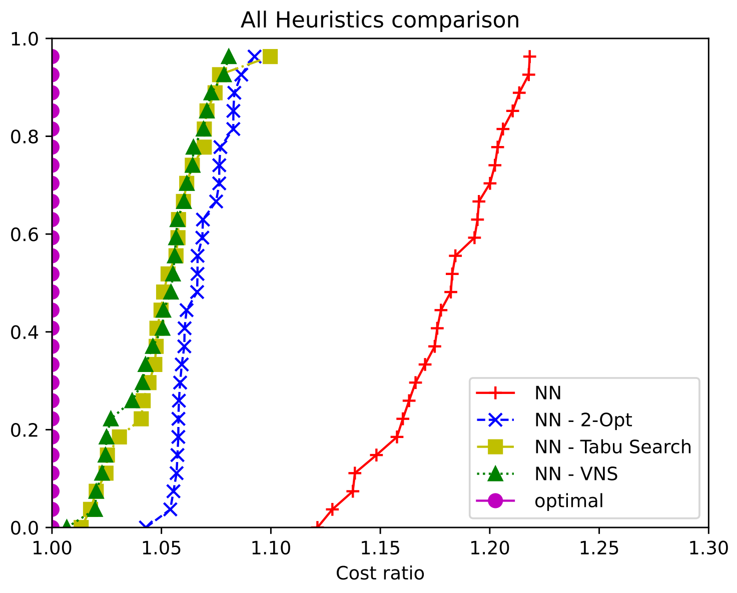

Figure 6.1: Performance profile of all Heuristics methods.

6.3 Exact Methods

The diagram in Figure 6.2 compares the performance profiles of various exact methods for solving optimization problems, focusing on Benders' decomposition and Branch-and-Cut (B&C) techniques with different enhancements. Utilizing all available features, such as MIP Start, candidate callback, user-cut callback, and posting, yields the best performance, significantly improving efficiency and solution quality. Even using only the posting method the performance remains highly effective, indicating that it is a particularly powerful technique. The plain B&C

approach with callbacks is effective as much as Benders' loop method, indicating that with our modern computational power even a simpler method can reach very good results.

Overall, the analysis demonstrates that while both Benders' decomposition and B&C methods are powerful, the strategic use of enhancements like callbacks, MIP start and especially posting significantly influences their performance, making them more efficient and adaptable to real-world scenarios.

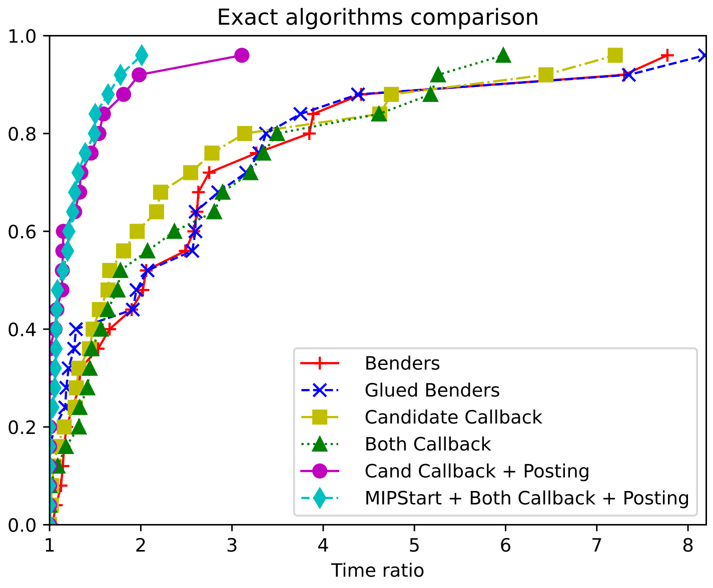

Figure 6.2: Performance profile of all Exact methods.

6.4 All Heuristics and Matheuristics Methods

From the results presented in the performance profile in Figure 6.3, it is evident that matheuristic algorithms significantly outperform both heuristic and metaheuristic algorithms. Within the same category of algorithms, performance differences are almost negligible, except for the Nearest Neighbor (NN) algorithm, which stands out as the least performing among all the considered algorithms. The application of VNS and Tabu Search significantly improves the performance of the Nearest Neighbor algorithm, although the results obtained still do not reach those of Local Branching, which clearly stands out for its better performance. However, Local Branching is consistently outperformed by the Diving algorithm, who seems to be the best one among all the non-exact algorithms.

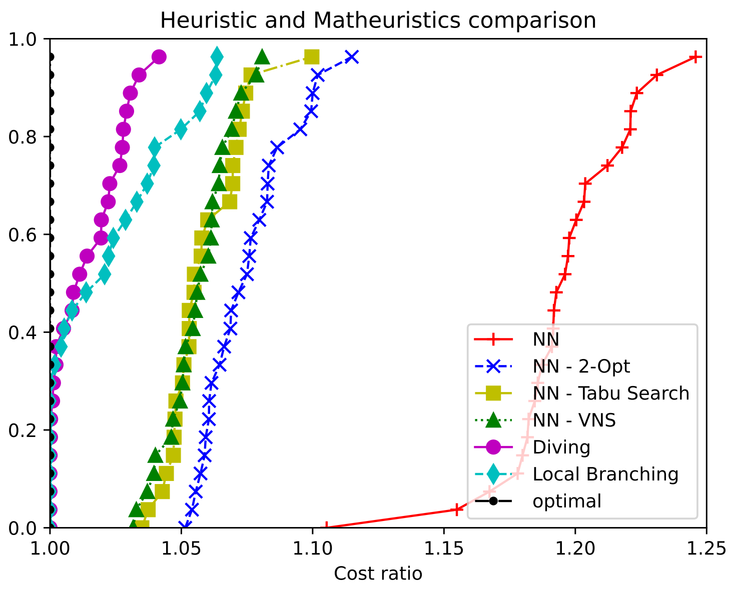

Figure 6.3: Performance profile of all Heuristic and Matheuristic methods.

6.5 Final Conclusions

In conclusion, we can affirm that each algorithm has unique characteristics that make it suitable for specific scenarios and objectives. The choice of the most appropriate algorithm depends on the specific needs of the problem to be solved and the available resources.

If the main goal is to obtain a solution in a short time, even at the cost of not necessarily achieving optimality, heuristic, metaheuristic, and matheuristic algorithms are a valid choice. These approaches use approximation strategies and advanced search techniques to find good solutions quickly, making them ideal for complex problems where speed is essential.

On the other hand, if the aim is to obtain a solution that is as close as possible to the optimum, without considering computation time as a critical factor, exact algorithms are the best choice. These algorithms are designed to explore the entire solution space, thus ensuring the possibility of finding the optimal solution or the one closest to the optimum, albeit at the cost of longer computation times.

In summary, the selection of the algorithm depends on the need to balance solution quality and execution time. Understanding the characteristics and capabilities of each type of algorithm allows for informed decisions and optimization of resources to achieve the desired results in the most efficient way possible.

Bibliography

[1] T. E. o. E. Britannica, *Np-complete problem*, https://www.britannica.com/science/NP-complete-problem, 2024.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009, isbn: 0262033844.

[3] D. Biron, "The traveling salesman problem: Deceptively easy to state; notoriously hard to solve," Senior Thesis, Liberty University, 2006. [Online]. Available: https://digitalcommons.liberty.edu/cgi/viewcontent.cgi?article=1196&context=honors.

[4] P. U. Press, *The tsp book*, —-. [Online]. Available: http://assets.press.princeton.edu/chapters/s8451.pdf.

[5] G. Reinelt, *Tsplib - a traveling salesman problem library*, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html, 1991.

[6] Wikipedia, *Travelling salesman problem — wikipedia, the free encyclopedia*, [Online; accessed 5-April-2024], 2024. [Online]. Available: https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[7] M. Fortini, "Lp-based heuristics for the traveling salesman problem," PhD thesis, University of Bologna, Bologna, Italy, 2007.

[8] M. Fischetti, *Introduction to Mathematical Optimization*, 3rd. Independently published, 2019, isbn: 1692792024.

[9] C. Nilsson, "Heuristics for the traveling salesman problem," *Link¨oping University*, Jan. 2003.

[10] K. Steiglitz and P. Weiner, "Some improved algorithms for computer solution of the traveling salesman problem," -, Jan. 1968.

[11] G. A. Croes, "A method for solving traveling-salesman problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958, issn: 0030364X, 15265463. [Online]. Available: http://www.jstor.org/stable/167074 (visited on 07/19/2024).

[12] F. Glover and M. Laguna, *Tabu search I*. KLUWER, Jan. 1999, vol. 1, isbn: 978-0-7923-9965-0. doi: 10.1287/ijoc.1.3.190.

[13] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997, issn: 0305-0548. doi: 10.1016/S0305-0548(97)00031-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054897000312.

[14] P. Hansen and N. Mladenovic, "Variable neighborhood search methods," in —-, Jan. 2009, vol. 22, pp. 3978–, isbn: 978-0-387-74758-3. doi: 10.1007/978-0-387-74759-0_694.

[15] P. Hansen, N. Mladenović, J. Brimberg, and J. A. M. Pérez, "Variable neighborhood search," in *Handbook of Metaheuristics*, M. Gendreau and J.-Y. Potvin, Eds. Cham: Springer International Publishing, 2019, pp. 57–97, isbn: 978-3-319-91086-4. doi: 10.1007/978-3-319-91086-4_3. [Online]. Available: https://doi.org/10.1007/978-3-319-91086-4_3.

[16] IBM, *Ibm ilog cplex optimization studio*, —-. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio?mhsrc=ibmsearch_a&mhq=cplex.

[17] Concorde, *Concorde tsp solver*, https://www.math.uwaterloo.ca/tsp/concorde.html.

[18] Wikipedia, *Heuristic — wikipedia, the free encyclopedia*, [Online; accessed 5-April-2024], —-. [Online]. Available: https://en.wikipedia.org/wiki/Heuristic.

[19] M. Fischetti and A. Lodi, "Local branching," *Mathematical Programming*, vol. 98, no. 1, pp. 23–47, 2003. doi: 10.1007/s10107-003-0395-5. [Online]. Available: https://doi.org/10.1007/s10107-003-0395-5.

[20] E. D. Dolan and J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, 2002.

[21] D. Salvagnin, *Performance profile*, 2016. [Online]. Available: https://www.dei.unipd.it/~fisch/ricop/OR2/PerfProf/read_me.txt.