UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Department of Information Engineering
Master's degree in Computer Engineering

Operations Research 2 Thesis

Solutions, strategies and algorithms for the Symmetric Traveling Salesman Problem

Bellin Leonardo
matr. 2080466

Pepaj Mariateresa
matr.2074284

Academic Year 2022/2023

Prof. Fischetti Matteo

Contents

Chapter 1

Introduction

The Traveling Salesman Problem (TSP) stands as one of the most extensively studied combinatorial optimization challenges within the domains of computer science and operations research. While its core concept may appear straightforward, the TSP conceals a profound level of complexity that has both historical and practical significance. This problem finds application across a spectrum of fields, including logistics, transportation and circuit layout showing also the extreme practicality of the problem.
This thesis embarks on an exploration across TSP-solving methodologies. We'll delve into some algorithmic approaches, encompassing heuristics, metaheuristics, exact methods, and matheuristics. This thesis is composed of 6 chapters describing different approaches for this problem:

- Chapter 1: We will trace the historical roots of the TSP and describe its formal mathematical formulation, providing the fundamental context needed to comprehend the problem.

- Chapter 2: The groundwork for our research project is meticulously laid out.

- Chapter 3: We present heuristic algorithms and their implementation discussing on the comparison between them.

- Chapter 4: We transition into metaheuristics, unveiling their potential to navigate the complex TSP solution space.

- Chapter 5: As we move into exact methods using tools like CPLEX, our focus sharpens on achieving precise solutions to the TSP. We will explore these methods in detail and how they can find the best possible TSP solutions.

- Chapter 6: Our journey culminates in the domain of matheuristics, an intersection of mathematical modeling and heuristic approaches. Here, we harness heuristics to enhance the mathematical modeling of TSP, bridging theory and practice.

In summary, this thesis is a comprehensive guide that covers how we implement and improve methods for solving the Traveling Salesman Problem. We'll explore the theory, algorithms, and fine-tune parameters using performance analysis, aiming to gain a better understanding of this challenging optimization problem.

1.1 History of the Problem

The problem of the travelling salesman (TSP) has no clear origin. In the German handbook 'Der Handlungsreisende Von einem alten Commis-Voyageur', it is informally described by a travelling salesman as the problem of traversing a set of German cities in the most efficient and least expensive way without passing through the same city twice. Hamilton and Kriman in the 19th century formulated the problem mathematically, but only later it became more popular: an important example is Hassler Whitney's '48-state problem.
In the 1950s and 1960s, the problem was expressed in the form of a linear integer problem by Dantzig, Fulkerson and Johnson, who developed the cutting plane method.
An important step was taken in 1972 by Richard M. Karp, who proved that the Hamiltonian cycle problem is NP-complete, which implies that the same applies to the TSP.


Der
Handlungsreisende
wie er sein soll
und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein.
Von
einem alten Commis - Voyageur.
Mit einem Titelkupfer.
Ilmenau 1832,
Druck und Verlag von B. Fr. Voigt.

Figure 1.1: Der Handlungsreisende Von einem alten Commis-Voyageur

1.2 Problem Formulation

The travelling salesman problem (TSP) can be seen informally as the problem of finding a minimum cost path that traverses a set of cities exactly once.
Mathematically, the TSP can be described as the problem of finding a minimum cost Hamiltonian circuit in a weighted complete graph, where a Hamiltonian circuit is a cycle that visits each node of the graph only once.

1.2.1 Graph model

The problem can be modeled using a undirected weighted complete graph $G = (V, E)$ where V is the set of vertices and E the set of edges e connecting two vertices. We also represent the weight of the edge $e = (i, j)$ connecting two vertices with a positive cost c_e provided by the cost function $c : E \to \mathbb{R}^+$ representing a distance function. In this thesis, the eucledian distance is used, as it is typically done for the TSP problem.

1.2.2 ILP model

The symmetic TSP can be described in the form of an integer linear programming model, in order to do so we have to describe the previous formulation as variables and linear constraints.

One of the more notable formulation is the one presented by Dantzing-Fulkerson-Johnson. The variables x_e are introduced in order to represent the selection of a determined edge:

$$x_e = \begin{cases} 1 \text{ the edge } e \text{ is selected} \\ 0 \text{ otherwise} \end{cases}$$

The TSP is a minimization problem and the function to minimize is the following:

$$min \sum_{e \in E} c_e x_e$$

where c_e is the cost previously defined. The following called degree constraints are to ensure that each node has exactly one upcoming edge an one incoming edge as per definition:

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

The last set of constraints are the subtour elimination constraints(SEC) in order to ensure that the final solution is one tour instead of a set of smaller tours:

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

The degree constraints introduced a linear number of equation but the SEC leads to an exponential number of them.

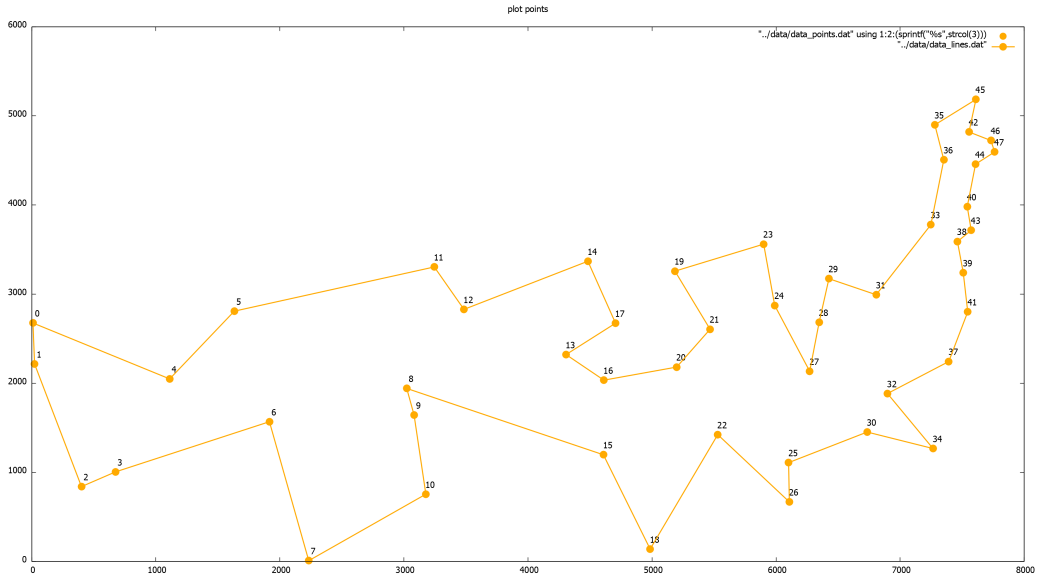


Figure 1.2: Solution of well-known instance of 48 most populated cities in the USA

Chapter 2

Setup

2.1 Software

In the development of the application utilized for running the algorithms, various software tools were employed to assist in accomplishing the study's objective. This section aims to provide a brief description of these software tools to better clarify their specific utility.

2.1.1 Visual Studio

The main software used to develop, compile and run code is Microsoft Visual Studio 2022[1], this is widely regarded as the pillar of C/C++ code development.
The software is very complete with tools and extensions that help integrate a lot of functionalities of other softwares, such as GIT.

2.1.2 GitHub

As the project entailed collaborative efforts, the utilization of a distributed version control system became necessary. For this purpose, GitHub was employed to store, share, and facilitate collaborative work on the codebase.
Additionally, GitHub provides an integrated extension with Visual Studio, further enhancing its usability in the project.
The whole source code is available at `https://github.com/THEBELLIN/Traveling-Salesman-Optimization`.

2.1.3 TSPLIB

To verify the functionality and obtain realistic input instances for testing the code, TSPLIB[2] was employed. TSPLIB serves as a comprehensive database comprising approximately 100 sample inputs for the problem at hand, several of which include the corresponding optimal solutions. The utilization of TSPLIB proved invaluable in assessing the caliber and efficacy of the utilized algorithms.

[1] https://visualstudio.microsoft.com/it/vs/
[2] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

2.1.4 IBM ILOG CPLEX

IBM ILOG CPLEX Optimization Studio[3] was employed to leverage a preexisting, highly efficient software solution that could be customized to meet specific requirements. This software possesses the capability to yield the exact optimal solution, provided sufficient time, which was unattainable solely through the utilization of heuristic algorithms.

2.1.5 GnuPlot

For the purpose of visualizing the obtained solutions, Gnuplot[4] was utilized, owing to its user-friendly interface and robust graphical capabilities. This tool proved highly valuable, particularly during the debugging phase of the project.

2.1.6 Concorde TSP Solver

In subsequent applications of CPLEX, we incorporated the utilization of certain open-source functions sourced from the Concorde TSP Solver[5], an alternative exact solver for the Traveling Salesman Problem (TSP). Specifically, these functions replaced the ones we had originally developed to perform equivalent tasks, namely, determining the count of connected components within a partial solution and partitioning the nodes within each component. This substitution resulted in a discernible enhancement in computational speed, albeit one that did not significantly impact the overall execution time.

2.2 Code structure

The source code has been compartmentalized into distinct files, encompassing both header and C code files. Each macro category was allocated a dedicated file to promote code cleanliness, readability, and facilitate ease of navigation, particularly when specific sections are of relevance.
Within each file, a multitude of ad-hoc functions were developed, primarily tailored to address specific algorithms needs. Subsequently, these functions were generalized to maximize their reusability across the entirety of the project.
Below we briefly describe the functions of each of the main files.

2.2.1 Instance Structure

This is the main structure that holds the input and the parameters of the problem. This provides an easy way to access all of these information thorughout all functions in the project by using only a pointer.
It holds input parameters such as:

- input file name: the TSPLIB file name, if any is used.
- output file name: the output file name for printing the solution with Gnuplot.
- points: the pointer to an array of points, saved as a structure with 2 real coordinates (x, y).

[3]https://www.ibm.com/it-it/products/ilog-cplex-optimization-studio
[4]http://www.gnuplot.info/
[5]https://www.math.uwaterloo.ca/tsp/concorde.html

- number of points: the number of points in the above mentioned array.
- solver to use: the algorithm specified to solve the instance, together with its specific parameters.
- time limit: the time limit, in seconds, of the running time of the program.
- current solution: current cycle found by the algorithm used, together with its current cost.
- best solution: best TSP solution found by the algorithm so far, together with its best cost.
- verbosity level: the verbosity level of the program, mostly used for debugging.
- cost matrix: the matrix with the precomputed distance between all pair of points.

2.2.2 "Main" file

The primary file encompasses the core execution sequence of the program. It follows a structured flow, commencing with the initialization of the instance, followed by parsing the command line arguments. Subsequently, the TSPLIB file is parsed, and the points within the instance are stored in the designated data structure. Alternatively, if the user specifies, the points can be randomly generated. Following this, the cost matrix is computed. The designated algorithm is then executed on the instance to generate the solution. Finally, the solution is written to a file and, if desired, visualized through plotting using Gnuplot.

2.2.3 "TSP" file

Within the TSP files, one can find the comprehensive set of problem-specific data structures, encompassing the instance, point, and edge representations. These files also incorporate enums, which enhance the readability of function arguments. Furthermore, the TSP files contain the principal functions utilized throughout program execution, specifically those not tied to algorithm-specific implementations.

2.2.4 "Heuristics" file

The heuristics files house the data structures responsible for consolidating information regarding user-specified parameters and the selected solver. These files also include enums that, once again, contribute to improve the readability of certain functions.
The functions within these files are specifically associated to the implementation of the heuristic algorithms developed, namely: extra mileage, nearest neighbor, variable neighborhood search (VNS), tabu search, genetic and simulated annealing.

2.2.5 "Cplex" file

The cplex files encompass the entirety of functions and code pertaining to the utilization of IBM ILOG CPLEX Optimization Studio. Primarily, these files contain utility functions, such as transforming a solution from the CPLEX form into the instance form, among others. Additionally, these files comprise the implementation of the Benders loop method, along with all the necessary functions employed in callbacks, as well as those responsible for generating and incorporating cuts into the Mixed Integer Programming (MIP) problem.

2.3 Performance Profiling

In order to facilitate a comprehensive comparison of the algorithms and assess their performance, as well as fine-tune algorithmic parameters, we employed Python code[6] for conducting performance profiling [1]. Our experimentation involved the utilization of 20 randomly generated instances, each comprising points distributed within a 10,000 by 10,000 square domain. Specifically:

- Heuristics were executed with a time limit of 100 seconds on instances containing 300-400 nodes.
- Metaheuristics were executed with a time limit of 200 seconds on instances with 300-400 nodes.
- Exact models were executed with a time limit of 300 seconds on instances containing 200-350 nodes.
- Matheuristics were executed with a time limit of 500 seconds on instances containing 450-600 nodes.

[6] Available at `http://www.dei.unipd.it/~fisch/ricop/OR2/PerfProf/`

Chapter 3

Heuristic Algorithms

Heuristics are problem-solving techniques that have as main scope efficiency over guaranteeing optimal solutions.
These methods, while not guaranteed to yield the absolute best solution, aim to produce reasonably good solutions in a reasonable amount of time.
In this chapter we're going to introduce some heuristic algorithms and their implementation.

3.1 Greedy Heuristics

Greedy heuristics make locally optimal choices at each step hoping to provide a good solution.
The term greedy comes from the fact that at each decision point we try to maximize or minimize a certain criterion .
The main advantage of the greedy heuristic lies in its speed and simplicity, making it a valuable tool for large instances.
The algorithms we're going to describe are: the nearest neighbor algorithm and its improvements and the extra mileage algorithm.

3.1.1 Nearest Neighbour

One of the most intuitive algorithms used for solving the TSP problem is the Nearest Neighbour algorithm.
This algorithm makes a greedy choice by selecting, at each step, the incumbent edge with the minimum cost among all not already selected. We continue to find the nearest node until we have visited all possible nodes, resulting in an ordered list of visited nodes seen as a solution of the problem.
Although the Nearest Neighbour algorithm is heuristic and there is no proof that the solution is optimal, it is still widely used due to its ease of implementation and quick execution time of $O(n^2)$, where n is the number of vertices. This cost is due to the continuous search for the minimum distance for every node.
Despite not being exact, the Nearest Neighbor algorithm can provide a reasonable solution, especially under certain conditions.
In summary, the Nearest Neighbour algorithm is an effective and efficient heuristic algorithm for solving certain types of problems.
While it may not always provide the optimal solution, it is still widely used due to its simplicity and speed. By incorporating additional techniques such as selecting the best starting

point or introducing randomness, the algorithm's solution can be further improved.

Algorithm 1 Nearest Neighbour

Require: Instance
Ensure: a valid tsp tour
$len \leftarrow 0$
$cost \leftarrow 0$
$last\_visited \leftarrow start$
$min \leftarrow +\infty$
$min\_pos \leftarrow 0$
// initialize current with the trivial permutation 0-1-2....-n
$currsol \leftarrow$ permutation
// moves at the beginning the first covered node which is the starting node
swap(currsol, 0, $start$)
$len \leftarrow len + 1$
for $i \leftarrow 1$ **to** $nnodes - 1$ **do**
// finds minimum cost node from the last visited node of the tour
min_pos, min $\leftarrow$ minimum(currsol, len)
// update the cost and rearranges the current array
$costo \leftarrow costo +$ min
swap(currsol, len, min_pos)
min $\leftarrow +\infty$
$last \leftarrow$ currsol[len]
$len \leftarrow len + 1$
end for
$costo \leftarrow costo +$ COST($start, last$)

3.1.2 Techniques for Nearest Neighbour

To imporve further the solution of the NN algorithm we also implemented the following techniques:

- **All start**: which selects the best starting point, this tecnique can be easily implemented in $O(n^3)$ time by trying all n possible starting nodes in a for loop.

- **GRASP(Greedy Randomized Adaptive Search Procedure)**: we incorporate an element of randomness by introducing a stochastic element in node selection. This involves making choices from the set of nearest, second-nearest, or third-nearest nodes with specific probabilities.
  The GRASP method offers two adjustable parameters: the probabilities associated with selecting the second or third nearest nodes.
  These parameter values were determined through a heuristic process that involved performance profiling across 20 different instances.
  The outcomes of this tuning process are as follows:

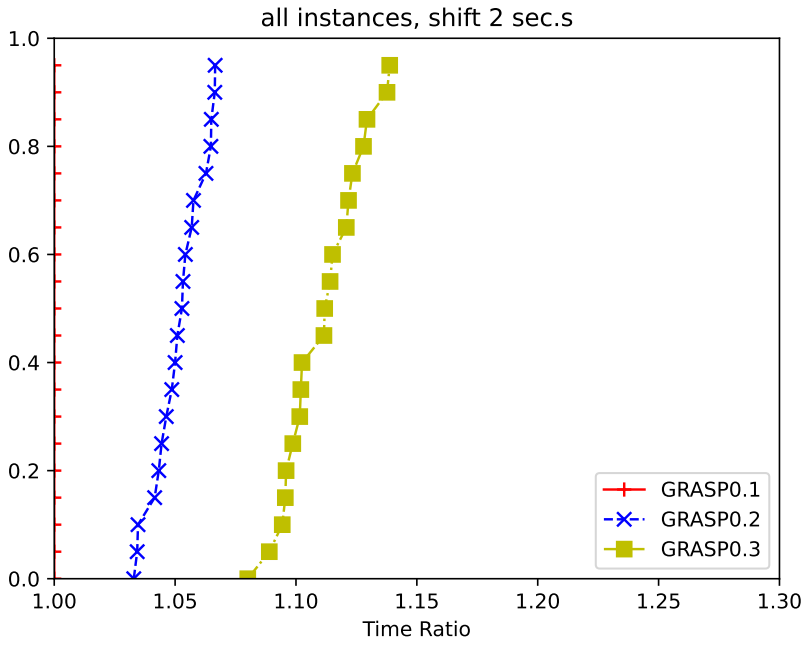


Figure 3.1: parameter tuning for p2

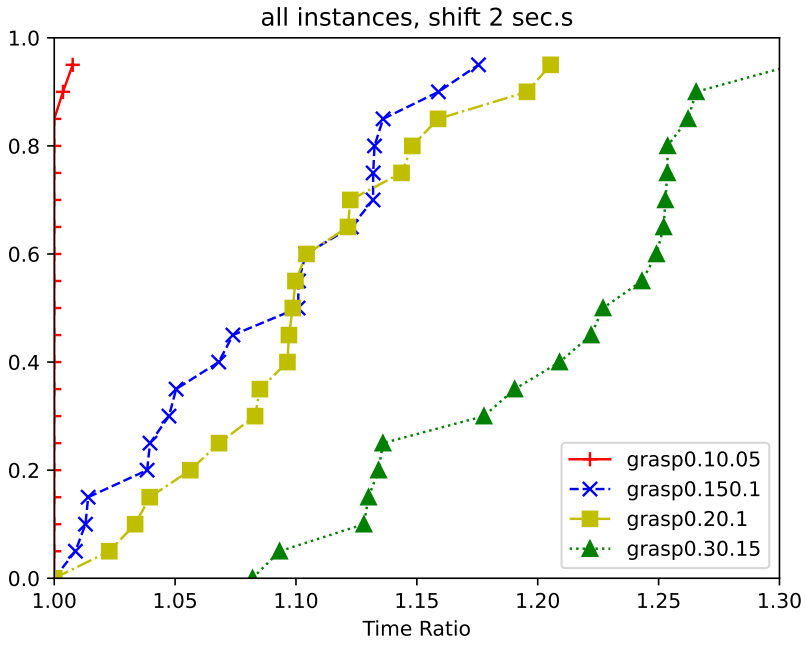


Figure 3.2: parameter tuning for p2 and p3

The visual analysis of the images leaves us with a distinct conclusion: among the instances under consideration, the optimal parameters are identified as $p_2 = 0.1$ and $p_3 = 0.05$. Overall, these techniques improve substantially the standard nearest neighbor, as we will see in section 3.3.

3.1.3 Extra Mileage

Another simple yet effective algorithm is what's called Extra Mileage algorithm. The name resembles the main idea behind its iterative process, that starts with a subtour of the whole instance or even with just some starting points at random.
The algorithm follows a loop of iterations, each of which aims at adding to the current tour one of the points that is not already in it, by computing the added cost that this operation would imply (the so called "Extra mileage") and then choosing the one that has the lowest.

Algorithm 2 Extra Mileage

Require: Starting subtour or starting points
Ensure: a valid tsp tour
starting_points ← collection of starting points, chosen at will
while *current_nodes* < *nnodes* **do**
min_extra_cost ← $-\infty$
//Scans all the edges in the current tour
for *i* ← 0 to *curr_nodes* **do**
//Considering edge e=(currsol[i], currsol[i+1])
e.to ← currsol[i]
$e.from$ ← currsol[i+1]
for *j* ← *curr_nodes* to *nndoes* **do**
//Considering point currsol[j], not already in the subtour
added_cost ← - COST(e.from, e.to) + COST(e.from, inst->currsol[j]) +
COST(inst->currsol[j], e.to)
if *added_cost* < min_extra_cost **then**
min_extra_cost ← *added_cost*
new_point ← *j*
place ← *i* + 1
end if
end for
end for
add node j to the subtour
curr_nodes + +
end while

3.1.4 Techniques for Extra Mileage

As we seen above, the extra mileage algorithm is flexible on how the starting points are chosen. We tested and implemented three possible methods for choosing the starting points:

- **Random**: the algorithm randomly chooses 2 nodes a, b to be the starting points, thus generating the starting subtour a->b, b->a.
- **Maximum distance**: the algorithm finds the 2 points whose relative distance is maximised and uses those two as generating the starting subtour.
- **Convex Hull**: the algorithm finds the convex hull of the set of points in the TSP instance and uses it as the starting subtour.

We conducted performance profiling tests on the three starting options to analyze their relative effectiveness.

The results are presented in Figure 3.3, which provides a visual representation of the findings. From the graph, it is evident that the convex hull starting option outperforms the other two alternatives, while the remaining two options exhibit similar performance.

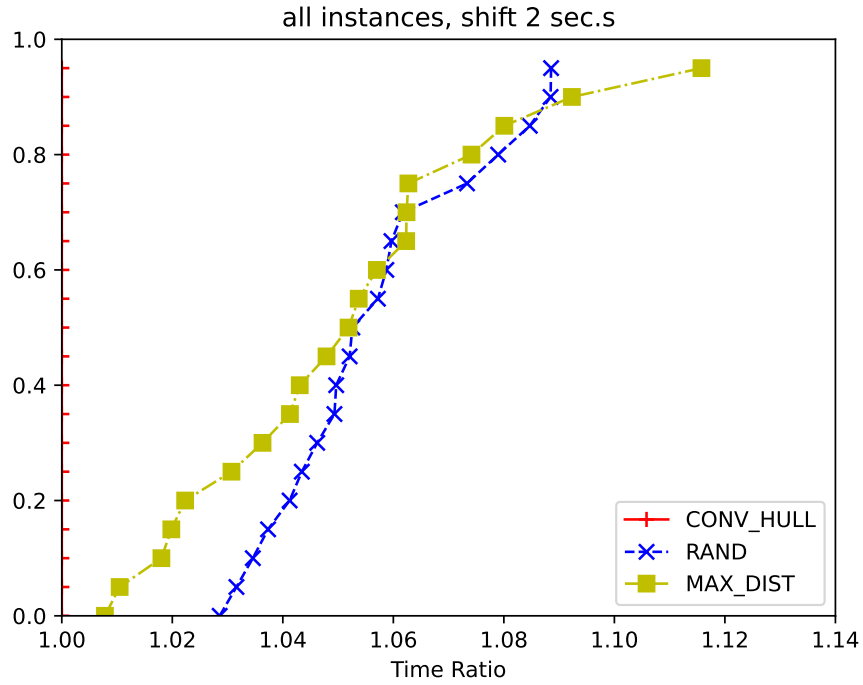


Figure 3.3: performance profiling for the best starting points option

As is the case for Nearest Neighbour we also implemented a GRASP version of the Extra mileage algorithm, both with the possibility of choosing only the second best node aswell as the one which can choose up to the third best node. The respective algorithms use the same pseudocode shown above with the adjustment to keep track of the second and third best option, and then chooses which to introduce based on a probablity draw.
In this scenario, we also conducted experiments using the same parameter values as those used for the Nearest Neighbor grasp for determining the probability of selecting the second-best decision.
As indicated by the performance profiling in Figure 3.4, it is evident that the optimal parameter value in this case is $p_2 = 0.3$. The starting points option considered was the previously tuned parameter convex hull.

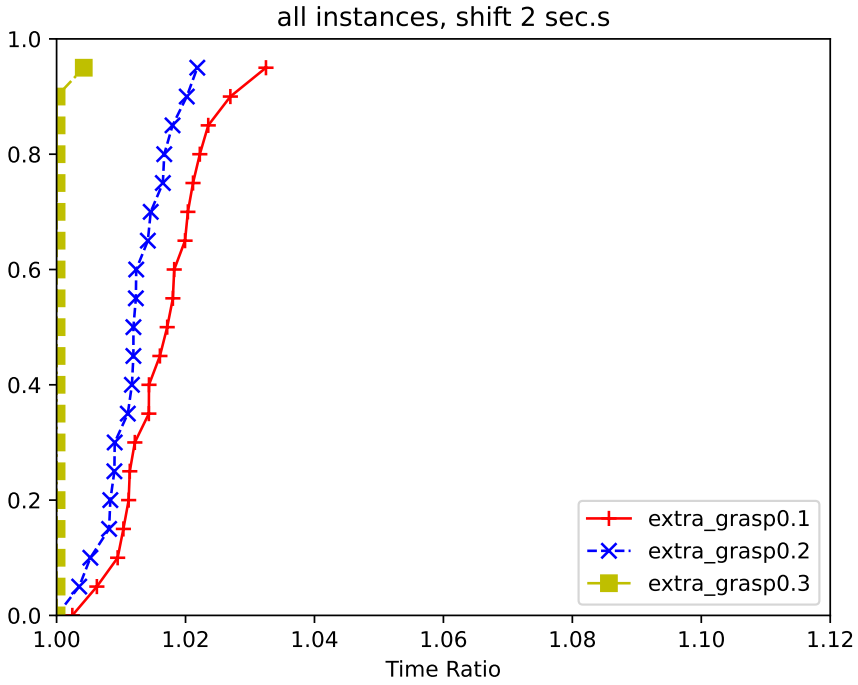


Figure 3.4: performance profiling for the best parameter for grasp2

3.2 Refinement heuristic

Starting with an initial solution, refinement heuristics examine neighboring solutions by making minor improvements to the current solution.
These adjustments could involve swapping and reordering elements to explore potential improvements.
The core principle is to gradually improve the solution within the immediate vicinity of the current solution. This localized exploration enables the algorithm to navigate the solution space more efficiently.

3.2.1 2-OPT

The 2-OPT algorithm is a widely used refining heuristic that aims to improve the quality of solutions to the traveling salesman problem (TSP).
Typically, the algorithm begins with an initial solution, which is either generated randomly or with a greedy heuristic.
It is essential to note that the quality of the initial solution highly impacts the resulting solution obtained by the algorithm. Specifically, the 2-OPT algorithm focuses on improving solutions within the 2-neighborhood of the initial solution, leading to a suboptimal solution.

The primary goal of the 2-OPT algorithm is to iteratively swap pairs of edges in a tour to minimize the total cost of the tour. At each iteration, the algorithm considers all pairs of edges that do not share an endpoint and calculates the best cost reduction that would result from swapping them.
To perform a 2-OPT swap, the algorithm selects two edges in the tour, $(i, i+1)$ and $(j, j+1)$, and swaps them with the edges (i, j) and $(i+1, j+1)$, respectively. The algorithm checks

whether the delta of the cost improves, i.e., if $\Delta c = c(i,j) + c(i+1,j+1) - c(i,i+1) - c(j,j+1) \leq 0$.
If the swap reduces the total cost of the tour, it is performed, and the process continues until no further improvements can be made, or the algorithm reaches the time limit. An example of a 2-OPT swap is shown in Figure 3.5.

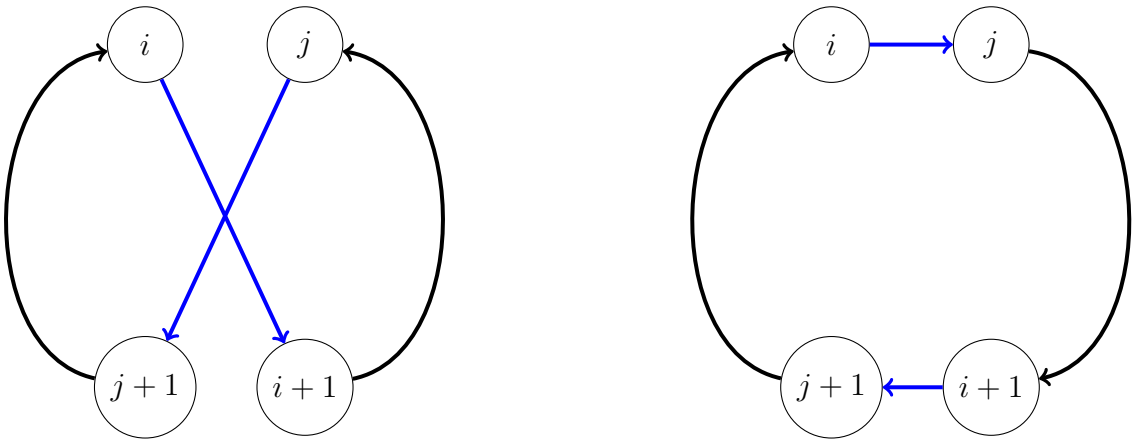


Figure 3.5: example of a 2opt swap

In summary, the 2-OPT algorithm is a local search technique that works by iteratively swapping pairs of edges in a tour to minimize the total cost of the tour. The algorithm's effectiveness is highly dependent on the initial solution, and it focuses on improving the solution within the 2-neighborhood. To choose an initial solution, we experimented with different greedy starting points and conducted a comparison using performance profiling. We evaluated five options, namely:

1. Nearest Neighbor All Start
2. NN_GRASP iterative grasp with optimal parameter $p_2 = 0.1$ for 100 seconds
3. NN_GRASP one iteration of grasp with parameter $p_2 = 0.3$ and random choice each 20 iterations
4. NN_GRASP one iteration of grasp with parameter $p_2 = 0.3$ and random choice each 50 iterations
5. Random Permutation

The results of these comparisons are as follows:

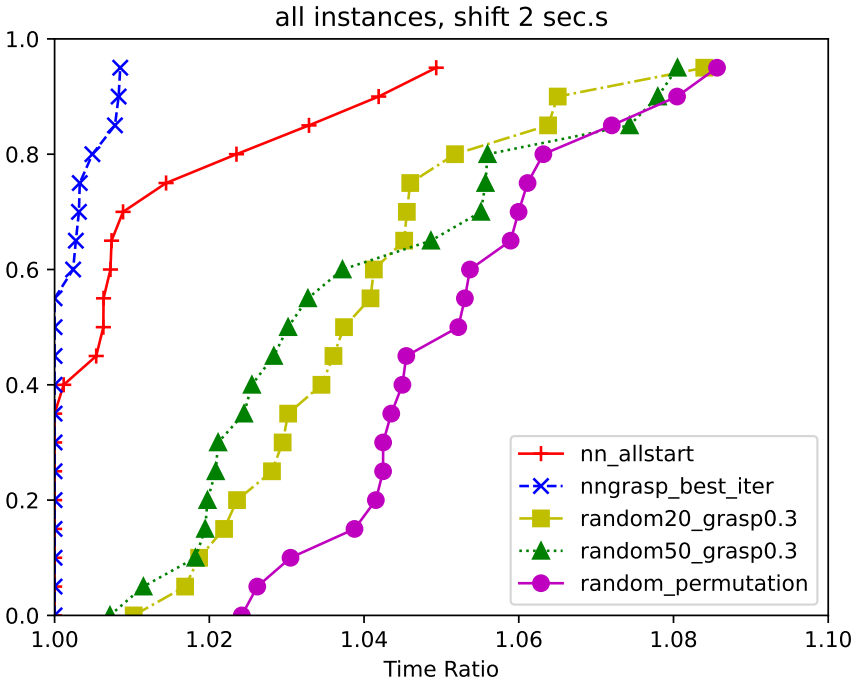


Figure 3.6: performance profiling for the best starting solution for 2 opt

the optimal iterative nearest neighbor GRASP algorithm emerged as the best initial solution.

Algorithm 3 two opt

Require: Instance
Ensure: a valid tsp tour
    best_delta $\leftarrow -1$
    while best_delta < 0 **do**
        best_delta $\leftarrow 0$
        best_i $\leftarrow -1$
        best_j $\leftarrow -1$
        for $i \leftarrow 0$ **to** $n-2$ **do**
            for $j \leftarrow i+2$ **to** n **do**
                $\delta \leftarrow c_{i,j} + c_{i+1,j+1} - c_{i,i+1} - c_{j,j+1}$
                if $\delta <$ best_delta **then**
                    best_i $\leftarrow i$
                    best_j $\leftarrow j$
                    best_delta $\leftarrow \delta$
                end if
            end for
        end for
        if best_delta < 0 **then**
            invert_nodes(currsol, best_i $+ 1$, best_j)
            currcost $\leftarrow$ currcost $+$ best_delta
        end if
    end while

3.3 Comparison between heuristics

In this section, we delve into an in-depth evaluation of the implemented heuristic algorithms, focusing on their comparative performance. Through meticulous evaluation using the previously fine-tuned parameters, we conducted a performance profile.
As illustrated in Figure 3.7 among the various greedy heuristics, the extra mileage greedy and deterministic algorithms demonstrate better performance. These two algorithms consistently outperform the allstart and iterative grasp nearest neighbor approaches, which exhibit relatively similar performance. Finally, at the lower end of the spectrum, we find the iterative grasp with two probabilities, which yields comparatively worse results.

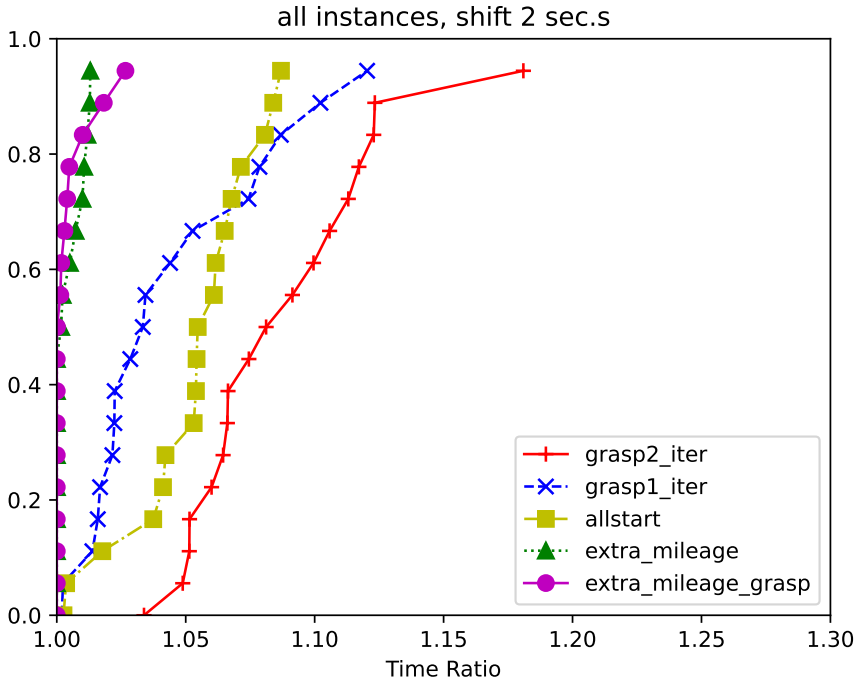


Figure 3.7: performance profiling heuristics

Chapter 4

Metaheuristics Algorithms

Metaheuristic algorithms are a class of computational techniques that can be used for solving optimization problems without relying on explicit problem-specific knowledge.
These algorithms draw inspiration from natural phenomena, social behavior, and physical processes to simulate powerful search strategies, allowing them to transcend local optima and explore global solution spaces.
These type of algorithms use various tecniques to escape a local minimum, which can be found with the 2-opt algorithm, and explores different neighborhoods reoptimizing the problem in order to find different local minima leading hopefully to a global minimum.
Still making use of refinement heuristics for optimizing the solution metaheuristics don't lead necesserly to an optimal solution.
In this chapter, we'll dive deeper into several well-known metheuristic algorithms, exploring their inner workings within the context of solving the Traveling Salesman Problem (TSP). The algorithms we'll be discussing include Variable Neighborhood Search (VNS), Simulated Annealing, Tabu Search, and Genetic Algorithm.

4.1 Variable Neighborhood Search (VNS)

The Variable Neighborhood Search (VNS) is a metaheuristic algorithm first proposed by Mladenović and Hansen in 1997[2], it aims to improve the quality of solutions by iteratively exploring different neighborhoods of a given solution in a systematic way.
At each iteration, the algorithm applies a local search method to obtain a locally optimal solution within a given neighborhood.
When the algorithm gets stuck on a local minimum, it escapes the local minimum by changing a set of edges, known as a "kick," and then tries to find a suboptimal solution in the new neighborhood using the 2-OPT algorithm.
It basically identifies a new starting point for the local optimization process ensuring that the previous solution is not revisited in the next neighborhood.

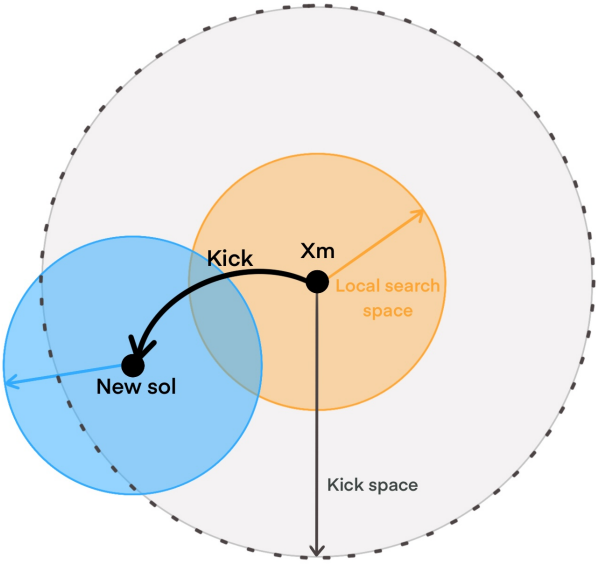


Figure 4.1: VNS local and kick space

The kick operation in VNS involves rearranging a set of edges in a way that is different from the local optimum and it has to change more than the neighborhood that is optimized. This operation is performed by randomly selecting a set of edges and changing the order in which they appear in the tour.In our case the kick is completely deterministic and the number of edges selected is 3.
The resulting solution is then improved using the 2-OPT algorithm. The VNS algorithm continues to explore different neighborhoods and apply local search methods until a stopping criterion is met, which can be a fixed number of iterations or a time limit.

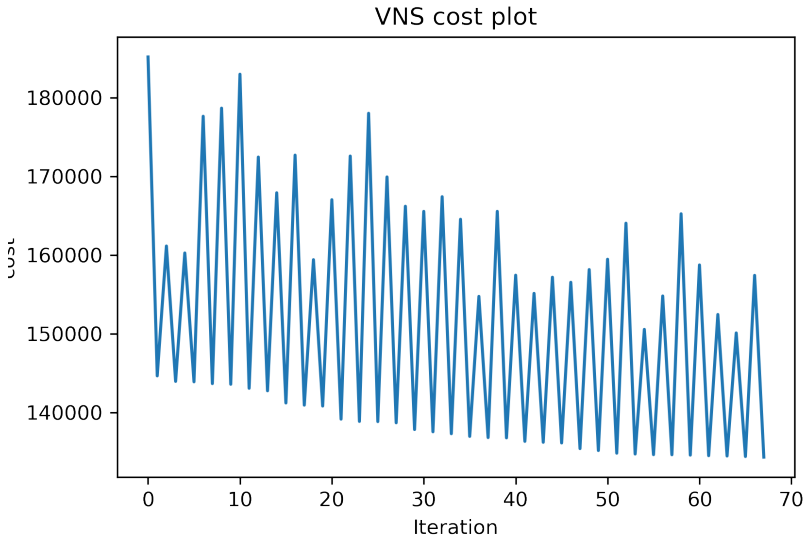


Figure 4.2: cost plot through iterations

Algorithm 4 VNS

Require: Instance, time_limit
Ensure: a valid tsp tour
    bestsol ← two_opt(instance)
    while time_limit is not reached **do**
        currsol ← bestsol
        currsol ← kick(currsol).
        currsol ← two_opt(instance)
        if currcost < bestcost **then**
            bestsol ← currsol
            bestcost ← currcost
        end if
    end while

4.2 Simulated annealing

Simulated annealing is a probabilistic algorithm first introduced for optimization problems by S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi in 1983 [3].
The algorithm, functioning as a metaheuristic, aims to escape local minima in pursuit of global minima or better solutions. The main concept emulates the physical process of cooling a solid until it reaches equilibrium, corresponding to the state of minimum energy configuration.
Initially, the algorithm starts with a given solution and applies a random perturbation to generate a new possible solution.
This new solution is accepted either if it yields an improvement in the cost or with a probability that diminishes exponentially, defined as:

$$\exp\left(-\frac{z_{\text{new}} - z_{\text{best}}}{T * z_{\text{best}}/n}\right)$$

T represents the current system temperature, which is gradually reduced over iterations following a cooling schedule.
In our case, we employed a proportional cooling strategy:

$$T_{i+1} = \alpha \cdot Ti$$

starting with an initial temperature T_0 in the first iteration, the cooling parameter α was fine-tuned.
We conducted performance profiling using a fixed $T_0 = 100$, yielding the outcomes depicted in Figure 4.2.

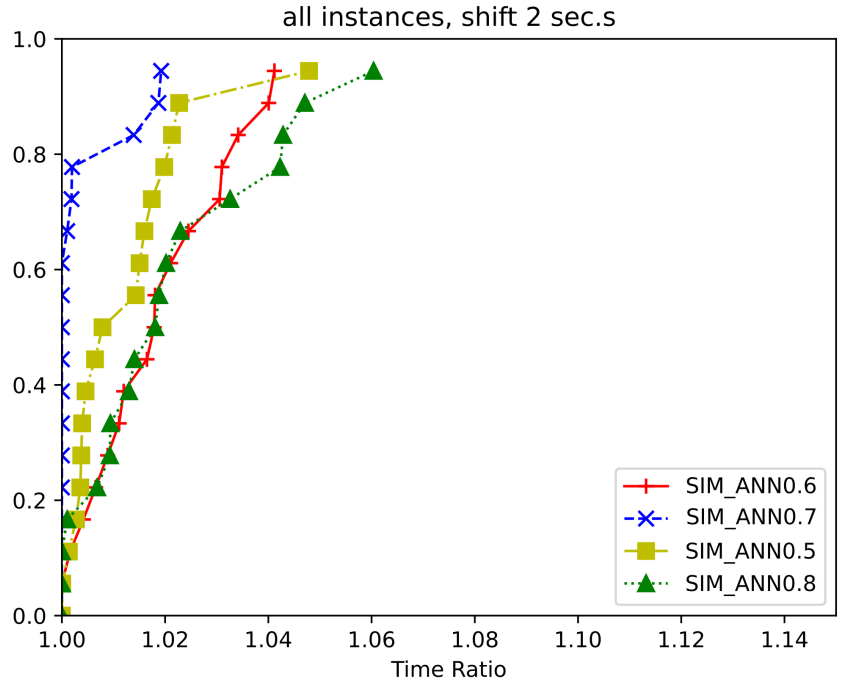


Figure 4.3: Cooling factor parameter tuning

Upon analyzing the figure, we determined that a cooling factor $\alpha = 0.7$ was most suitable for the considered instance type. With this technique, we intentionally allow the solution quality to degrade initially, allowing to escape from local minima. As the temperature gradually decreases, only moves leading to improved solutions are accepted.

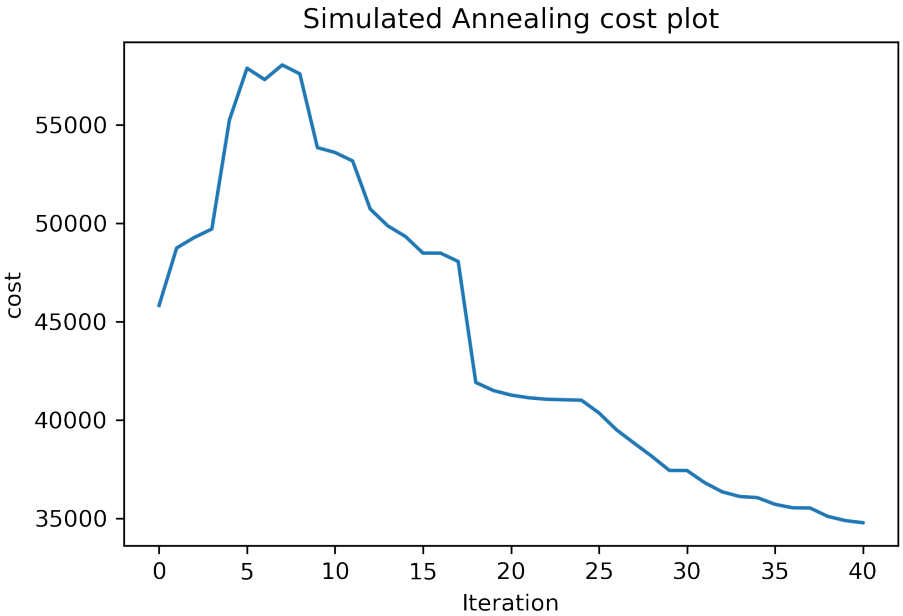


Figure 4.4: cost of the accepted solutions

Algorithm 5 Simulated annealing

Require: Instance, time_limit
Ensure: a valid tsp tour
bestsol ← two_opt(inst)
currsol ← bestsol
T ← T_0
while time_limit not reached **do**
costDiff ← random_two_opt_move(currsol, inst) ▷ Swap two random cities
deltaz ← $\frac{\text{costDiff}}{\frac{\text{bestcost}}{n}}$
random ← rand01()
expo ← $\exp\left(-\frac{\text{deltaz}}{T}\right)$
if costDiff < 0 **or** expo $>$ random **then** ▷ Accept solution
bestsol ← currsol
bestcost ← currcost
end if
T ← T $\times$ α ▷ Update the temperature
end while

4.3 Tabu Search

Tabu Search is a metaheuristic optimization algorithm commonly used to solve complex combinatorial optimization problems, including the Traveling Salesman Problem (TSP). It was introduced by Fred Glover [4] in the late 1980s and is inspired by various search and optimization principles, such as local search, diversification, and intensification.

Here's a brief explaination of the main components of the Tabu Search algorithm for the TSP problem:

- **Initialization**: The algorithm begins with an initial solution, which can be generated randomly or using a heuristic method like the ones described in chapter 3.
  This initial solution is treated as the current best solution.

- **Tabu List**: Tabu Search maintains a "tabu list" to keep track of recently visited solutions or solution modifications.
  This list prevents the algorithm from revisiting the same solutions repeatedly, promoting diversification in the search process and avoid getting stuck on local minima.

- **Neighborhood Search**: Tabu Search explores the neighborhood of the current solution by making small modifications.
  These modifications can include swapping two nodes in the tour or reversing a segment of the tour.
  The objective is to find a neighboring solution that improves upon the current best solution, if this is not possible the algorithm chooses the best permitted tour.

- **Termination Condition**: Tabu Search continues to explore the solution space, iteratively improving the best-known solution, until a stopping criterion is met.
  This criterion can be a maximum number of iterations, a time limit, or a specified level of improvement.

- **Output**: The best solution found during the search is returned as the final solution to the TSP problem.

Tabu Search is known for its ability to efficiently explore the solution space, balance between exploration and exploitation, and find high-quality solutions for TSP instances. By maintaining a tabu list and applying various search strategies, it avoids getting trapped in local optima and systematically refines the tour until a near-optimal solution is reached. It is a versatile algorithm that can be adapted and fine-tuned for specific TSP variants and problem instances.

The tenure parameter plays a crucial role in the effectiveness of the tabu search algorithm. To determine the optimal value, we experimented with different values and fine-tuned it using performance profiling.
The results are illustrated in Figure 4.5, which showcases the performance variations for different tenure values.

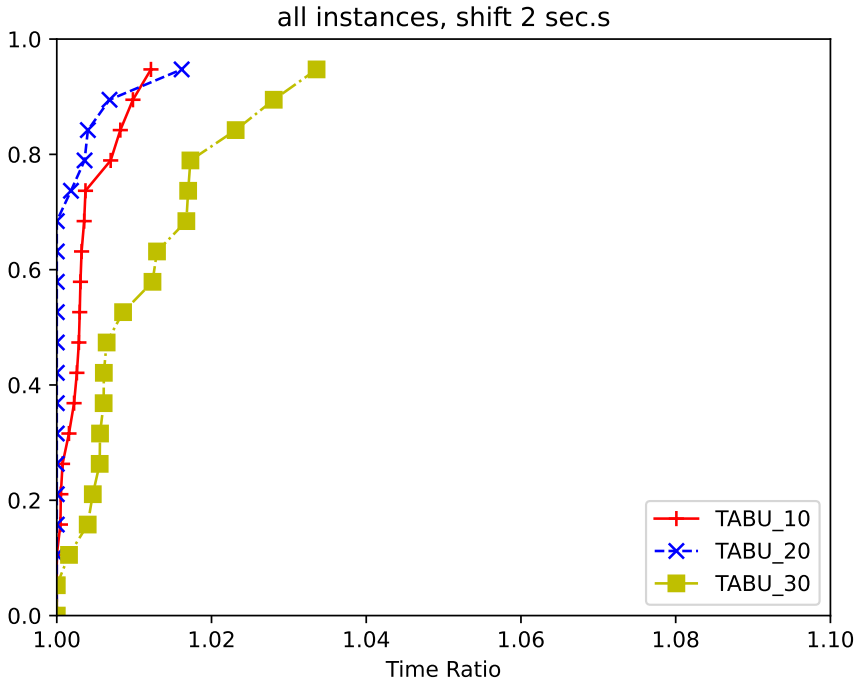


Figure 4.5: performance profiling for tuning the tabu tenure

Upon analysis, it is apparent that the three tenure values exhibit relatively similar performance. However, the 30 value shows slightly inferior results compared to 20 and 10. After careful consideration, we decided to select a tabu tenure of 20 as the optimal choice,despite its close proximity to 10.

Algorithm 6 Tabu Search

```
//Initialization
  S_current ← Initial solution (e.g., random or nearest neighbor)
  S_best ← S_current
  Tabu list TabuList ← Empty
  Termination condition (e.g., maximum iterations)
while Termination condition not met do
    Find the best neighbor S_neighbor of S_current not in TabuList.
    Update TabuList with the move that led from S_current to S_neighbor.
    if TabuList is too long (based on tabu tenure) then
        Remove the oldest entry from TabuList.
    end if
    if S_neighbor is better than S_best then
        S_best ← S_neighbor
    end if
    S_current ← S_neighbor
end while
Output: S_best (the best solution found)
```

4.4 Genetic Algorithm

A Genetic Algorithm is a powerful optimization technique inspired by the process of natural selection and evolution [5]. When applied to the Traveling Salesman Problem, it offers a unique and effective approach to finding near-optimal solutions.
In a Genetic Algorithm for TSP, a population of potential solutions (represented as tours that visit nodes in a specific order) evolves over generations.
The algorithm uses genetic operators such as selection, crossover, and mutation to create new solutions and iteratively improve the population's quality.

Here's a brief explaination of the main components of the Genetic Algorithm for the TSP problem:

- **Initialization**: A population of potential solutions is created. These initial tours can be generated randomly or using heuristics.
- **Selection**: Tours in the population are selected to form a mating pool, with a preference for tours that represent shorter routes, i.e. better solutions. This selection process simulates the survival of the fittest.
- **Crossover**: Pairs of tours from the mating pool are combined to create new tours, known as offspring. The crossover operator combines genetic information from the parent tours to generate potentially better solutions.
- **Mutation**: Random changes are introduced into some of the offspring tours to maintain diversity and explore the solution space more thoroughly. Mutation prevents the algorithm from getting stuck in local optima.
- **Evaluation**: The fitness of each tour (the total distance traveled) is calculated, and tours are ranked based on their fitness.

- **Replacement**: The least fit tours in the population are replaced with the new offspring, ensuring that the population size remains constant.
- **Termination**: The algorithm repeats the selection, crossover, mutation, evaluation, and replacement steps for multiple generations. Termination criteria, such as a maximum number of generations or a target improvement level, determine when to stop the search.
- **Output**: The best tour found during the algorithm's execution is returned as the solution to the TSP.

Genetic Algorithms offer a flexible and robust approach to solving the TSP. They can handle large problem instances and often find high-quality solutions, though not necessarily the global optimum. Through the interplay of selection, crossover, and mutation, genetic algorithms efficiently explore the solution space, gradually improving the tours until a satisfactory or near-optimal solution is discovered.

Even if they are not among the fastest or well performing algorithm nowdays, Genetic Algorithms for the TSP have still been widely applied and adapted to various TSP variants and problem sizes, making them a valuable tool in the field of combinatorial optimization.

Here is the pseudocode for the Genetic algorithm in order to better understand the steps it takes:

Algorithm 7 Genetic Algorithm

```
//Initialization
  Population ← Randomly generated tours
  Termination condition (e.g., maximum generations)
while Termination condition not met do
    Evaluate fitness for each tour in Population
    Select parents for crossover based on fitness using roulette wheel selection
    Create offspring by performing crossover on pairs of parents
    Apply mutation to some of the offspring tours
    Evaluate fitness for the offspring
    Replace some tours in Population with the offspring
end while
Output: The best tour found in Population
```

4.5 Comparison between metaheuristics

In this section, we provide an analysis on the implemented metaheuristic algorithm and their relative performance in comparison to each other. Utilizing the previously fine-tuned parameters obtained, we conducted a thorough evaluation. The results, as depicted in Figure 4.6, clearly indicate that the Variable Neighborhood Search (VNS) stands out as the most effective metaheuristic. On the other hand, the genetic algorithm exhibits the poorest performance among the evaluated methods. Additionally, both the tabu search and simulated annealing demonstrate relatively similar performance, with no significant deviations observed between them.

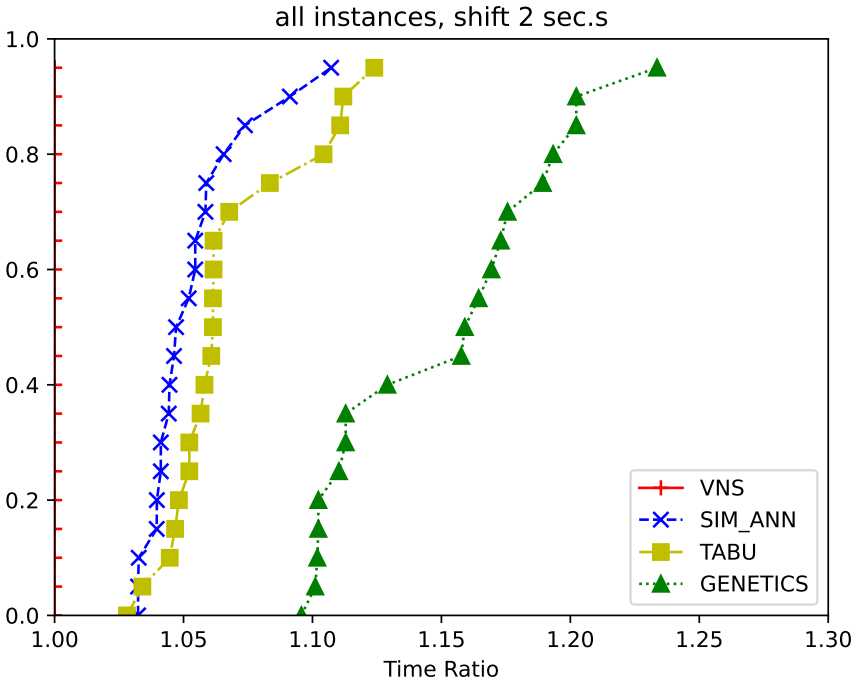


Figure 4.6: performance profiling metaheuristics

Chapter 5

Exact Algorithms

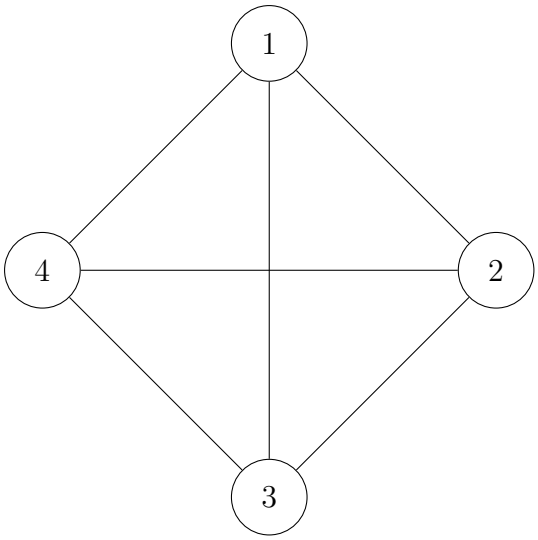
In this chapter, we will introduce algorithms that harness the power of the CPLEX Mixed-Integer Programming (MIP) solver to achieve optimal solutions to the problem at hand. The specific algorithms we will present comprise two techniques: the first being the Benders's loop. This technique gradually incorporates the subtour elimination constraint (SEC) into the basic TSP model, resulting in a feasible solution.
The second method revolves around the implementation of Branch and Cut. This approach employs a callback function that allows users to actively interact with the Branch-and-Cut (B&C) tree. This interaction facilitates the addition of cuts and customised modifications, improving the effectiveness of the algorithm.

5.1 Benders loop

The main concept of the Benders loop, introduced by J.F. Benders in 1962 [6], has its roots in the field of mathematical optimization and has since become a fundamental technique in solving complex optimization problems.
Not all, exponentially growing, Subtour Elimination Constraints (SEC) contribute significantly; most of them are useless and only adds to the complexity of the problem. The main objective of the Benders cycle is to address this exponential challenge.
In the Benders loop approach, we start by formulating the basic model of the TSP, which includes degree constraints. However, this initial model does not inherently produce a feasible solution. To overcome this problem, we exploit the loop methodology to incorporate SEC constraints.
First, we use the MIP CPLEX solver to solve the degree constrained problem.
Then, through an iterative loop, we examine the eventual subtours. For every identified subtour, we incorporate the corresponding subtour elimination constraints (SECs) into the model, resulting in a significantly reduced number of SECs within the formulation. After these improvements, we optimise the modified problem.
This iterative process continues as we systematically identify and correct subtours, leading to the derivation of new feasible solutions at each iteration, until the time limit is reached.
The number of constraints added is far less than exponential: for each subtour found we add the following constraint:

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

wher S_c represents a subtour for one of the non connected componenets.



SEC: $x_{12} + x_{23} + x_{34} + x_{41} + x_{13} + x_{24} \leq 3$

This method can be powerful for instances up to 400 nodes and has greatly imporved since CPLEX solver is now faster and omptimized.

5.1.1 Improvements with patching heuristic

One method to enhance the efficiency of Benders loop is by incorporating a patching heuristic.
This heuristic can be applied after each iteration, facilitating the creation of a feasible solution.
By implementing a repairing function, it seeks to improve the upperbound, resulting in earlier pruning of certain nodes in the Branch & Cut tree.
The repairing function progressively reduces the number of components by 1. At each iteration, the repair function identifies the most favorable pair of arcs, connecting different components, and merges them into a new component.
This iterative process continues until only one component remains, resulting in a feasible solution.
To further optimize the solution, we can utilize a local optimization technique such as two-opt. By applying the two-opt algorithm, we aim to enhance the feasibility of the solution, ultimately achieving an improved outcome.

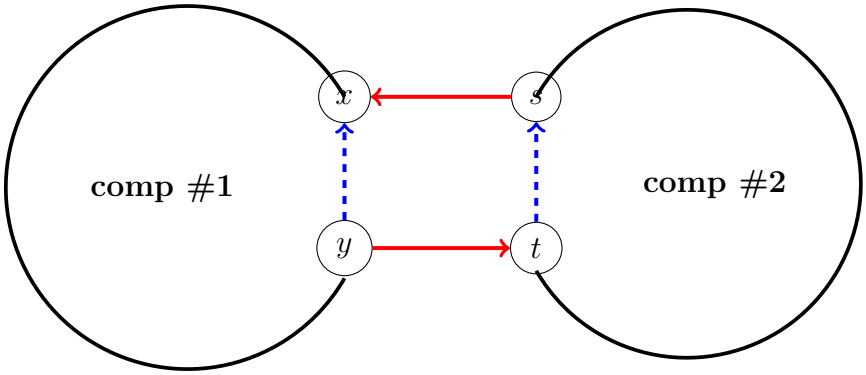


Figure 5.1: how patching heuristics connects components

Algorithm 8 Benders loop

Require: Instance, time_limit
Ensure: a valid tsp tour
    //set up initial CPLEX model with degree constraints
    while time_limit not reached **do**
        call CPXmiopt to optimize model
        find connected components
        if #(components) = 1 **then**
            break
        else
            add SEC's for all components
        end if
    end while

5.2 Branch and Cut using Callbacks

As the number of cities increases, solving the TSP optimally becomes computationally demanding, as the number of constraints grows exponentially.

IBM CPLEX, employs the Branch-and-Cut method to address combinatorial optimization problems like the TSP. This method combines several techniques to systematically explore the solution space and tighten the problem's formulation as it progresses. In this way we can avoid a lot of useless constraints and formulate on the go the ones needed by the solver, significantly lowering the time needed to solve the problem.

Callbacks in CPLEX are user-defined functions that can be triggered at various points during the optimization process. In our implementation we used two particular callback functions, one triggered when a
The key components and concepts of the Branch-and-Cut method with callbacks are the follwing:

1. **Branching**: CPLEX starts with an initial relaxation of the TSP problem, which is typically solved using linear programming techniques. When fractional solutions (non-integer) are encountered, the Branch-and-Cut method branches the problem into multiple subproblems by selecting one of the fractional variables and creating two child nodes. This process continues recursively until integer solutions are found.

2. **Fractional cuts**: CPLEX adds cutting planes to the problem formulation to eliminate fractional solutions that do not correspond to feasible TSP tours. These cutting planes are generated dynamically during the optimization process and help tighten the relaxation.

3. **Integer Solution Callback**: In the context of the TSP, an Integer Solution Callback is particularly valuable. When CPLEX finds an integer solution (a feasible TSP tour), the callback can be used to record and potentially improve this solution.

4. **Lazy Constraints Callback**: In addition to the Integer Solution Callback, CPLEX supports Lazy Constraints Callback. This callback allows you to dynamically add constraints that eliminate unwanted solutions as they are encountered, reducing the search space.

5. **Termination Criteria**: You can specify termination criteria for the Branch-and-Cut method, such as a time limit, an optimality gap, or a maximum number of nodes explored.

IBM CPLEX's Branch-and-Cut method with callbacks for the TSP leverages the power of both mathematical programming and heuristic search. It systematically explores the solution space using integer programming techniques, while callbacks enable the customization of the optimization process. This approach often leads to the discovery of high-quality solutions, and it can be extended and tailored to handle various TSP variants and problem instances.

Algorithm 9 callback_solution

Require: Instance,env,lp
Ensure: a valid tsp tour
contexid ← CPX_CALLBACKCONTEXT_CANDIDATE or CPX_CALLBACKCONTEXT_RELAXATION
install callback function with both lazy constraints and user cuts
optimize the problem and get the solution

5.3 Comparison between exact methods

After implementing Benders with patching heuristics and Callback with both lazy and fractional cuts, we conducted performance profiling to compare their efficiency on 20 instances. Figure 5.2 shows that the clear winner is the Callback approach. As mentioned earlier, the Callback method outperforms Benders because it eliminates the need to call the optimizer at each iteration, which is the most time-consuming part of the Benders algorithm.
In contrast, the Callback approach dynamically adds the required constraints during the execution, resulting in superior speed. However, Benders remained competitive for instances with up to 250 nodes and exhibited good performance within a reasonable timeframe.

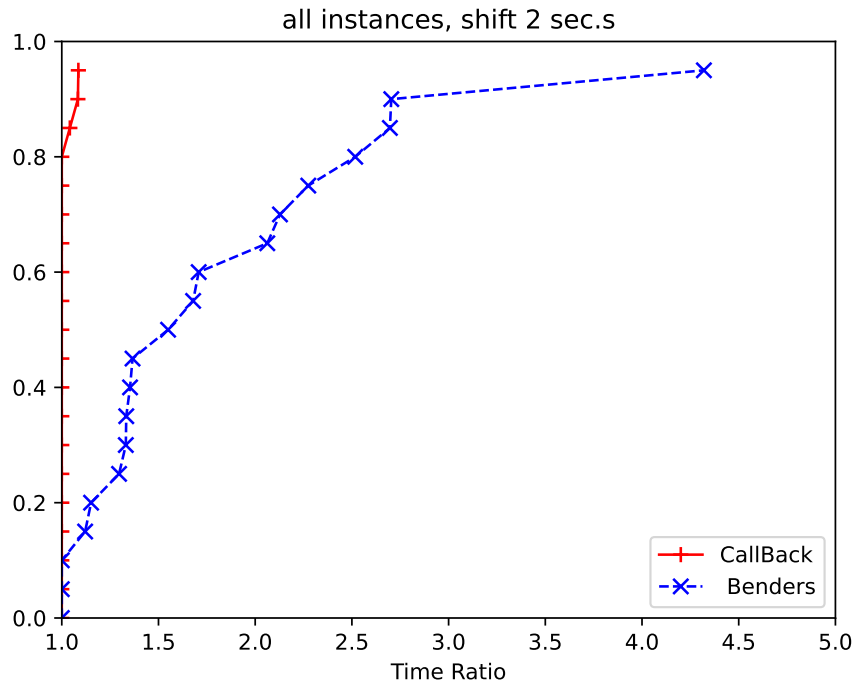


Figure 5.2: performance profiling for exact methods

Chapter 6

Matheuristic Algorithms

Matheuristic algorithms, introduced in the ealy 2000s, represent a dynamic fusion of mathematical programming and heuristic strategies.
While these algorithms incorporate mathematical models to address problems, they simultaneously incorporate heuristic elements. Although these heuristics may not guarantee optimization, they can enhance performance.
This chapter introduces two distinctive matheuristic algorithms that leverage distinct heuristic techniques to make the model easier, this enables MIP solvers to explore a more confined local space and yield a solution is less time.
The algorithms we're going to describe are hard fixing and local branching, each harnessing unique heuristic methods to simplify models and enhance problem-solving outcomes.

6.1 Hard Fixing

The term "Hard fixing" is derived from its distinctive feature of imposing strict constraints on a predetermined percentage of edges, preventing the swapping of those nodes.
This characteristic transforms the problem by simplifying it, as only a portion of the edges are allowed to be modified, resulting in a problem that is more manageable to solve.
The core principle behind this approach is to stabilize certain variables, thereby converting the original complex problem into a simplified version that can be efficiently solved using Mixed-Integer Programming (MIP) solvers.
This is particularly beneficial for handling larger-scale problems that might otherwise overwhelm exact methods like callback or Benders loop.
The algorithm starts by initializing a first solution. Subsequently, in an iterative manner, random variables x_{ij} are selectively set to 1, this adjustment involves altering the lower bound of the variable to 1. Following this, the modified problem is optimized using an MIP solver for a portion of the total time limit.
At the conclusion of each iteration, all variables are restored to their unfixed state so that in the next iteration we fix another random set of variables.
The crucial parameter in this algorithm is the initial variable percentage in fact it could be changed during the iterations.
This parameter embodies the trade off between creating a simpler integer problem that is easier to solve and constraining the local search space to a degree that might impede progress being to small.
Determining the optimal percentage involves extensive parameter tuning and performance profiling, with the following results: from the observed results both 0.4 and 0.5 were pretty

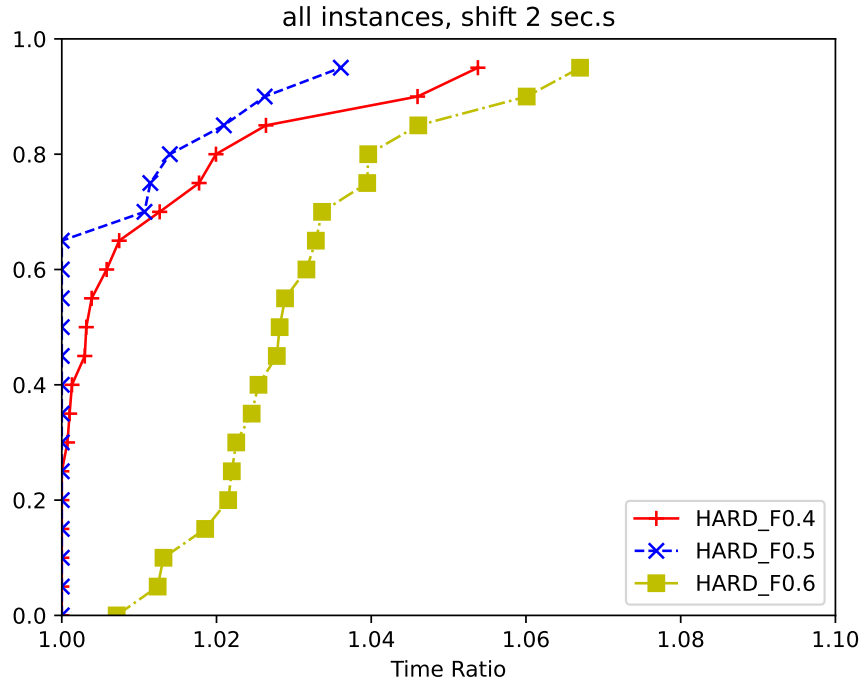


Figure 6.1: hardfixing prameter tuning

similar in performance but still 0.5 performed slighlty better so we opted for this value.

Algorithm 10 Hard Fixing

Require: Instance, time_limit
Ensure: a valid tsp tour
    //set up initial CPLEX model with degree constraints
    incumbent ← VNS(instance)
    p ← 0.5
    while time_limit not reached **do**
        fix some edges of the incumbent with probability p
        optimize model with benders or callback
        update instance.bestsol
        unfix all edges
    end while

6.2 Local branching

The local branching algorithm, first presented by M. Fischetti and A. Lodi [7], is similar to hard fixing which has as its primary goal to constrain the model by fixing certain constraints. However, in this case, we do not choose the restricted variables directly; instead, we attempt to restrict the search space.
This approach can address the challenges associated with hard fixing, which, due to its strong constraints, may require more time to achieve an improved solution.
Considering the solution as a series of variables, each taking on values of 0 or 1, we aim to restrict the model to search only within the solution space that has a certain distance from

the initial solution. In our case, the solution vector consists only of 1's and 0's, so we define this distance as the Hamming distance.
One way to think of the constraint in terms of the Hamming distance is to count the number of flipped 1's and the number of flipped 0's to a maximum value:

$$\sum_{e:x_e^j=1} 1 - x_e + \sum_{e:x_e^j=0} x_e \leq k$$

here, k represents the value of the local searching radius and the maximum hamming distance between solutions. Since the number of 1's fixed in a TSP solution is constant, the two quantities are equal. Thus, we can derive the simpler formulation where we just consider the number of flipped 1's:

$$\sum_{e:x_e^j=1} 1 - x_e \leq \frac{k}{2}$$

If we set the value of k to 2, we obtain a simple two-opt search. Due to the considerable computational power of CPLEX MIP solver, we can experiment with higher values for the local search radius.
In our implementation, we tested different values to determine the optimal parameter through performance profiling. Ultimately, we found that a parameter value of 40 yielded the best results. While the differences compared to other values were not significant, the value 40 consistently produced slightly improved outcomes, leading to its selection.

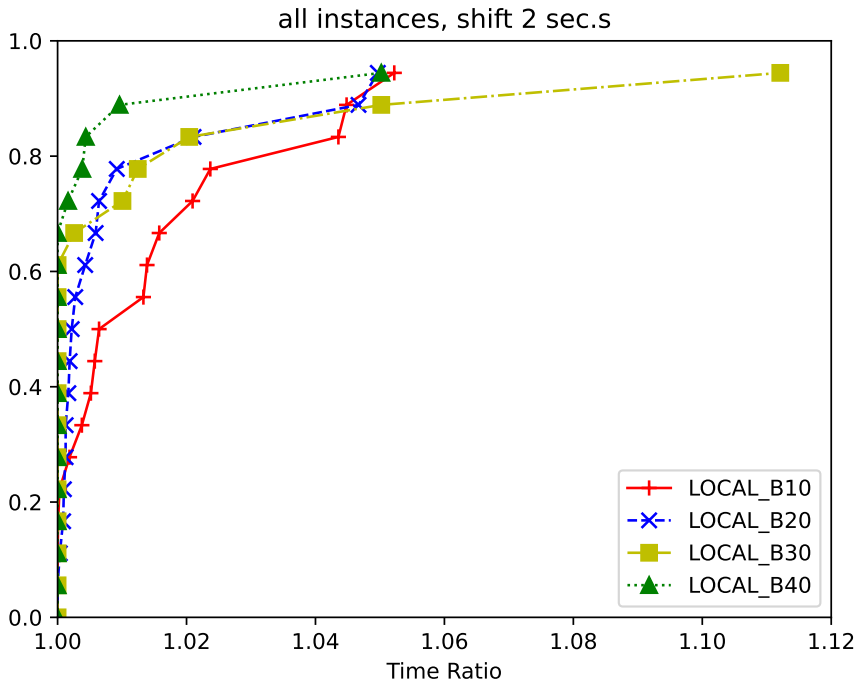


Figure 6.2: hardfixing prameter tuning

6.3 Comparison between matheuristics

We will now delve into a detailed comparison of the performance between hard fixing and local branching techniques.

Algorithm 11 Local Branching

Require: Instance, time_limit
Ensure: a valid tsp tour
    //set up initial CPLEX model with degree constraints
    incumbent ← VNS(instance)
    k ← 40
    while time_limit not reached **do**
        Add constraint to fix local search space
        optimize model with benders or callback
        update instance.bestsol
        eliminate the constraint
    end while

In Figure 6.3, we present a comprehensive performance profiling analysis that clearly demonstrates the superior efficacy of local branching over hard fixing.
Upon evaluation, it becomes evident that local branching outperforms hard fixing in terms of solution quality and flexibility.
Hard fixing, although useful in certain scenarios, tends to impose strong constraints on variables, potentially limiting the exploration of alternative solutions. In contrast, local branching provides the advantage of finding optimal solutions within a specific neighborhood, thereby offering greater freedom in the search process.

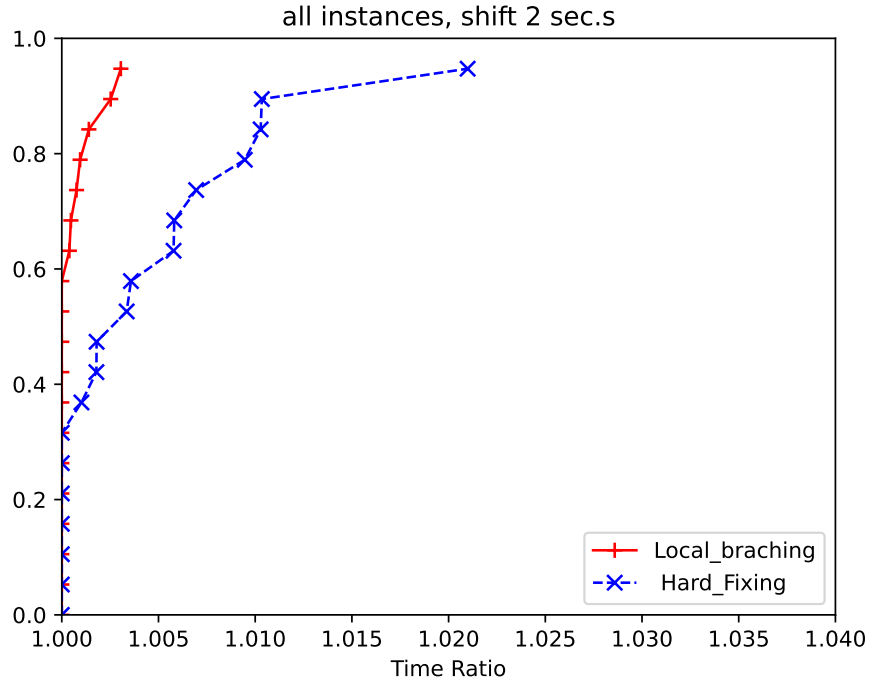


Figure 6.3: performance profiling for matheuristics

Bibliography

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

[2] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[3] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[4] F. Glover, "Tabu search—part i," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[5] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[6] J. F. Benders, "Partitioning procedures for solving mixed-variables programming problems," *Numerische Mathematik*, vol. 4, no. 1, pp. 238–252, 1962.

[7] M. Fischetti and A. Lodi, "Local branching," *Mathematical Programming*, vol. 98, pp. 23–47, September 1, 2003 2003.