UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Algorithms for Symmetric TSP

Operation Research

Master's Degree in Computer Engineering

Elnur Isgandarov (2014431)
Professor: Matteo Fischetti

2024

Table of Contents

List of Figures

1 Introduction: Symmetric TSP

1.1 Historical description

1.1.1 Traveling salesman

The origins of the travelling salesman problem (TSP) are unclear. A handbook for travelling salesmen [29] from 1832 mentions the problem and includes example tours through Germany and Switzerland, but contains no mathematical treatment.

Figure 1.1: Optimal route for a traveling salesman through the 15 largest cities in Germany. [29]

The TSP was first introduced in the realm of mathematics by Karl Menger, an Austrian mathematician, in a paper published in 1930. Menger's formulation aimed to find the shortest path that connects a given set of cities, with the additional constraint that each city must be visited exactly once and the journey must return to the starting city. [23]

During the mid-20th century, the TSP found itself at the heart of linear

programming, a major development in optimization theory. George Dantzig [9], Delbert Fulkerson [13], and Selmer Johnson [19] are prominent figures who applied linear programming techniques to TSP and played a pivotal role in advancing the field .

In the 1950s and 60s, as computing power increased, researchers began developing algorithms to tackle TSP instances of larger sizes. Computer scientists and mathematicians like Christofides [6] and Lin-Kernighan [21] were instrumental in developing heuristics and approximation algorithms for solving TSP instances efficiently.

The TSP has continued to evolve with the advent of new technologies and approaches. While exact algorithms like branch-and-bound are effective at finding optimal solutions for small to moderate-sized instances, their computational complexity often makes them impractical for handling increasingly larger instances. In contrast, metaheuristics have been developed not only to handle larger instances more efficiently but also to provide good-quality solutions for a wide range of problem sizes. Despite their focus on computational efficiency over optimality, these methods are valuable tools for tackling large-scale TSP instances. Researchers like David Applegate, Robert Bixby, Vasek Chvatal, and William Cook have made substantial contributions to the development and refinement of such approaches for solving TSP instances. [1]

1.2 Mathematical Description of the Problem

The TSP is a fundamental combinatorial optimization problem where the goal is to determine the shortest possible tour that visits a set of cities exactly once and returns to the starting city. This problem is classified as NP-hard, signifying that there is no known polynomial-time algorithm to find the optimal solution for all instances. To address this challenge, a variety of techniques, including heuristics like nearest neighbor, extra mileage, and metaheuristics such as simulated annealing, genetic algorithm, variable neighbourhood search, tabu search are employed to identify near-optimal solutions.

1.2.1 Graph Model

In the Symmetric TSP, we can represent the cities and their distances using a graph model. Each city is depicted as a node within a complete graph, where the edges between nodes signify the distances between the corresponding cities. The objective is to determine a Hamiltonian cycle, a tour that visits each city exactly once, with the minimum total distance.

Let $G = (V, E)$ denote the complete graph representing the cities, where:

- V is the set of vertices, with each vertex representing a city.
- E is the set of edges, where each edge denotes the distance between two cities.

The aim is to identify a permutation of cities that corresponds to a Hamiltonian cycle within the graph.

1.2.2 Linear Programming Model

The Symmetric TSP can be mathematically modeled as a linear programming problem. We introduce decision variables x_{ij} to indicate whether the tour includes the edge from city i to city j, and the objective is to minimize the total distance traveled.

Decision Variables:

$$x_{ij} = \begin{cases} 1, & \text{if the tour includes the edge from city } i \text{ to city } j \\ 0, & \text{otherwise} \end{cases} \tag{1.1}$$

Objective Function:

$$\text{Minimize:} \quad \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} d_{ij} \cdot x_{ij} \tag{1.2}$$

Constraints:

$$\sum_{i=1}^{n} x_{ij} = 1, \forall j \in C \qquad \text{(Each city must be visited exactly once)} \tag{1.3}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \forall i \in C \qquad \text{(Each city must be departed from exactly once)} \tag{1.4}$$

$$\sum_{i \in S} \sum_{j \in S, j \neq i} x_{ij} \leq |S| - 1, \qquad \text{(Subtour elimination constraints for all subsets)} \tag{1.5}$$

$$x_{ij} \in \{0, 1\}, \qquad \text{(Binary constraints)} \tag{1.6}$$

This description captures the core elements of the Symmetric TSP and its mathematical model, emphasizing the problem's symmetry, where the distance between city A and city B is equal to the distance between city B and city A.

2 Project Design

2.1 Objective

The primary objective of this project is to explore various algorithms and methodologies for efficiently solving the Traveling Salesman Problem (TSP). The project encompasses a wide range of techniques including constructive heuristics, metaheuristics, matheuristics, and exact methods from CPLEX.

2.2 Methodology

The project follows a systematic methodology for implementing and analyzing the performance of each algorithm. This involves:

1. Implementation of each algorithm using appropriate programming paradigms and libraries, primarily utilizing the C language for algorithms and Python scripts for analysis.

2. Testing the algorithms on randomly generated graphs representing TSP instances, with varying sizes and characteristics.

3. Analyzing the computational performance, solution quality, and convergence behavior of algorithms using Python scripts for cost plotting (only for VNS) and performance profiling. [10]

4. Utilizing shell scripts for managing file and folder operations, including logging, extracting information, and creating CSV files for performance profiling.

5. Utilizing GNU Plot for visualizing the resulting tour if provided through the command line.

6. Organizing files and folders in a structured manner, with each group of algorithms grouped together in separate files, and header files in an include folder while C files reside in the src folder. A Makefile is used for compilation.

7. Conducting experiments in the perf_prof folder, with subfolders named after algorithm types and further subfolders named after specific algorithms if hyperparameter comparisons are necessary.

8. Using a main file to execute the project with configurable parameters defined in a CONFIG data structure. Additionally, an instance data structure is used to store information about each instance.

9. Interfacing with the CPLEX software through the IBM ILOG CPLEX Optimization Studio, version 22.1.1, C library, enabling interaction with CPLEX solvers for solving TSP instances and integrating solver algorithms into the project [32].

10. Documenting implemented algorithms with Doxygen. [31]

3 Heuristics

3.1 Constructive heuristics

Constructive heuristics are forward-thinking algorithms that incrementally build solutions by adding elements or components. Employing a greedy strategy, these algorithms make locally optimal choices at each step, starting with an empty or partial solution. Despite not guaranteeing optimality, they address computational challenges in finding exact solutions, making them valuable tools in operations research. The three subsections detail implemented heuristics: the Nearest Neighbor (a greedy heuristic), GRASP (randomized), and the Extra-Mileage Heuristic.

3.1.1 Nearest neighbor (greedy heuristic)

The Nearest Neighbor algorithm, a fundamental greedy heuristic, constructs solutions by iteratively selecting the most proximate unvisited element at each step. Applied to combinatorial optimization problems, particularly the traveling salesman problem, this approach efficiently builds a solution by prioritizing locally optimal choices. By iteratively choosing the nearest unvisited element, the algorithm demonstrates simplicity and speed in finding reasonably good solutions. It can have 3 starting modes, and can be specified as either initiating from node 0, selecting a random node, or exhaustively evaluating all nodes to determine the optimal starting point.

The pseudocode for the computation is shown in Algorithm 1.

3.1.2 Greedy heuristic with GRASP

The Greedy Heuristic with GRASP (Greedy Randomized Adaptive Search Procedure) is introduced as an extension and refinement of the basic greedy approach. GRASP injects randomness and adaptability into the algorithm, improving its capacity to navigate beyond local optima and explore diverse solution spaces. Our application further refines this approach by dynamically selecting among the best, second-best, or third-best options with probabilities of 40%, 30%, and 30%, respectively. This tailored strategy expands the algorithm's versatility in

addressing combinatorial optimization challenges.

The pseudocode for the computation is shown in Algorithm 1.

Algorithm 1: Greedy heuristic algorithm

Input : instance, starting_mode, grasp_flag

Output: tour

```
Initialize the starting node with the given mode
while |uncovered_nodes| ≠ 0 do
    next_node ← greedy_step(grasp_flag)
    add next_node to tour
    remove next_node from uncovered_nodes
end
calculate_tour_cost(instance)
return tour
```

Note: 'instance' refers to a structured dataset containing pertinent information about the problem at hand. The 'starting_mode' parameter denotes a flag dictating the method for selecting the initial node. Lastly, the 'grasp_flag' parameter determines the node selection criteria for the 'greedy_step' function."

3.1.3 Insertion Heuristic (Extra-mileage)

The Insertion Heuristic initiates its process by forming an initial subtour from an arbitrary starting point and progressively incorporates nodes until a complete tour is established. During each iteration, the heuristic strategically selects an edge e and a node v that minimizes the additional mileage introduced by the path $< e_1, v, e_2 >$, considering all edges within the current subtour (E'). Formally expressed as:

$$\text{argmin}\{c(e_1, v) + c(e_2, v) - c(e_1, e_2) : e = \{e_1, e_2\} \in E' \wedge v \in V\} \tag{3.1}$$

This optimization guides the construction of the evolving tour. Our implementation offers three distinctive variants based on different initial subtours: the edge between the farthest nodes, the random edge and the best edge in the con-

vex hull of the nodes. For the details on the convex hull, please see Appendix D.

The pseudocode for the computation is shown in Algorithm 7.3.

Algorithm 2: Extra mileage heuristic

Input : instance, starting_mode

Output: tour

```
1 Initialize the starting node couple with the given mode
2 while |uncovered_nodes| ≠ 0 do
3 |   next_node ← extra_mileage_step()
4 |   add next_node to tour
5 |   decrease |uncovered_nodes|
6 end
7 calculate_tour_cost(instance)
8 return tour
```

Note: 'instance' refers to a structured dataset containing pertinent information about the problem at hand. The 'starting_mode' parameter denotes a flag dictating the method for selecting the initial node couple. Lastly, the 'extra_mileage_step' function exhaustively search for all edges in the given incomplete tour and uncovered nodes and return the one with the smallest delta cost as showed in 3.1"

3.2 Refinement Heuristics

3.2.1 2-OPT Refining

The 2-OPT heuristic method, given the tour which is found by constructive or randomly initialized, aims to decrease the total length of the route/tour through edge swaps. Croes introduced a two-optimization (2-opt) algorithm to solve the TSP case [7]. This algorithm identifies and rearranges crossing routes so that they do not intersect. An illustration of this algorithm can be seen in Fig. 3.1, where $d(\cdot, \cdot)$ is the distance metric. If the conditions in Eq. (3.2) are met, then the A and B paths are replaced with C and D paths to eliminate cross paths.

$$d(A, B) + d(C, D) < d(A, D) + d(B, C) \tag{3.2}$$

The 2-opt algorithm, proposed by Croes in 1958 [7], reorders routes that cross over themselves. This technique is applicable not only to the traveling salesman problem but also to related problems such as the vehicle routing problem (VRP) and the capacitated VRP.

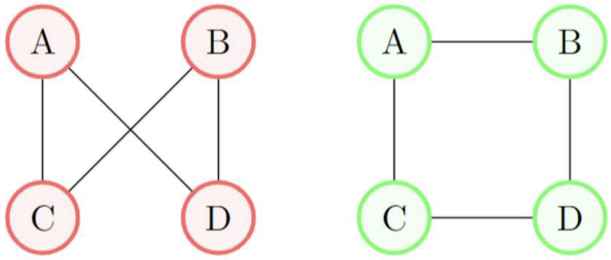


Figure 3.1: Illustration of the 2-OPT algorithm. Left: Initial tour with crossing edges. Right: Tour after applying the 2-OPT algorithm. [16]

3.2.2 Patching Heuristic

The Patching Heuristic enhances feasibility and improves the convergence of solutions in the Traveling Salesman Problem (TSP) when using Exact methods. This heuristic iteratively combines connected components. During the patching process, the heuristic ensures the total cost remains low by evaluating and choosing connections that add the least cost, maintaining a near-optimal path as components merge into a single tour.

Adding Subtour Elimination Constraints (SECs) for all components, including intermediate ones, significantly enhances the speed of convergence towards the optimal solution. This ensures that even under time constraints, a feasible solution is maintained, improving the overall efficiency of the algorithm. We have implemented the Patching Heuristic in Exact methods for solving TSP, as detailed in Section 5.

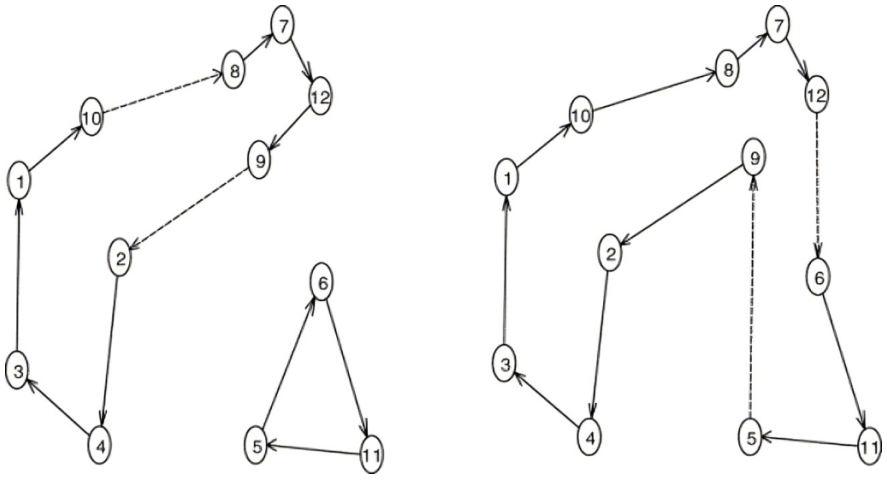


Figure 3.2: Demonstration of the Patching Heuristic for asymmetric TSP.[28] Note that symmetric TSP is a special case of asymmetric TSP.

4 Metaheuristics

A metaheuristic is a high-level algorithmic framework that provides guidelines for developing heuristic optimization algorithms. Coined by Fred Glover in 1986, the term combines the Greek prefix 'meta-' ($\mu\epsilon\tau\alpha$, beyond) with 'heuristic' (from the Greek 'heuriskein' or $\epsilon\upsilon\rho\iota\sigma\kappa\epsilon\iota\nu$, to search) [27].

Metaheuristics, as described by Luke in "Essentials of Metaheuristics," are a diverse set of algorithms applied in stochastic optimization. They offer problem-independent approaches to finding solutions that approximate the optimal solution within a reasonable computation time. Particularly suited for "I know it when I see it" problems, where heuristic information is limited, and brute-force search is infeasible due to the vast solution space [22].

In our approach, we define the neighborhood of a solution as the set of solutions obtained by replacing a certain number of edges, known as the kth-neighborhood [27]. The search for the optimal solution within a neighborhood, termed the intensification phase, often leads to a local optimum. Escaping local optimality is crucial, leading metaheuristic algorithms to alternate between intensification and diversification phases to explore unvisited solutions.

Employed in stochastic optimization, metaheuristics utilize randomness to find optimal or near-optimal solutions to complex problems. They are applied to scenarios where the optimal solution is unknown, and brute-force search is impractical due to the large solution space. Instead, metaheuristics rely on testing and assessing candidate solutions to identify promising solutions when encountered.

4.1 Tabu Search

Tabu Search (TS), a metaheuristic algorithm designed to evade local optima, employs a tabu list—a dynamic record of forbidden moves, guided by a critical hyperparameter: the tabu tenure length. Introduced by Glover in 1959 [14], TS combines local search methods with memory structures called tabu-list. By preventing revisits to recently explored solutions, it effectively navigates away from local minima, enhancing the quest for a global optimum. See Fig. 4.1 for an illustrative example.

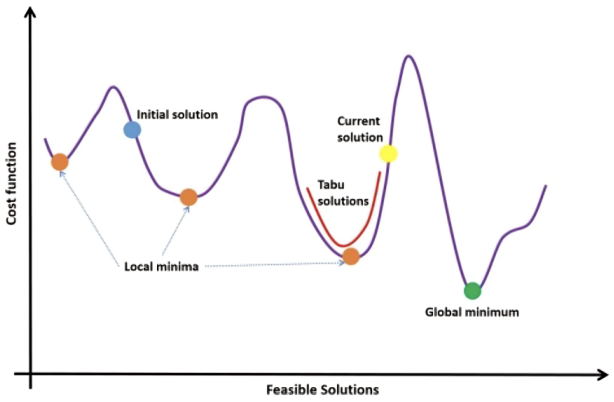


Figure 4.1: Illustration of the solution space of the Tabu search algorithm. [4]

The performance of this algorithm depends dramatically on the tenure of the tabu 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. Instead, if the tenure is too large, the search is not effective because the neighborhood is too small. To overcome this, we can change the tenure dynamically during the execution. In our experimentation, we explored three distinct approaches for determining the tabu tenure length:

- **Line Tenure**: In this mode, the tabu tenure length is adjusted linearly based on certain criteria, such as the number of iterations or the performance of the algorithm.
- **Step Tenure**: Here, the tabu tenure length changes in discrete steps at predefined intervals during the execution of the algorithm.
- **Random Tenure**: In this mode, the tabu tenure length is randomly adjusted within a predefined range during the execution, providing a degree of unpredictability to the algorithm.

We introduced an aspiration criterion alongside the tabu list: moves within the tabu list that potentially offer an enhancement to the current incumbent solution (the best encountered so far) are permitted. To manage the bounds of the tabu tenure length, upper and lower limits are established based on the number of nodes in the problem instance. Specifically, the upper bound is set at

one-tenth of the total number of nodes, capped at 100, while the lower bound is one-fifth of the upper bound. Our tabu search framework operates on a node tabu strategy, incrementally adding nodes and restricting all moves involving the affected node. The application of a 2-opt refinement remains consistent across iterations, commencing from an initial heuristic solution. We categorize the tabu tenure length: when it's short, it's akin to the intensification phase, whereas extending the tabu tenure length signifies the engagement of the diversification phase. Regarding the removal of nodes from the tabu list, any node listed as tabu is eliminated if the number of iterations since its addition exceeds the tabu tenure length.

The pseudocode for the computation is shown in Algorithm 3.

Algorithm 3: Tabu Search

Input : instance, tenure_mode, aspiration_criteria
Output: tour

```
best_solution ← heuristic_solution
while time_limit not expired do
    tenure_length_update(tenure_mode)
    for nodes not in tabu_list do
        two_opt(aspiration_criteria)
    end
    if curr_solution_cost < best_solution_cost then
        best_solution ← curr_solution
    end
end
return best_solution
```

Note: 'instance' refers to a structured dataset containing pertinent information about the problem at hand. The 'tenure_mode' parameter denotes a flag dictating the policy for updating the tenure length. Lastly, the 'aspiration_criteria' is a flag if aspiration criteria is applied during two_opt()

4.2 Variable Neighbourhood Search (VNS)

Introduced in a seminal paper [25], Variable Neighborhood Search (VNS) represents a powerful metaheuristic approach that significantly advanced the field of combinatorial optimization. The fundamental concept behind VNS involves shifting between different neighborhoods while searching for improved solutions. VNS operates through a descent approach towards a local minimum, then systematically or randomly explores neighborhoods progressively farther from this solution. During each iteration, one or more points within the current neighborhood serve as starting solutions for local descent. The algorithm transitions to a new solution only if a superior solution is identified.

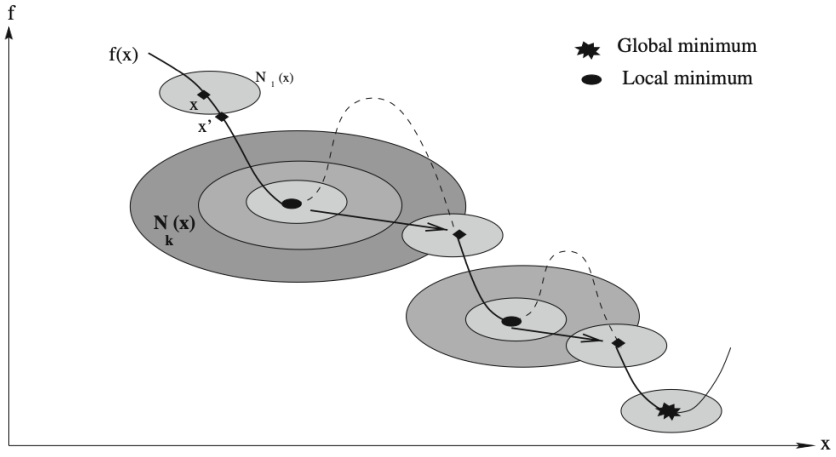


Figure 4.2: Illustration of VNS neighbourhoods in solution space [17]

In contrast to numerous other metaheuristics, the core methodologies of VNS and its extensions are straightforward and often require minimal, or even no, parameterization. This simplicity not only yields excellent solutions, often through more streamlined processes compared to alternative methods, but also facilitates a deeper understanding of its performance, consequently paving the way for more efficient and sophisticated implementations. Despite its apparent simplicity, VNS demonstrates remarkable effectiveness. It systematically leverages the following insights:

1. The existence of a local minimum within one neighborhood structure does not guarantee its existence within another.

2. A global minimum represents a local minimum across all conceivable neighborhood structures.

3. In many cases, local minima within one or more neighborhoods exhibit proximity to each other.

This latter observation, based on empirical evidence, suggests that a local optimum often offers valuable clues about the global one. Shared variable values between them, for instance, hint at potential overlap. However, identifying such variables remains a challenge. Consequently, a structured exploration of the neighborhoods surrounding this local optimum is conducted sequentially until a superior solution is discovered. [26]

In our implementation of Variable Neighborhood Search (VNS), we have adopted a practical approach to enhance the algorithm's exploration capabilities. One notable aspect of our implementation is the utilization of a parameterization strategy known as 'kicks.' This strategy allows us to vary the size of neighborhoods explored by VNS, with the parameter ranging from 3 to one-third of the instance length. By implementing this approach, we aim to facilitate more extensive exploration of solution spaces, thereby increasing the algorithm's effectiveness in finding high-quality solutions to combinatorial optimization problems.

The pseudocode for the computation is shown in Algorithm 4.

Algorithm 4: Variable Neighborhood Search

```
Input  : instance, kick_neighborhood
Output: tour
best_solution ← heuristic_solution
while time_limit not expired do
    n_opt_kick(kick_neighborhood)
    two_opt()
    if curr_solution_cost < best_solution_cost then
        best_solution ← curr_solution
    end
end
return best_solution
```

Note: 'instance' refers to a structured input dataset information.

4.3 Simulated Annealing

Simulated Annealing (SA) represents a widely used metaheuristic algorithm for solving combinatorial optimization problems. Invented in the early 1980s by Scott Kirkpatrick, Daniel Gelatt, and Mario Vecchi while they were working at IBM's Thomas J. Watson Research Center, Simulated Annealing was originally developed as a method inspired by the annealing process in metallurgy for solving optimization problems. The seminal paper introducing Simulated Annealing was published in 1983 [20].

The fundamental concept behind Simulated Annealing is to mimic the annealing process in metallurgy, where a material is gradually cooled to reach a low-energy state. Similarly, in Simulated Annealing, the algorithm starts with an initial solution and iteratively explores neighboring solutions. At each iteration, the algorithm evaluates whether the neighboring solution is better than the current solution. If it is, the neighboring solution becomes the new current solution. If the neighboring solution is worse, the algorithm may still accept it probabilistically, based on the difference in objective function values and a temperature parameter.

The temperature parameter controls the level of randomness in the search process. Initially set to a high value, it allows the algorithm to accept worse solutions with higher probability, enabling exploration of the solution space. As the algorithm progresses, the temperature gradually decreases according to a cooling schedule, reducing the probability of accepting worse solutions and shifting towards exploitation of promising regions [24]. In our implementation, we utilize the Metropolis formula to determine the acceptance probability of new solutions during Simulated Annealing. This formula considers the difference in cost between the current solution and its neighboring solutions. The temperature parameter is initialized with the solution cost obtained from the heuristic method, while the scaling factor, representing the average change in cost between adjacent edges in the heuristic solution, is determined beforehand. We express this formula as Equation 4.1. It's worth noting that we apply the heuristic method prior to initiating Simulated Annealing.

$$P(\Delta cost, Temprature, Scaler) = e^{-\frac{\Delta cost/Scaler}{Temprature}} \tag{4.1}$$

In Simulated Annealing, the process involves gradually cooling down the system, reheating it, and then cooling it down again for multiple iterations. In our implementation, we've added support for exploring different cooling schedules, known as annealing iterations. This feature enhances adaptability and tuning, allowing for flexibility in optimizing the algorithm's performance by trying out various cooling strategies.

The pseudocode for the computation is shown in Algorithm 5.

Algorithm 5: Simulated Annealing

Input : instance, annealing_iterations
Output: tour

```
1  best_solution ← heuristic_solution
2  while time_limit not expired do
3      for i = 1 to annealing_iterations do
4          for T = T_max to 1 do
5              a ← random_node()
6              b ← random_node_after_a_and_not_its_succ()
7              delta_cost ← delta_cost(a, b)
8              if random01() ≤ metropolis_formula(delta_cost, T, scaler)
                then
9                  update_tour(a, b, curr_solution)
10             end
11         end
12     end
13     two_opt()
14     if curr_solution_cost < best_solution_cost then
15         best_solution ← curr_solution
16     end
17 end
18 return best_solution
```

Note: 'instance' refers to a structured input dataset information. Function 'random01()' returns a random number between 0 and 1. Function 'update_tour()' updates a tour by swapping the positions 'a' and 'b'.

4.4 Genetic Algorithm

Genetic Algorithms (GA) are optimization techniques inspired by the principles of natural selection and genetics. They were first introduced by John Holland in the 1960s and further developed by Holland and his colleagues at the University of Michigan. The seminal paper introducing Genetic Algorithms was published in 1975 [18].

Genetic Algorithms operate on principles of natural selection, crossover, and mutation. They begin with a population of individuals or chromosomes, each representing a candidate solution. Evaluation via a fitness function determines reproduction likelihood, favoring individuals with higher fitness values. During reproduction, individuals are selected to mate and produce offspring through crossover and mutation operations. Crossover involves exchanging genetic information between two parent individuals to create new offspring with traits inherited from both parents. Mutation introduces random changes to individual chromosomes to explore new areas of the solution space.

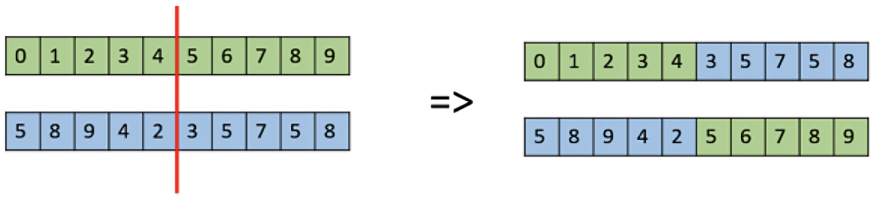


Figure 4.3: Crossover from the center point.[34]

This process of selection, crossover, and mutation iterates over multiple generations, gradually improving the quality of solutions in the population. Through iterative evolution and selection pressure, Genetic Algorithms efficiently explore large solution spaces.

In our implementation of GA for solving the Traveling Salesman Problem (TSP), we adopted a chromosome representation where each gene represents a node, and the chromosome represents a tour. The fitness of each individual is determined by the cost of the corresponding tour, with lower costs indicating more favorable solutions. To promote genetic diversity and explore different solution spaces, we implemented a single-point crossover operator with three selectable modes: middle, one-fourth, and random position.

Additionally, we incorporated three repair modes to address solution validity for the new offspring:

- Without repair mode: This mode punishes the bad genes in terms of their quality by increasing the cost artificially.
- Repair mode: This mode removes duplicate visits and adds missing nodes to ensure solution validity.
- Full-repair mode: This mode applies repair and 2-opt refining to improve solution quality.

In our implementation, regardless of the repair mode chosen (with or without repair), we ensure the champion individual selected through natural selection always undergoes repair to maintain solution validity. Additionally, alongside repair, we apply 2-opt refining to further improve the quality of solutions. This approach aims to address the potential scenario where the evolutionary process may take a significant amount of time to evolve to a valid TSP tour. By applying repair and 2-opt refining to the champion individual, we guarantee a valid solution while continuously improving its quality.

The pseudocode for the computation is shown in Algorithm 6.

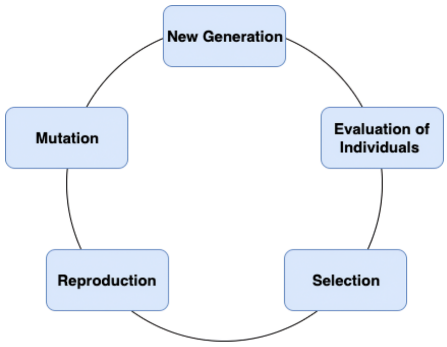


Figure 4.4: Genetic algorithm cycle.

Algorithm 6: Genetic Algorithm

Input : instance, repair_mode, cutting_type

Output: tour

```
1  population ← initialize_random_population()
2  while time_limit not expired do
3  |   children ← crossover(population, cutting_type)
4  |   repair(children, repair_mode)
5  |   mutants ← mutate(population)
6  |   generation ← elitism(population, children, mutants)
7  |   population ← evolve(generation)
8  |   champion ← fittest_individual(population)
9  end
10 full_repair(champion)
11 if champion_solution_cost < best_solution_cost then
12 |   best_solution ← champion_solution
13 end
14 return best_solution
```

Note: 'instance' refers to a structured input dataset information. As champion of each generation always survives to the next one, after selection is done, champion of the last generation is also the selection champion.

5 Exact methods

Exact methods for solving the Traveling Salesman Problem (TSP) guarantee finding an optimal solution by exhaustively exploring the solution space. These methods, while computationally intensive, ensure that the solution is the best possible. One such method involves using the CPLEX Mixed-Integer Programming (MIP) solver, which effectively handles Subtour Elimination Constraints (SECs) to ensure that the solution constitutes a single valid tour.

The Dantzig-Fulkerson-Johnson (DFJ) model is a classical approach to solving the TSP that incorporates these Subtour Elimination Constraints. The primary goal of the DFJ model is to prevent subtours, or disjoint cycles, within the solution. This ensures that the solution is a single tour that visits all cities exactly once. The DFJ model is also known for its application of the handshaking lemma to enforce these constraints [8].

The handshaking lemma, a principle from graph theory, states that in any undirected graph, the sum of the degrees of all vertices is equal to twice the number of edges. Formally, for a graph $G = (V, E)$ with vertex set V and edge set E:

$$\sum_{v \in V} \deg(v) = 2|E| \tag{5.1}$$

In the context of the TSP and the DFJ model, the handshaking lemma is used to ensure that each city (node) is visited exactly once in a single tour (cycle). The constraints derived from the DFJ model leverage this principle to prevent the formation of subtours by ensuring that every node in the graph has an even degree, which implies that each city is entered and left exactly once.

The DFJ subtour elimination constraints can be formulated as follows:

$$\sum_{i \in S} \sum_{j \in S, j \neq i} x_{ij} \leq |S| - 1, \quad \forall S \subset V,\ 2 \leq |S| \leq |V| - 1 \tag{5.2}$$

where V represents the set of all nodes (cities), S is any subset of V, x_{ij} is a binary decision variable indicating whether the edge between node i and node j is included in the tour, and $|S|$ is the number of nodes in subset S. This constraint ensures that no subset of nodes forms a tour on its own, thereby enforcing a

single, valid tour in the solution.

By applying the DFJ model with the CPLEX MIP solver, we can accurately and efficiently solve the TSP while ensuring that the solution adheres to the necessary constraints to form a single valid tour.

5.1 Benders' loop

Benders' Loop method is an exact approach for solving the Traveling Salesman Problem (TSP) by iteratively adding Subtour Elimination Constraints (SECs) until the optimal solution is found. This method leverages Benders' decomposition, where the problem is divided into a master problem and a subproblem, iteratively refining the solution by adding necessary constraints to eliminate subtours [2].

To expedite the process, the CUTUP value is initialized using a greedy heuristic (*try_all* starting mode), which is further improved by a 2-opt technique, as detailed in Section 3.1.1. The CUTUP value, representing the cost of the initial solution, allows the algorithm to ignore solutions with higher costs, thus focusing on more promising regions of the search space.

The Benders' Loop method iterates by solving the relaxed master problem and checking for subtours. If subtours are detected, the corresponding SECs are added to the model and the problem is resolved. This loop continues until the optimal solution is found, indicated by the presence of only one connected component, or until a predefined time limit is reached. If a time limit is imposed, to provide a feasible solution, a Patching heuristic is applied. As detailed in Section 3.2.2, this also speeds up the convergence of the algorithm.

The pseudocode for the computation is shown in Algorithm 7.

Algorithm 7: Benders' loop

Input : instance

Output: tour

```
1  solution ← heuristic_solution(instance)
2  upper_bound ← solution_cost
3  repeat
4  |  cplex_solution ← CPLEX_solver(instance)
5  |  for subtour in cplex_solution do
6  |  |  add_subtour_constraints(subtour)
7  |  end
8  |  solution ← patching_heuristic(cplex_solution)
9  until time_limit expired or solution has no subtours;
10 return solution
```

Note: 'instance' refers to a structured input dataset information. For details of CPLEX solvers and which functions to use please see the guidelines in its website [32]

5.2 Callback method

Another method to use SEC is through CPLEX callbacks for a Branch and Cut implementation. This approach applies the constraints only when a candidate solution is found, a situation referred to as lazy constraints. Lazy constraints are constraints that should be part of the model but are not initially included, such as SEC. When adding SEC, we also use a patching heuristic (for details, see Section 3.2.2) and incorporate those constraints into the model.

The prototype of the callback function in CPLEX is fixed, but its name can be customized. It is important that this function is thread-safe, as CPLEX runs in multithreaded mode by default (this setting can be modified if needed). The patching heuristic is applied inside this function, and if it is better than the incumbent solution, we share this solution with CPLEX to speed up the process using **CPXcallbackpostheursoln**.

The definition of the callback function is as follows:

```
static int CPXPUBLIC my_cut_callback(CPXCALLBACKCONTEXTptr context,
CPXLONG contextid, void *userhandle)
```

To speed up the process, the CUTUP value from the heuristic solution is used, as in Benders' loop (details in Section 5.1), and in addition to that, a mipstart solution is added with **CPXaddmipstarts**. This is different from `PARAM_CUT`, as it ensures that CPLEX starts from this specific solution. This combination of lazy constraints and starting solutions helps to efficiently guide the solver towards optimal solutions.

The pseudocode for the computation is shown in Algorithm 8.

Algorithm 8: Callback method

Input : instance
Output: tour

```
solution ← heuristic_solution(instance)
upper_bound ← solution_cost
CPLEX_time_limit(instance_time_left)
solution_to_mipstart(solution)
cplex_solution ← CPLEX_solver(instance)
if cplex_solution has subtours then
    for each subtour in cplex_solution do
        add_subtour_constraints(subtour)
    end
    solution ← patching_heuristic(cplex_solution)
end
if cplex_solution_cost > solution_cost then
    post_heuristic(solution)
end
return cplex_solution
```

Note: 'instance' refers to a structured input dataset information. For details of CPLEX solvers and functions please see the guidelines in its website [32]

6 Matheuristics

Matheuristics, a fusion of "mathematical" and "heuristics," blend optimization techniques from mathematical programming with heuristic approaches. These methods have gained prominence through various contributions, including significant work by Berbeglia et al. [3]. Matheuristics harness the combined power of heuristic methods and exact algorithms (see Figure 6.1). This synergy allows for a rigorous mathematical description of problem structures while employing heuristic strategies to explore the solution space efficiently. Matheuristics have proven to be highly adaptable and effective in various fields such as logistics, manufacturing, and transportation, making them a go-to approach for solving complex optimization problems. For more details about matheuristics, see the work by Martina and Matteo Fischetti [11].

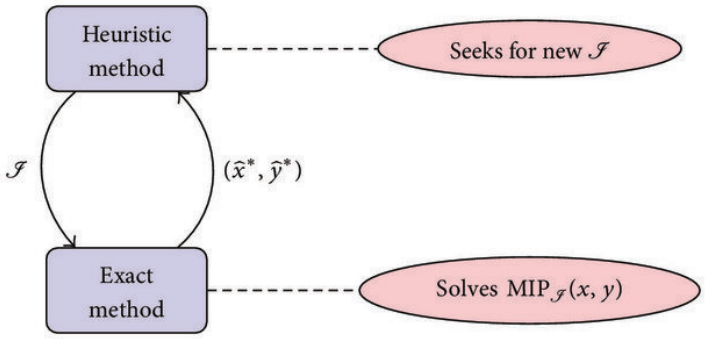


Figure 6.1: Matheuristics: Interaction between a heuristic method and an exact method. [5]

There can be situations where there is a time limit, and in such cases, it is preferred that an algorithm updates or improves the solution more significantly at the beginning, even if it takes longer to reach optimality. Algorithms are often evaluated based on their total time to reach optimality. However, sometimes they are evaluated using the primal integral instead. The goal of this evaluation is to reduce the area under the curve, ensuring that better solutions are found early in the process (see Figure 6.2).

In matheuristics, the primal integral is the main evaluation metric. Additionally, in CPLEX, there are methods to internally tune the model to reduce

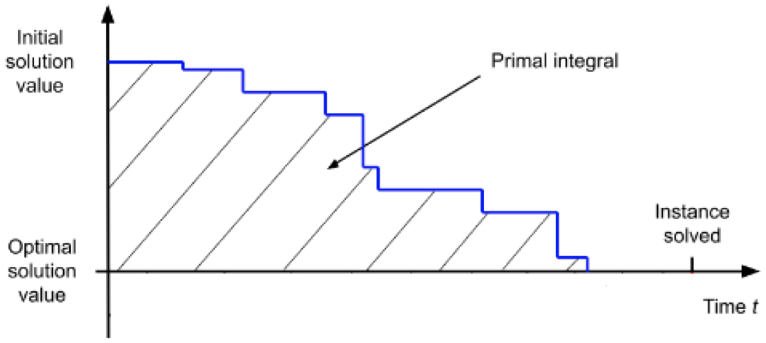


Figure 6.2: Primal integral. [30]

the total time to optimality. One such method involves setting the MIP emphasis value to 5, which aims to reduce the primal integral. This approach can be effective but is not always the best, as its performance can vary.

Another method is the application of the Relaxation Induced Neighbourhood Search (RINS) internal heuristic, which is very powerful. Using RINS frequently requires a significant amount of time, but this time is well spent because it improves the solution. By default, the RINS frequency is set to 0, which is well-tuned, but increasing this value can often be beneficial for the primal integral as it dedicates more time to the heuristic. Typically, a high frequency is applied initially and then gradually decreased.

The Polishing heuristic is another method, which is akin to a genetic algorithm based on mixed-integer models. It was developed because, at times, CPLEX provides a good result that can still be improved by simple heuristics like 2-OPT, which is not ideal for a commercial solver. Therefore, instead of allocating the entire time limit to the branch-and-cut method, we can allocate a large portion of the time to branch-and-cut and a smaller portion to the Polishing heuristic.

Since 2000, methods have been proposed to avoid modifying CPLEX and to use it as a black-box solver. For instance, if the optimal solution is known, fixing the edges in CPLEX that belong to the optimal solution is recommended; however, the optimal solution is typically unknown. Therefore, a feasible and heuristic-based solution is preferred. We have implemented two such matheuristic methods: Hard Fixing and Local Branching.

6.1 Hard-Fixing (Diving Heuristic)

Hard-Fixing, also known as the Diving heuristic, is an approach that involves iteratively fixing certain variables in the solution to expedite the optimization process. This method, which has been developed through contributions from various researchers in the field, begins with a solution computed by a heuristic or provided by CPLEX within a short time limit. Subsequently, some of the edges in the solution are randomly selected and fixed to 1 (i.e., $X_{ij} = 1$ where $X_{ij} \in \{0, 1\}$).

After fixing these values, CPLEX is called as a black-box solver to determine if a better solution can be found given the fixed values. This process of fixing and solving is repeated until the time limit is reached. Unfixing the variables involves resetting the lower bound of the edges back to 0. It is important to note that finding the same solution in multiple iterations is acceptable due to the inherent randomness of the policy; improvement is not guaranteed in every iteration.

Typically, significant improvements are seen at the beginning, especially if the initial solution is already of good quality. The incumbent solution can always be used as a MIP start to guide the solver. During the fixing phase, instead of completely random selection, one might consider fixing the shortest edges with a higher probability and longer edges with a lower probability. However, simple ideas like these often do not outperform a purely random fixing strategy.

In our implementation, we adopted the totally random version of this heuristic, with the probability of fixing edges (denoted as P_{fix}) being an input parameter. Notably, a higher P_{fix} means searching within a smaller neighborhood, which results in faster solutions. An adaptive P_{fix} strategy can also be employed by choice, starting with a large P_{fix} and gradually decreasing it, which accelerates the search process in the beginning.

The pseudocode for the computation is shown in Algorithm 9.

Algorithm 9: Hard-Fixing

Input : instance, probability_of_fixing

Output: tour

```
solution ← heuristic_solution(instance)
while time_limit not expired do
    fixing(instance, probability_of_fixing)
    cplex_solution ← branch_and_cut(instance)
    unfixing(instance)
    if cplex_solution_cost < solution_cost then
        solution ← cplex_solution
    end
end
return solution
```

Note: 'instance' refers to a structured input dataset. Note that `branch_and_cut` is the Callback method (see Section 5.2 for details) without applying the heuristic method. For information on CPLEX solvers and functions, please refer to the guidelines on its website [32].

6.2 Local Branching

Local Branching is a matheuristic technique introduced by Fischetti and Lodi [12]. It involves adding a constraint to the mathematical model, known as the local branching constraint, which restricts a certain number of variables without explicitly selecting which ones to fix. This technique is also referred to as "soft fixing."

Local Branching explores the neighborhood of the current solution within a specified radius, denoted as k. If the radius is zero, the solution remains unchanged; for $k = 2$, the solution must retain at least 98 out of 100 edges. Hence, the radius k acts as a hyperparameter defining the neighborhood size.

Given the current solution x^h, the local branching constraint is formulated as:

$$\sum_{(i,j):x_{ij}^h=1} x_{ij} \geq n - k \tag{6.1}$$

Here, x^h is a binary vector representing the presence of edges, with a length of $|E|$. A value of 1 means the edge is present, while a value of 0 means the edge is absent. The left-hand side of the equation represents the number of preserved edges. This constraint is asymmetric because it focuses on variables with a value of 1, making it particularly effective for problems like the Traveling Salesman Problem (TSP). However, in a more general context, fixing variables to 0 might also be important, leading to the more general symmetric form of the local branching constraint.

The Hamming distance measures the number of bit flips needed to transform one solution into another. For binary solutions x and x^h in $\{0,1\}^{|E|}$, the Hamming distance is defined as:

$$H(x, x^h) = \sum_{(i,j)\in E: x^h_{ij}=1} (1 - x_{ij}) + \sum_{(i,j)\in E: x^h_{ij}=0} x_{ij} \leq k' \tag{6.2}$$

The first sum represents flips from 1 to 0, and the second sum represents flips from 0 to 1. For a given x^h, the two sums are always equal, allowing the simplification:

$$\sum_{(i,j)\in E: x^h_{ij}=1} (1 - x_{ij}) = \sum_{(i,j)\in E: x^h_{ij}=0} x_{ij} \tag{6.3}$$

Thus, the local branching constraint can be rewritten as:

$$2 \sum_{(i,j)\in E: x^h_{ij}=1} (1 - x_{ij}) \leq k' \tag{6.4}$$

$$\sum_{(i,j): x^h_{ij}=1} x_{ij} \geq n - \frac{k'}{2} \tag{6.5}$$

Here, n is the number of edges equal to 1 in a TSP tour, which is equal to $|V|$, the number of vertices. The parameter k represents the number of edges of the incumbent solution that CPLEX is free to reconsider in the upcoming re-optimization of the problem.

Within the form in Equation 6.5, the constraint becomes sparser and more efficient, focusing on a smaller subset of variables.

We implemented a dynamic version of local branching. Starting with an initial value k_{initial}, this version increases k by k_{initial} each time no better solution is found, continuing until the time limit is reached. If a better solution is found

within the given time, the algorithm resets k to k_{initial} and repeats. Starting with a very small k, such as 2, resembles a 2-OPT local search, while a large k, such as $n/2$, might be excessive. Typically, a value between 10 and 30 is recommended.

The pseudocode for the computation is shown in Algorithm 10.

Algorithm 10: Local Branching

```
Input  : instance, branching_constraint
Output: tour
branching_constraint_initial ← branching_constraint
solution ← heuristic_solution(instance)
while time_limit not expired do
    add_local_branching_constraints(instance, branching_constraint)
    cplex_solution ← branch_and_cut(instance)
    remove_last_constraints(instance)
    if cplex_solution_cost > solution_cost then
        branching_constraint += branching_constraint_initial
    end
    else
        branching_constraint = branching_constraint_initial
        solution ← cplex_solution
    end
end
return solution
```

Note: 'instance' refers to a structured input dataset. Note that `branch_and_cut` is the Callback method (see Section 5.2 for details) without applying the heuristic method. For information on CPLEX solvers and functions, please refer to the guidelines on its website [32].

7 Experimental results

This section aims to compare different algorithms and their parameters if available for tuning. We use only the EUC_2D distance type. The instances are randomly generated, with x and y coordinates up to 10,000. Running algorithms on many different instances is a repetitive task, so we use shell scripts to automate this process. For more information about scripts, please see Appendix C. We log all the results and use shell scripts to extract the results into CSV files.

In evaluating the performance of optimization algorithms, both arithmetic average and geometric average are commonly used metrics. The arithmetic average is simple to calculate and interpret, providing a straightforward mean of the performance measures across different instances. However, it can be heavily influenced by outliers, where a few extremely high or low values can skew the results. The geometric average, on the other hand, mitigates the impact of outliers by considering the multiplicative relationships between performance measures. It is often preferred for comparing ratios or rates, as it gives a more balanced view when dealing with varying scales. Despite these advantages, both arithmetic and geometric averages have limitations. They reduce the performance across multiple problems to a single summary statistic, potentially obscuring important variations in algorithm performance across different problem instances. This is why performance profiling, which provides a comprehensive and intuitive way to assess algorithm performance across a set of problems, is often considered a more robust and informative method. [10]

Performance profiling provides a comprehensive and intuitive way to assess how well each algorithm performs across a set of benchmark problems. For each algorithm and problem, the performance ratio is computed, measuring the algorithm's performance relative to the best performance observed for that problem. The cumulative distribution function (CDF) for each algorithm is then plotted, showing the proportion of problems for which the algorithm's performance is within a given factor of the best performance. These CDF plots, where the x-axis represents the performance ratio and the y-axis represents the cumulative proportion of problems, allow for an easy comparison of the algorithms' efficiency and robustness, with higher curves indicating better overall performance.

We use two types of performance measures for the algorithms. For heuristics,

we use the cost ratio because they need to run to completion to provide a solution. For exact methods, we use the time of convergence since they run until they find the optimal values. For metaheuristics and matheuristics, we use a combination of time limits and cost ratios.

7.1 Heuristics

We used 50 instances with different seeds, each with a node size of 2000. When comparing algorithms based on the quality of the solutions they produce, it's important to note that those in this category must build a complete solution. Unlike algorithms in other categories, these cannot be constrained by a time limit because if a constructive algorithm fails, it would not yield any partial or current best solution.

Note that for the Greedy algorithm, starting mode 2 allows the user to try all nodes as starting points, effectively running the algorithm as many times as there are nodes in the graph. Since it remains very fast, we did not modify it or limit the number of nodes to try. Similarly, for the Insertion heuristic, starting mode 2 runs the algorithm multiple times, as it tries all node pairs in the convex hull as starting points. However, this drastically affects the runtime, so to keep the comparison fair, we decided to impose a time limit. Therefore, in our experiments, only one pair from the convex hull is used as the starting couple.

7.1.1 Greedy heuristic

We have three different starting modes for both the standard greedy algorithm and the GRASP version. It can clearly be observed from the plot that using the GRASP strategy does not directly improve the results; in fact, it often performs worse. Additionally, when using GRASP, all starting modes perform similarly without a clear winner. However, from the standard greedy algorithm's performances, we can see that starting modes provide some insights. Even though starting from a random node or just node 0 shows very similar results, trying all nodes as starting nodes can improve the performance of this algorithm.

The performance profiling is shown in Figure 7.1.

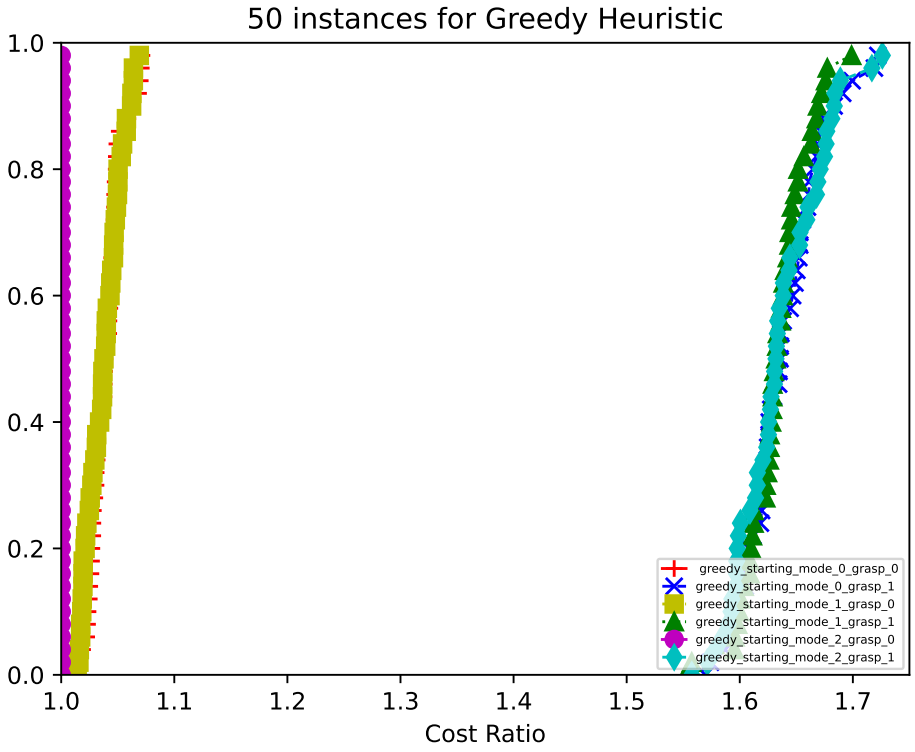


Figure 7.1: Greedy heuristic performance profile.

Applying 2-OPT refinement to the greedy algorithm significantly improves the results. For example, for seed 0, the result improves from 416941 to 343080. This refinement brings the lines closer together without changing their order from the previous figure. Although starting mode 2 is still slightly better than the other following two, the gap is not as pronounced as before.

The performance profiling is shown in Figure 7.2.

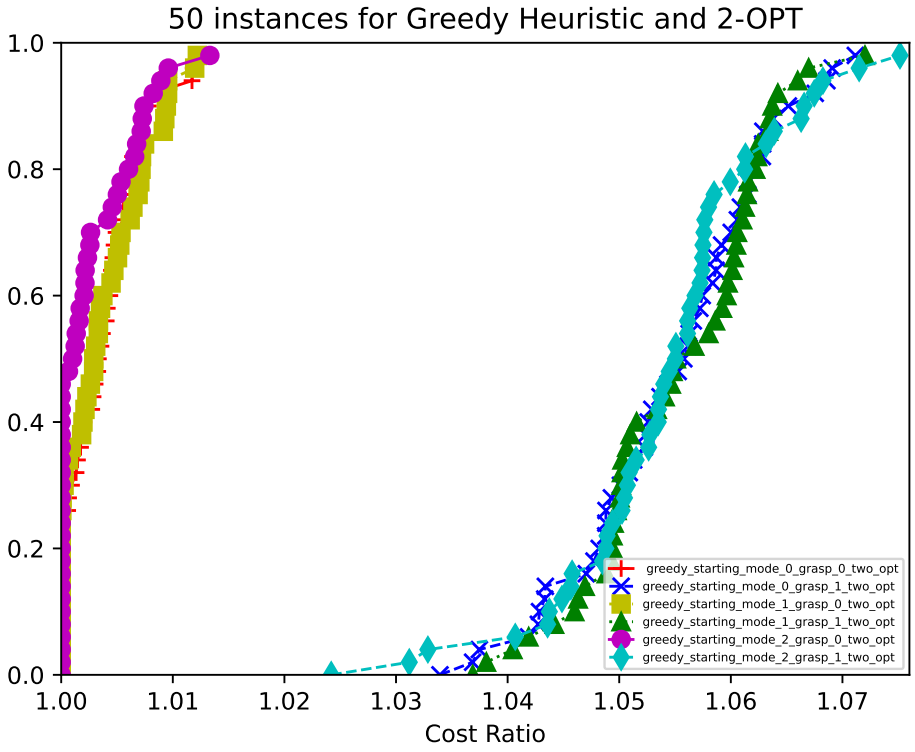


Figure 7.2: Greedy heuristic plus 2-OPT performance profile.

7.1.2 Insertion

For the Extra Mileage heuristic, we tested three different starting modes: starting from the edge between the farthest nodes, a random edge, and the edge in the convex hull of the nodes. Since the convex hull can contain many nodes, running the algorithm multiple times could be computationally expensive. Therefore, we applied a time limit to run the algorithm only once, meaning it just tries one node pair from the convex hull.

Despite not always choosing the best edge among those in the convex hull, starting mode 2 clearly performs better than starting mode 0 (starting from the farthest edge). Additionally, starting mode 2 mostly performs slightly better than starting mode 1 (starting from a random edge), although not consistently. Starting mode 1 performs slightly better than starting mode 0.

The performance profiling is shown in Figure 7.3.

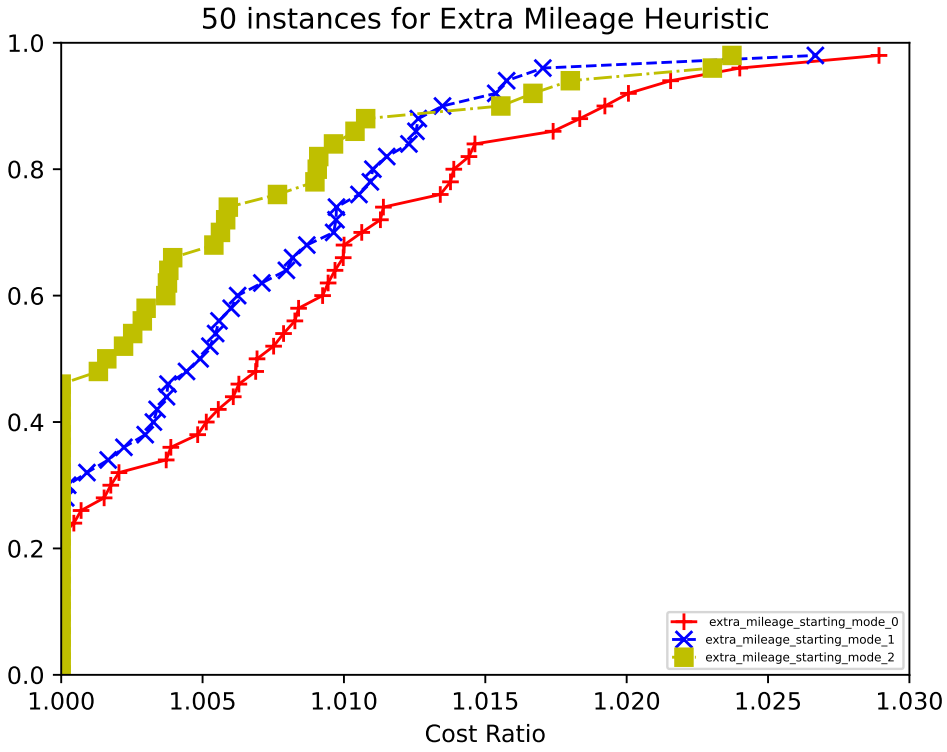


Figure 7.3: Extra mileage performance profile.

For the Extra Mileage heuristic combined with the 2-OPT optimization, we tested the same three starting modes: starting from the edge between the farthest nodes, a random edge, and the edge in the convex hull of the nodes. The addition of the 2-OPT optimization significantly improves performance across all starting modes. Notably, starting mode 0 (starting from the farthest edge) now performs better than starting mode 1 (starting from a random edge), reversing their previous standings without 2-OPT.

Starting mode 2 (starting from an edge in the convex hull) continues to generally perform the best among the three modes, benefiting greatly from the 2-OPT optimization. Even with the computational limitation of running the algorithm only once on a single node pair from the convex hull, this mode remains the most effective. In summary, combining the Extra Mileage heuristic with 2-OPT enhances solution quality across all modes, with starting mode 2 leading, followed by starting mode 0, and then starting mode 1.

The performance profiling is shown in Figure 7.4.

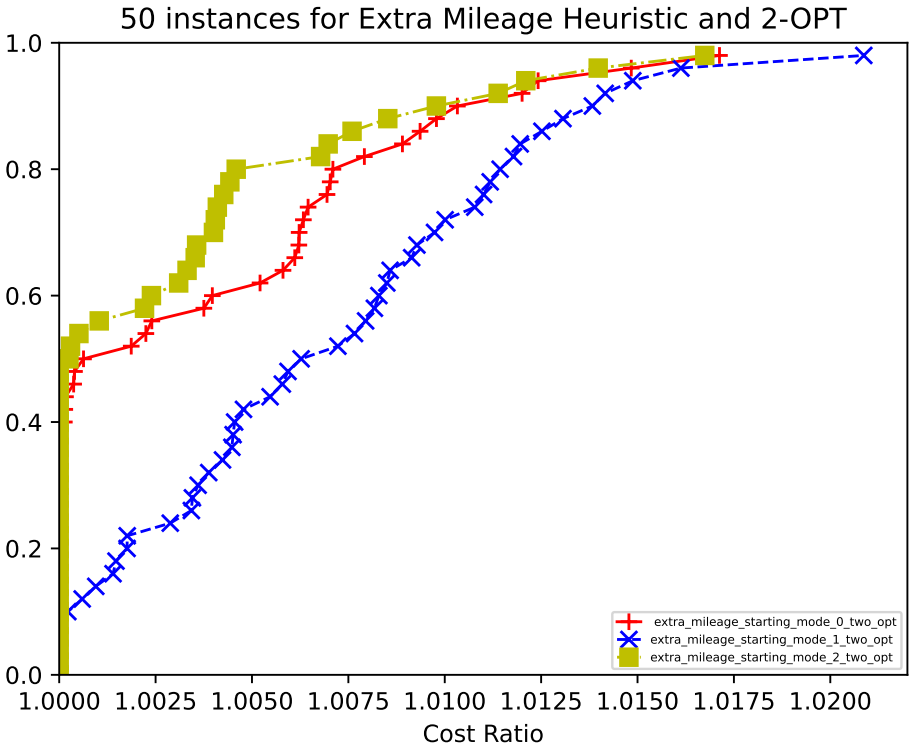


Figure 7.4: Extra mileage plus 2-OPT performance profile.

7.1.3 Best methods comparison

Among all the implemented heuristic methods, we selected one from each category to analyze the performance variance among them. It's evident from the plot that applying the 2-OPT operation significantly improves the results. Additionally, we can compare the Greedy and Insertion methods. Without applying 2-OPT, the Greedy algorithm performs slightly better. However, with the application of 2-OPT, the Greedy heuristic emerges as the clear winner among all.

The performance profiling is shown in Figure 7.5.

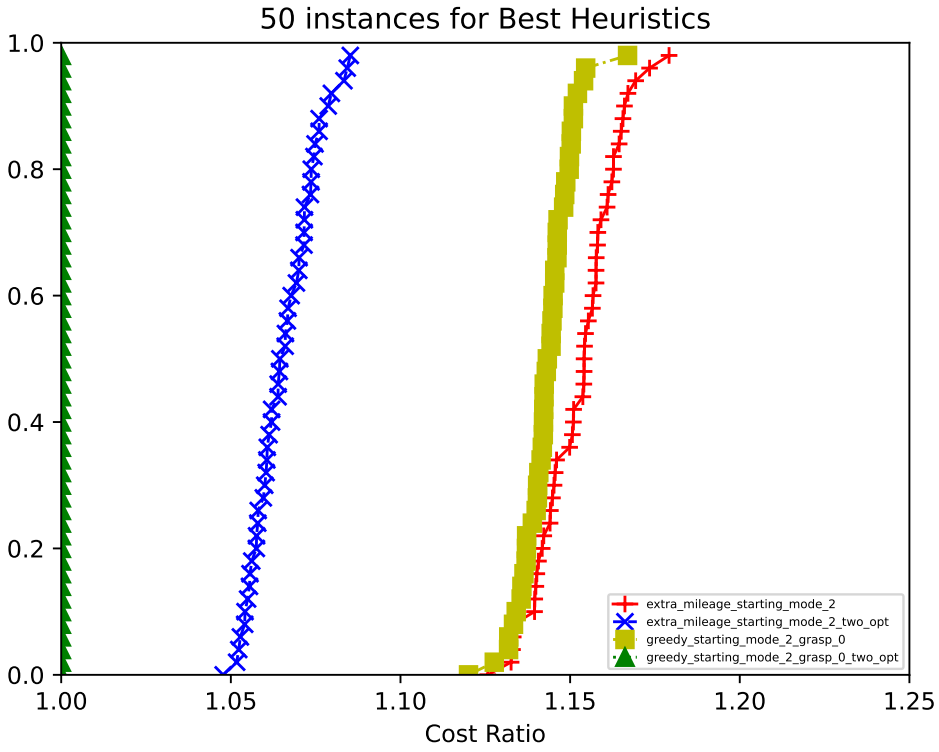


Figure 7.5: Best heuristic methods performance profile.

7.2 Metaheuristics

Due to the numerous parameters and methods involved in metaheuristics, we used only 10 instances with different seeds. Each instance had a node size of 500 and a time limit of 10 minutes. Please note that unlike heuristic solutions, even with a time limit, metaheuristic solutions will always provide a valid solution. In this category of solutions, we calculated optimal values for the given instances using exact methods, as the node size was not excessively large. This allowed us to better compare and assess the quality of the produced results.

For all implemented methods, it was necessary to apply a heuristic method initially, except for the Genetic algorithm, which starts from a random initialization of nodes. We chose the Standard Greedy algorithm, starting from a random position. This decision was made because the standard mode performs better, and we still wanted to introduce some randomness, hoping it would aid the metaheuristic search in the solution space.

7.2.1 Variable Neighbourhood Search

For the VNS (Variable Neighborhood Search) algorithm, we tested four different neighborhoods to determine which one yields the best performance. From the plot, it is evident that as the neighborhood size increases, the performance of VNS improves, coming closer to the optimal value.

The performance profiling is shown in Figure 7.6.

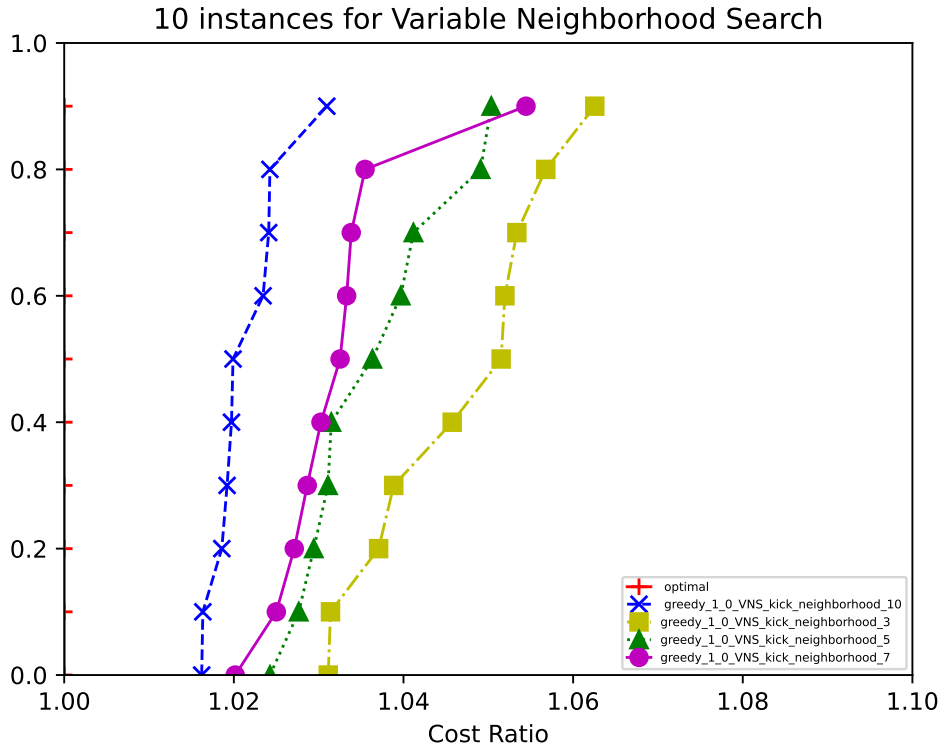


Figure 7.6: VNS performance profile.

We also analyzed the cost values for the VNS algorithm with a time limit of 5 seconds, using different types of kicks on the a280.tsp instance provided by Heidelberg University [33]. The VNS implementation is based on a Greedy Heuristic with the starting mode set to 2 (trying all initial nodes and choosing the best one), and without using GRASP. This analysis shows how the cost range changes as we vary the neighborhood size.

The cost analyze is shown in Figure 7.7

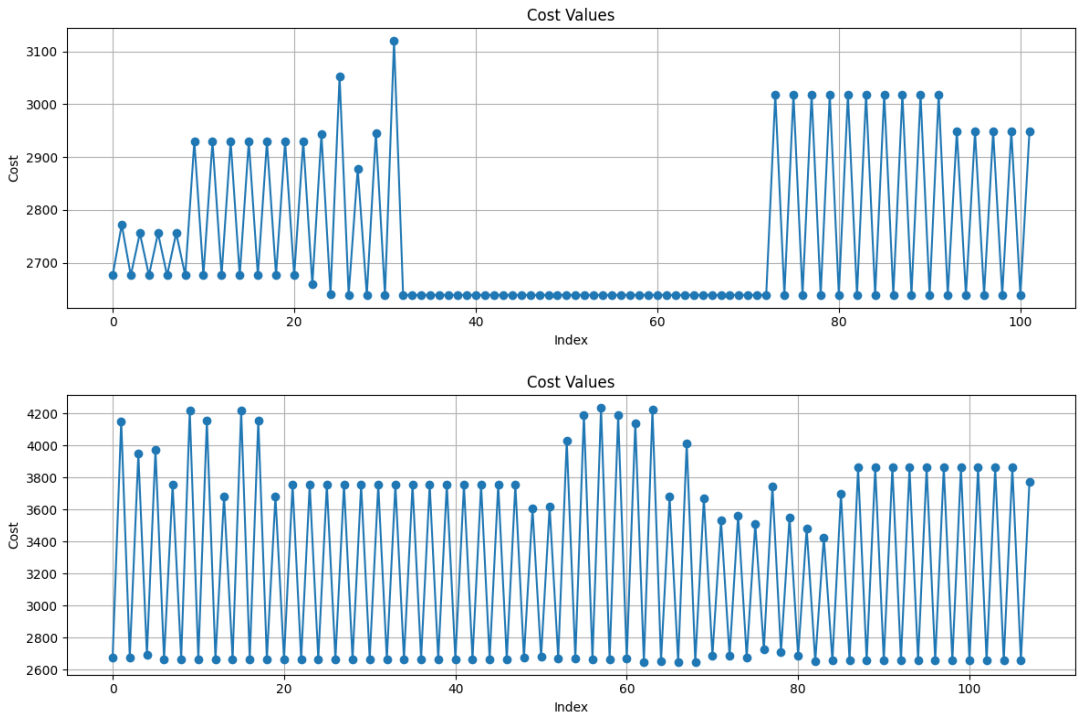


Figure 7.7: VNS cost values: 3-OPT kick (top) and 10-OPT kick (bottom).

7.2.2 Simulated Annealing

From the plot, it is evident that there is no clear winner in this experiment. To better tune the number of iterations for Simulated Annealing, we suggest trying more instances. Although using 2 iterations seems to perform slightly better in the beginning, it becomes either similar or worse for the later nodes. Since there is no definitive best option, any of these settings would be acceptable. For comparison with other metaheuristic results, we chose the configuration with 3 iterations, as its performance profile was more vertical and straightforward, making it easier to compare.

The performance profiling is shown in Figure 7.8.

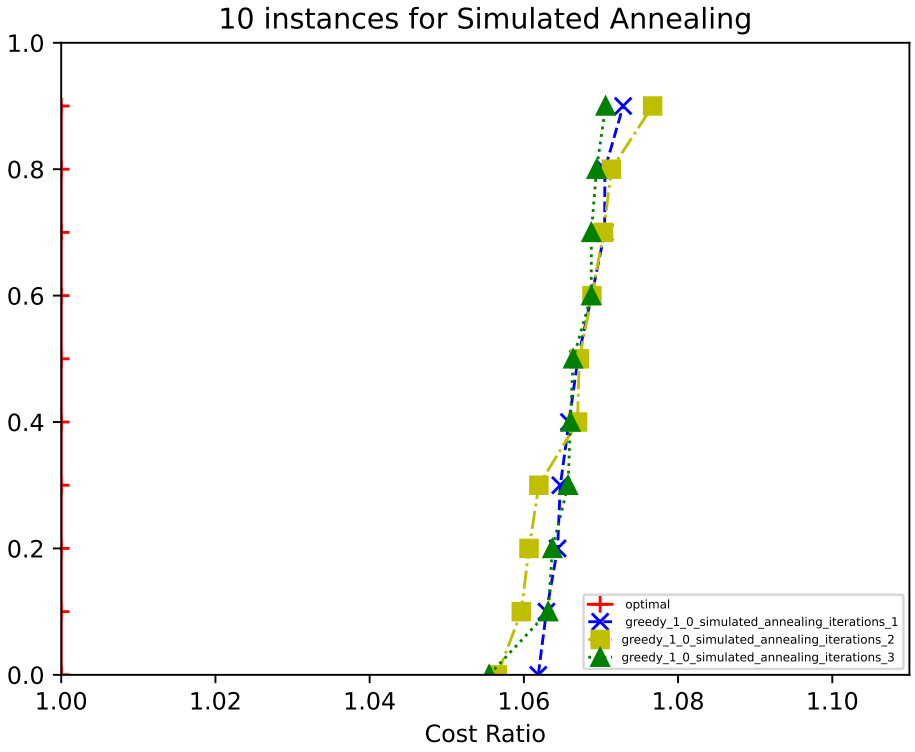


Figure 7.8: Simulated Annealing performance profile.

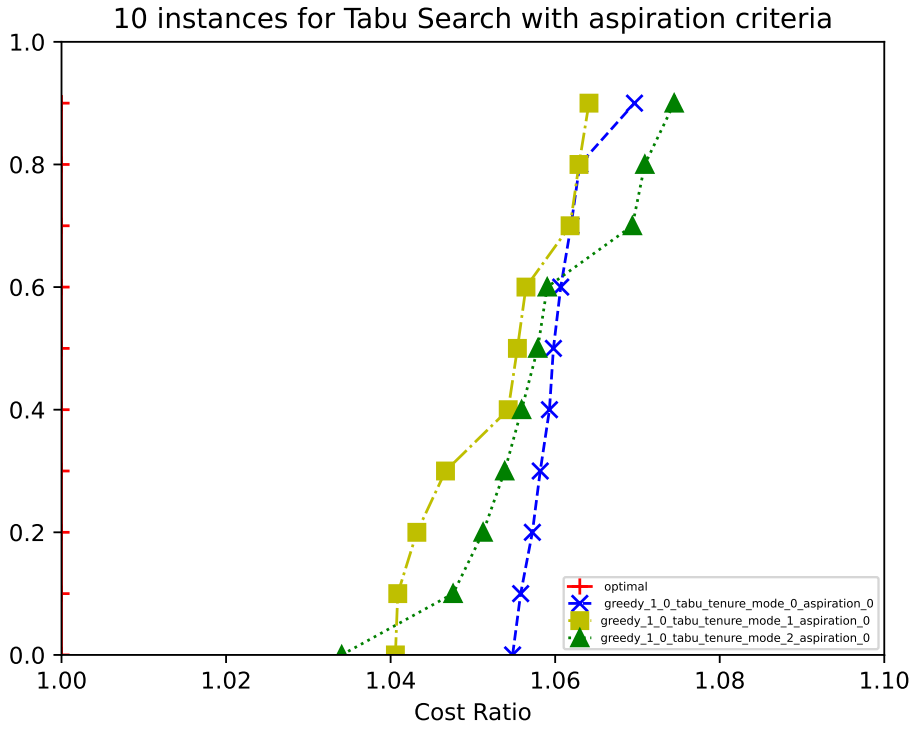
7.2.3 Tabu Search

For Tabu Search, we tested three tenure modes: Reactive step (0), Reactive line (1), and Random (2) for each aspiration mode. The results indicate that applying aspiration criteria (AC) generally improves performance, particularly for reactive line tenure. However, this is not consistent across all modes; reactive step tenure worsens, and random tenure exhibits increased dispersion in performance.

Reactive line with AC outperforms other modes, both with and without AC. Without AC, there is no clear winner, though reactive step mode seems closer to the optimal value, indicating the need for more instances for better comparison.

While additional instances would help clarify differences for modes with AC, reactive line tenure performed better in 9 out of 10 instances compared to random tenure, and in 8 compared to reactive step tenure. Thus, reactive line tenure with AC is chosen as the preferred method for comparing metaheuristic methods.

The performance profiling is shown in Figure 7.9.



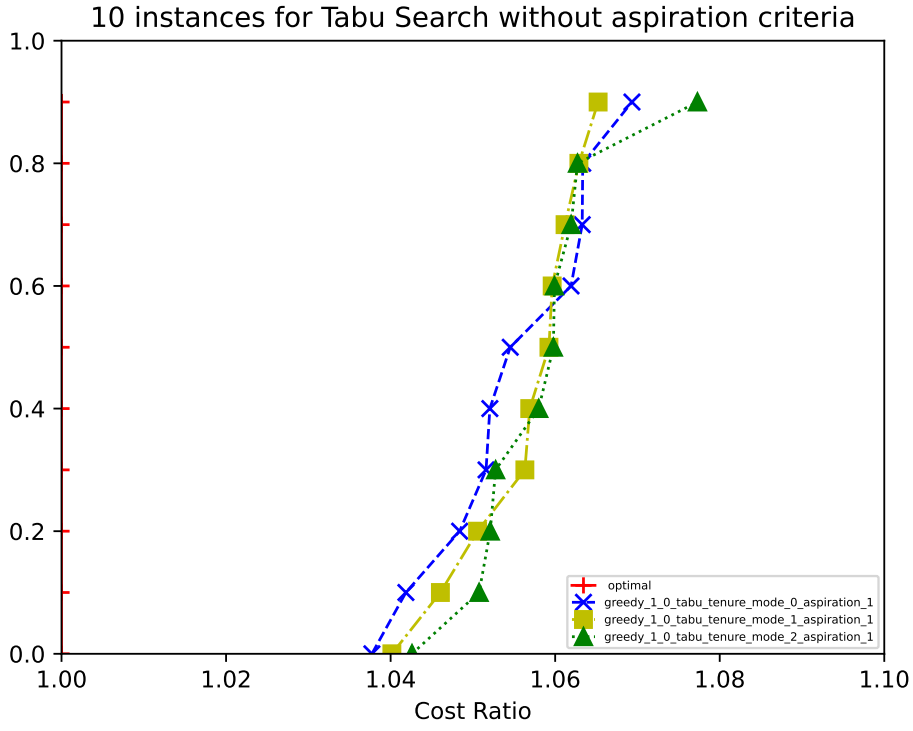


Figure 7.9: Tabu Search performance profile.

7.2.4 Genetic Algorithm

We assessed three repair modes (0: Penalizing fitness without repair, 1: Repairing bad genes, 2: Repairing bad genes with two-opt refinement) with three crossover cutting types. Despite similar performance between modes 0 and 1 with midpoint cutting, overall, the hierarchy is: mode 2, mode 1, mode 0.

The performance profiling is shown in Figure 7.10.

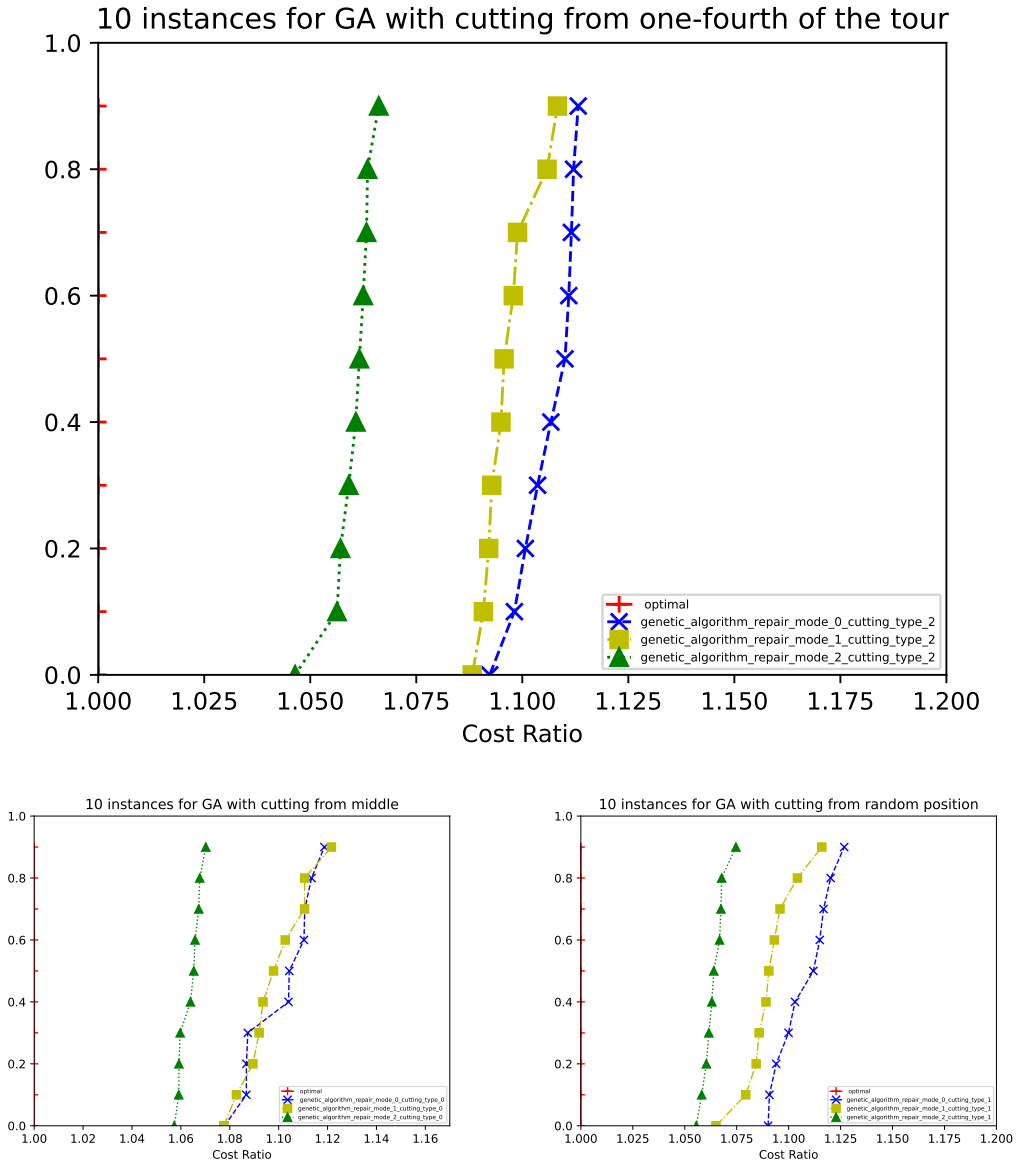


Figure 7.10: Genetic algorithm performance profile for cutting type.

After analyzing with a fixed cutting type, we have concluded that repair mode 2 yields the most optimal results. Now, our focus shifts to selecting the most suitable cutting type for the given repair mode. It's essential to recall that we have considered three different crossover cutting types: 0) Midpoint cutting, 1) Random position cutting, and 2) One-fourth tour cutting.

The performance profiling is shown in Figure 7.11.

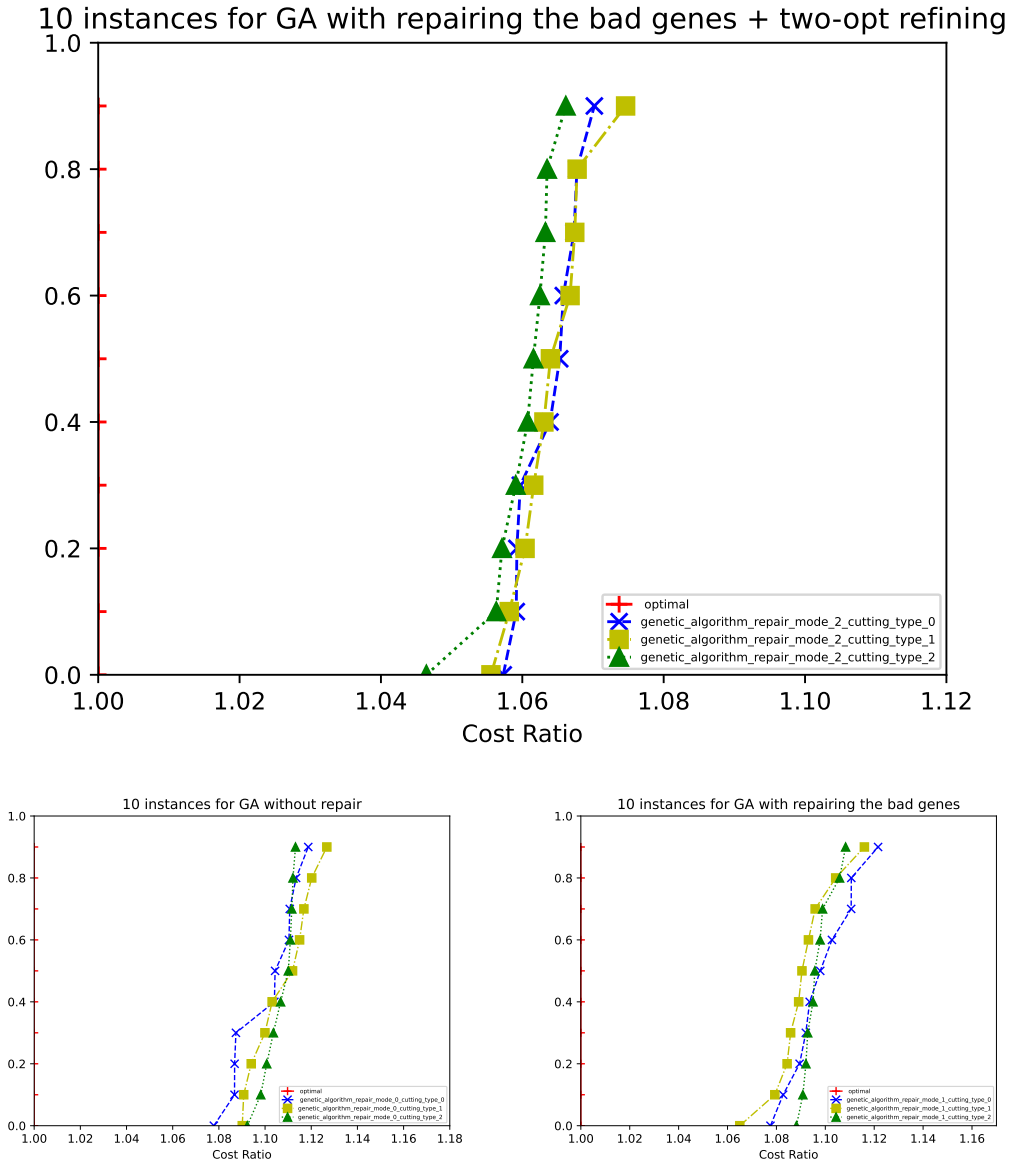


Figure 7.11: Genetic algorithm performance profile for repair mode.

From the plots, it's evident that cutting types exhibit diverse performances for each repair mode, yet they remain relatively close to one another. Given our emphasis on repair mode 2, the superior performer, we observe that cutting types 0 and 1 demonstrate comparable performance, while cutting type 2 consistently outperforms them albeit slightly across all 10 instances. Consequently, we select the Genetic Algorithm with full repair mode, utilizing cutting from one-fourth, for comparison with the best metaheuristic methods.

7.2.5 Best methods comparison

The plot clearly illustrates that Variable Neighborhood Search consistently outperforms all other implemented methods across a broad spectrum. The performance hierarchy can be summarized as follows: Variable Neighborhood Search, Tabu Search, Genetic Algorithm, and Simulated Annealing.

The performance profiling is shown in Figure 7.12.

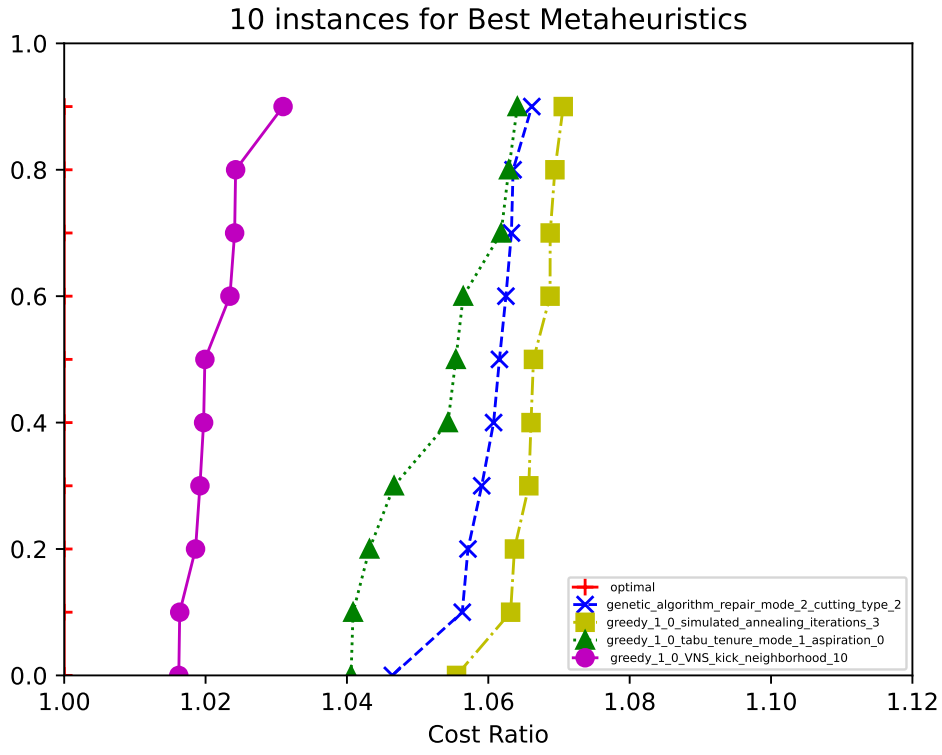


Figure 7.12: Best metaheuristic methods performance profile.

7.3 Exact Methods

For the exact methods, we used 20 instances with different seeds, each with a node size of 500. In this category, both methods run until convergence, meaning they continue until finding the optimal solution. They are compared based on the time taken to reach this solution. The plot clearly shows that the Callback method (Branch and Cut) outperforms the Benders' loop method. Although in some instances, the Benders' loop method approaches the performance of the Callback method, in others, it can approximately be 10 times slower, which could be highly detrimental in real-world scenarios. The significant disparity is likely because the Benders' loop method rebuilds the solution from scratch each time, whereas the Callback method does not.

The performance profiling is shown in Figure 7.13.

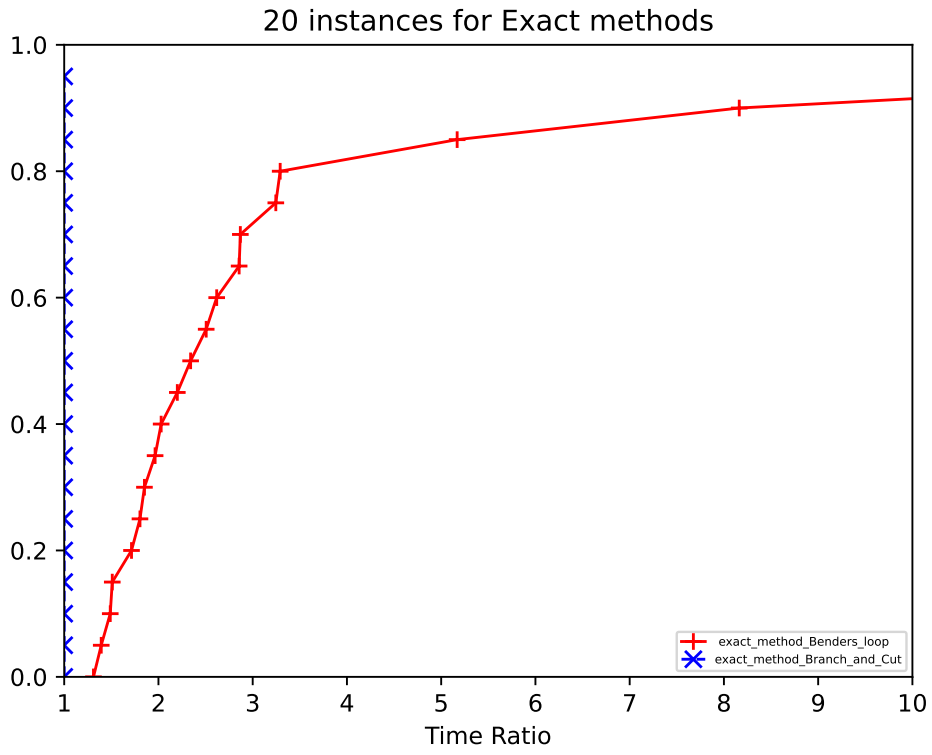


Figure 7.13: Exact methods performance profile.

7.4 Matheuristics

Similar to metaheuristic methods, matheuristic methods are evaluated within a set time limit, as they are designed to find the best possible solution within a given timeframe or, more precisely, with the smallest primal integral possible. For our experiments, we tested 10 instances with different seeds, each with a size of 1000 nodes and a time limit of one minute.

7.4.1 Hard Fixing

The plot shows that Hard-Fixing performs better when the probability of fixing is 0.5 or 0.8 compared to 0.2 or 0.3. This suggests that more instances should be used to fine-tune this parameter as needed. Although 0.5 is not a definitive winner, we chose it for comparison among the best matheuristic methods.

The performance profiling is shown in Figure 7.14.

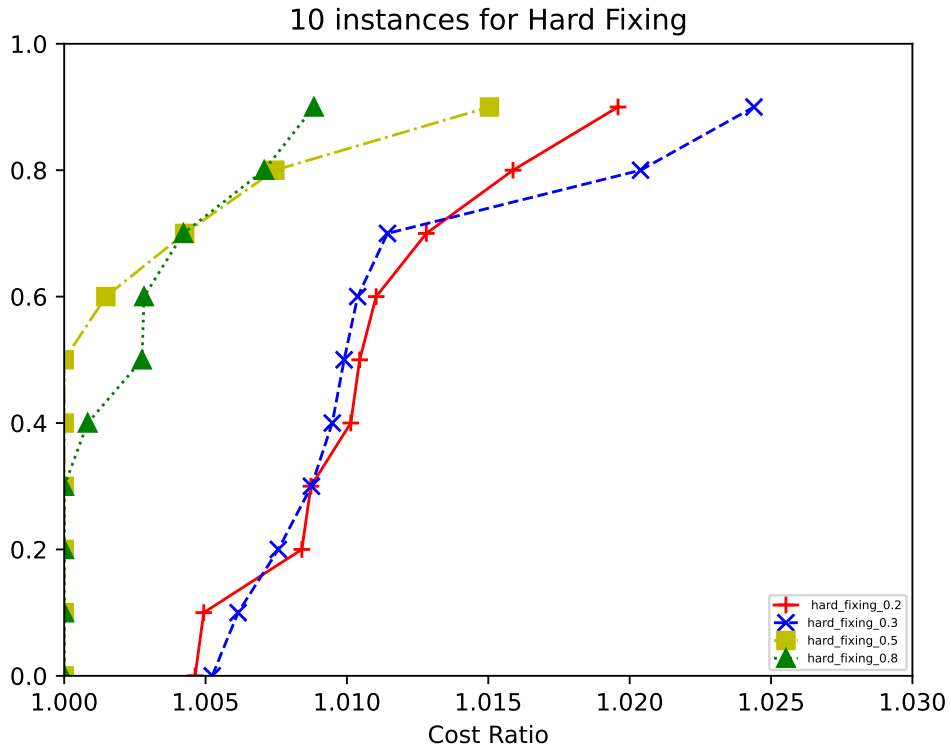


Figure 7.14: Hard Fixing performance profile.

7.4.2 Local Branching

The plot indicates that the method performs best when the local branching constraint (radius) is set to 10. For radius of 20 and 30, there is no clear winner, although a radius of 20 generally outperforms 30, it showed worse results in 2 out of 10 instances. Thus, testing more instances would be beneficial for making a well-informed decision. It was expected that a larger radius would increase performance due to a larger search space and neighborhood. However, the one-minute time limit likely played a significant role in these outcomes. Consequently, we chose a branching constraint of 10 for comparison among the best matheuristic methods.

The performance profiling is shown in Figure 7.15.

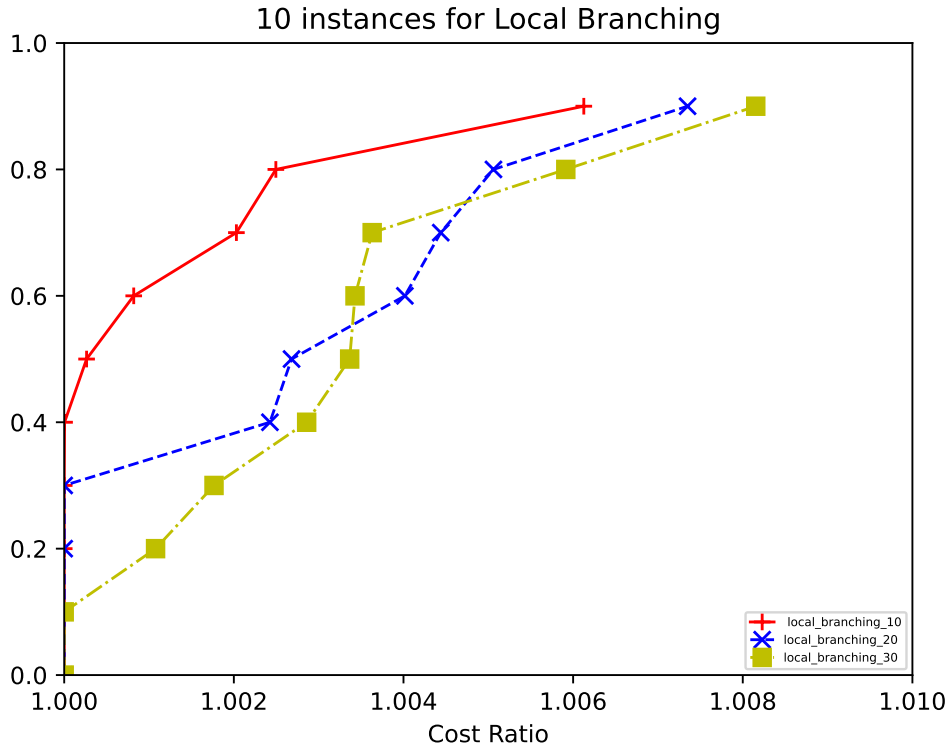


Figure 7.15: Local Branching performance profile.

7.4.3 Best methods comparison

The plot shows that Hard-Fixing method performs better than Local Branching. However, this is true for the given node size and time limit. Node size and time limit shows a huge role in deciding about which one to use. As the test we conducted was easy, which just 1000 nodes in 1 minute, so a simpler method showed better result. But it is likely to expect better results from Local Branching for larger node sizes and specially considering that the fact that parameter tuning of Local Branching was suspicious, that clearly shows that this algorithm has not shown its full potential.

The performance profiling is shown in Figure 7.16.

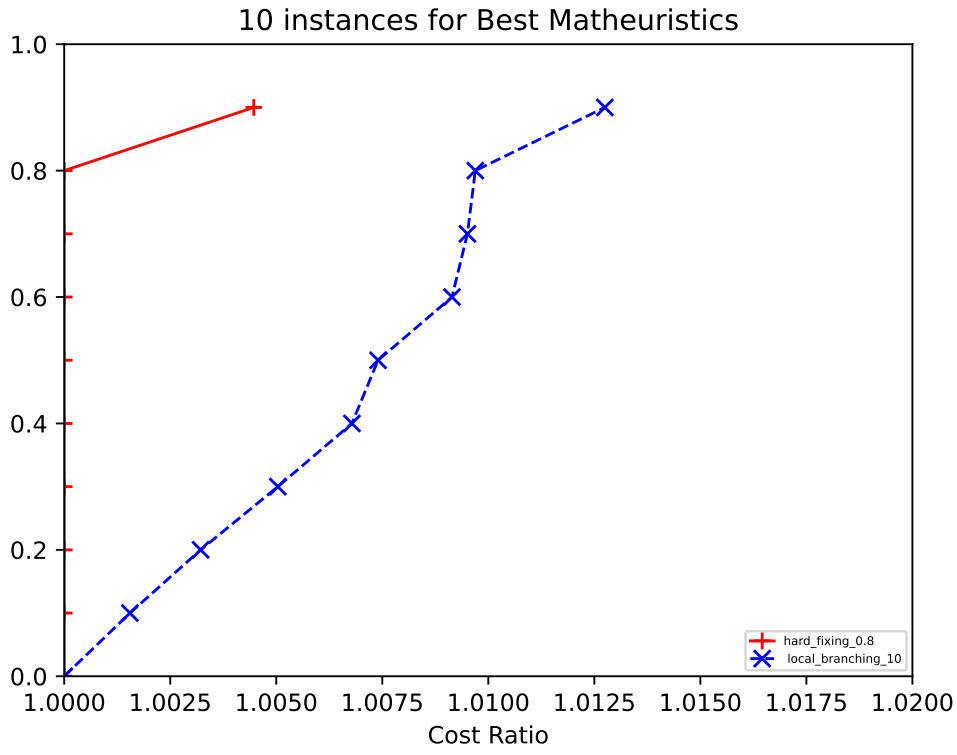


Figure 7.16: Best matheuristic methods performance profile.

8 Conclusions

This thesis aimed to explore and analyze various methods for solving the Symmetric Traveling Salesman Problem (TSP), with a focus on heuristics, metaheuristics, exact methods, and matheuristics.

8.1 Summary of Key Findings

- **Constructive Heuristics:** The Insertion Heuristic and Greedy Heuristics provided quick, although not optimal, solutions. The 2-OPT significantly improved initial solutions, highlighting the importance of refinement in heuristic methods.
- **Metaheuristics:** Variable Neighbourhood Search (VNS), Tabu Search, Genetic Algorithm, and Simulated Annealing each demonstrated strengths in exploring solution spaces, with VNS yielding particularly competitive results.
- **Exact Methods:** Benders' loop and Callback method provided exact solutions, effectively solving the problem by finding the optimal solution. The Callback method showed better results and was more practical, as the latter took too much time to converge for some instances.
- **Matheuristics:** Hard-Fixing and Local Branching combined heuristic and exact approaches, offering a balance between solution quality and computational effort. In our tests with a small node size of 1000 nodes and a 1-minute time limit, Hard-Fixing proved to be the better method.

8.2 Comparative Analysis

Among the methods tested, VNS stood out for its balance of solution quality and computational efficiency in metaheuristics, finding very good results early in the search. The exact methods, while providing precise solutions, were less practical for larger instances due to high computational demands. Matheuristics like Hard-Fixing showed promise in balancing these aspects, suggesting a viable path for practical implementations.

Appendices

A Command line parsing

This section provides an overview of the command line parameters and options available. The following info is also found in helper function, `print_help`, which is used to display these options along with their descriptions and current values. This can be useful for users to understand how to properly configure and execute the program.

- **Parameters**
  - `-file <input_file>`
    Specifies the input file.
  - `-time_limit <timelimit>`
    Sets the time limit for the execution.
  - `-generate_random <nnodes>`
    Generates a random instance with the specified number of nodes.
  - `-seed <random_seed>`
    Sets the random seed for reproducibility.
- **Algorithms**
  - `-greedy <starting_mode> <grasp_mode>`
    Executes the greedy algorithm with the specified starting mode and GRASP (Greedy Randomized Adaptive Search Procedure) mode.
  - `-extra_mileage <starting_mode>`
    Runs the extra mileage algorithm with the specified starting mode.
  - `-two_opt`
    Executes the 2-opt algorithm, a local search optimization technique.
  - `-VNS <kick_neighborhood>`
    Runs the Variable Neighborhood Search (VNS) with the specified kick neighborhood.

- `-SA <iterations>`
  Executes the Simulated Annealing (SA) algorithm with the specified number of iterations.
- `-tabu_search <tenure_mode> <aspiration>`
  Runs the Tabu Search algorithm with the specified tenure mode and aspiration criteria.
- `-genetic_algorithm <repair_mode> <cutting_type>`
  Executes the Genetic Algorithm with the specified repair mode and cutting type.
- `-tsp_opt <model_type>`
  Runs the TSP optimization algorithm with the specified model type. 0: Benders' Loop. - 1: Branch and Cut.
- `-hard_fixing <probability>`
  Executes the hard fixing procedure with the specified probability.
- `-local_branching <branching_constraint>`
  Runs the local branching algorithm with the specified branching constraint.

- **Helper**
  - For further assistance, enter `-help` or `--help`.

B Plotting: Gnuplot

Gnuplot is a powerful plotting program used for generating high-quality plots and graphs from data. It supports various types of plots, including 2D and 3D plots, histograms, and contour plots. Gnuplot is widely used in scientific research, data analysis, and visualization tasks due to its flexibility and ease of use.

The following simple Gnuplot code snippet demonstrates how to configure plot styles and plot data from a file:

```
set style line 1 \
        linecolor rgb '#FF0000' \
        linetype 1 linewidth 2 \
        pointtype 7 pointsize 2

plot "./plot/data.dat" with linespoints linestyle 1

pause mouse close
```

The above code sets the style for the plot, plots data from a file named `data.dat` using lines and points, and pauses the plot window until the user interacts with it.

This script can be executed in Gnuplot to generate a plot based on the data provided in the file `data.dat`. Adjustments can be made to the plot style and data source as needed.

C Shell Scripts

This appendix provides a detailed description of the shell scripts used to automate various tasks. Each method type has its own script to streamline the process, and additional scripts are used for extracting values from logs, creating performance profiles, and storing optimal values.

C.1 Log Scripts

For each method, there are several algorithms with their own parameters to tune. We need results from each of them, so we decided to log all outputs and save them. To run all the methods and log the output, the following scripts were implemented to automate the process:

- `log_heur.sh`: Heuristic algorithms.
- `log_exact.sh`: Exact methods.
- `log_matheur.sh`: Matheuristic algorithms.
- `log_metaheur.sh`: Metaheuristic algorithms.

C.2 Optimal Values Script

- `find_optimal.sh`: This script is used to store the optimal values for the given instances using our best exact method (Callback method). These values are crucial for comparing the performance of various algorithms against known optimal solutions. In our use case, it was applied only for metaheuristics due to computational expenses.

C.3 Log Extraction Script

- `logs_to_csv.sh`: This script extracts the corresponding values from the log files based on the given quality measure (cost or time) and creates a CSV file for the given algorithm. It is also used to create a CSV file for the optimal values logged by `find_optimal.sh`.

C.4 Performance Profile Script

- `csv_to_pp.sh`: This script automates the creation of performance profile files from the CSV files generated by `logs_to_csv.sh`. Performance profiles help visualize the effectiveness of different methods.

All scripts for the given methods are in the same folder as the method scripts. However, the script for logging optimal values is in the `perf_prof` folder, at the same level as `logs_to_csv.sh` and `csv_to_pp.sh`.

D Convex Hull: Graham scan

The Convex Hull problem involves finding the smallest convex polygon that wraps around a group of points on a plane. One way to solve this problem is by using the Graham Scan algorithm. [15]

1. **Choose a Starting Point:** Start by picking the point with the lowest y-coordinate. If there are multiple points with the same lowest y-coordinate, choose the one with the lowest x-coordinate.

2. **Sort the Points:** Arrange the remaining points in a clockwise order around the starting point by sorting them based on their polar angles with respect to the starting point.

3. **Scan the Points:** Go through the sorted points one by one. For each point:

   - If the point is collinear with the last two points in the convex hull, skip it. Collinear points do not contribute to the convex shape of the hull.
   - If it makes a left turn with the last two points in the hull, include it in the hull.
   - If it makes a right turn, remove the last point added to the hull. Repeat this until a left turn is made or until all the points are scanned.

4. **Done:** The points left in the hull after the scanning process are the vertices of the convex hull.

E Verbose Levels

Verbose levels are defined to provide detailed information during the execution of algorithms. These levels include:

- Verbose 20: Print algorithm names for Metaheuristics/Matheuristics.
- Verbose 50: Print updates on the best values found by M-heuristics.
- Verbose 100: Print current solution information for M-heuristics.
- Verbose 110: Print algorithm names for Exact Methods.
- Verbose 120: Print upper bound (UB), lower bound (LB), and other information for exact methods.
- Verbose 130: Print information such as thread, node, incumbent, and candidate values for Branch and Cut.
- Verbose 140: Print algorithm names for Heuristics.
- Verbose 150: Print starting positions for Heuristics.
- Verbose 500: Print information about 2-OPT algorithm.
- Verbose 550: Print updates on the value found by 2-OPT.
- Verbose 600: Print edges of the last CPLEX solution.
- Verbose 1000: Print if a possible improvement is found by 2-OPT.

Additionally, CPLEX has its own verbosity setting defined as `CPLEX_VERBOSE`, with options CPX_ON and CPX_OFF.

Bibliography

[1] David L. Applegate et al. *The Traveling Salesman Problem: Computational Solutions*. Princeton University Press, 2006.

[2] Jacques F Benders. "Partitioning procedures for solving mixed-variables programming problems". In: *Numerische Mathematik* 4.1 (1962), pp. 238–252.

[3] Gerardo Berbeglia et al. "Metaheuristics for the pickup and delivery problem with time windows". In: *Computers & Operations Research* 34.11 (2007), pp. 3281–3297.

[4] Miroslaw Blocho. "Chapter 4 - Heuristics, metaheuristics, and hyperheuristics for rich vehicle routing problems". In: *Smart Delivery Systems*. Ed. by Jakub Nalepa. Intelligent Data-Centric Systems. Elsevier, 2020, pp. 101–156. ISBN: 978-0-12-815715-2. DOI: https://doi.org/10.1016/B978-0-12-815715-2.00009-9. URL: https://www.sciencedirect.com/science/article/pii/B9780128157152000099.

[5] Guillermo Cabrera-Guerrero et al. "Parameter Tuning for Local-Search-Based Matheuristic Methods". In: *Complexity* 2017 (Dec. 2017), pp. 1–15. DOI: 10.1155/2017/1702506.

[6] Nicos Christofides. "Worst-case analysis of a new heuristic for the traveling salesman problem". In: *Operations Research* 21.4 (1976), pp. 1041–1044.

[7] George A. Croes. "A Method for Solving Travelling-Salesman Problems". In: *Operations Research* 6.6 (1958), pp. 791–812.

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

[9] George B. Dantzig. *Linear programming and extensions*. Princeton University Press, 1963.

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

[11] Martina Fischetti and Matteo Fischetti. "Matheuristics". In: *Handbook of Heuristics*. Ed. by Rafael Martí, Panos M. Pardalos, and Mauricio G. C. Resende. Cham: Springer International Publishing, 2018, pp. 121–153. ISBN: 978-3-319-07124-4. DOI: 10.1007/978-3-319-07124-4_14. URL: https://doi.org/10.1007/978-3-319-07124-4_14.

[12] Matteo Fischetti and Andrea Lodi. "Local branching". In: *Mathematical Programming* 98.1-3 (2003), pp. 23–47.

[13] Delbert R. Fulkerson. "The Traveling Salesman Problem". In: *Operations Research* 13.3 (1965), pp. 476–486.

[14] Fred Glover. "Future Paths for Integer Programming and Links to Artificial Intelligence". In: *Computers and Operations Research* 13.5 (1986), pp. 533–549.

[15] R. L. Graham. "An efficient algorithm for determining the convex hull of a finite planar set". In: *Information Processing Letters* 1.4 (1972), pp. 132–133.

[16] P. H. Gunawan and I. Iryanto. "Simulated Annealing – 2 Opt Algorithm for Solving Traveling Salesman Problem". In: *International Journal of Computing* 22.1 (2023), pp. 43–50. DOI: 10.47839/ijc.22.1.2878. URL: https://doi.org/10.47839/ijc.22.1.2878.

[17] Pierre Hansen, Nenad Mladenović, and José A. Moreno Pérez. "Variable neighbourhood search: methods and applications". In: *4OR* 6 (2008), pp. 319–360. DOI: 10.1007/s10288-008-0089-1. URL: https://doi.org/10.1007/s10288-008-0089-1.

[18] John H. Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975. ISBN: 978-0472084606.

[19] Selmer M. Johnson. "Optimal two-and three-level networks". In: *Operations Research* 2.6 (1954), pp. 614–641.

[20] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. "Optimization by simulated annealing". In: *Science* 220.4598 (1983), pp. 671–680.

[21] S. Lin and B. W. Kernighan. "An effective heuristic algorithm for the traveling-salesman problem". In: *Operations Research* 21.2 (1973), pp. 498–516.

[22] Sean Luke. *Essentials of Metaheuristics*. Second. Available at http://cs.gmu.edu/~sean/book/metaheuristics/. Lulu, 2013.

[23] Karl Menger. "On the problem of the least round trip from one point to n others". In: *Aequationes mathematicae* 8.1 (1930), pp. 225–247.

[24] Nicholas Metropolis et al. "Equation of state calculations by fast computing machines". In: *The journal of chemical physics* 21.6 (1953), pp. 1087–1092.

[25] Nenad Mladenović and Pierre Hansen. "Variable neighborhood search". In: *Computers & Operations Research* 24.11 (1997), pp. 1097–1100.

[26] José Moreno-Pérez et al. "Variable Neighbourhood Search". In: vol. 7. Jan. 2006, pp. 71–86. ISBN: 978-0-387-33415-8. DOI: 10.1007/0-387-33416-5_4.

[27] Kenneth Sörensen and Fred W. Glover. "Metaheuristics". In: *Encyclopedia of Operations Research and Management Science*. Ed. by S. I. Gass and M. C. Fu. Boston, MA: Springer, 2013.

[28] Marcel Turkensteen et al. "Iterative patching and the asymmetric traveling salesman problem". In: *Discrete Optimization* 3.1 (2006). The Traveling Salesman Problem, pp. 63–77. ISSN: 1572-5286. DOI: https://doi.org/10.1016/j.disopt.2005.10.005. URL: https://www.sciencedirect.com/science/article/pii/S1572528605000770.

[29] B. Fr. Voigt. *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*. Ilmenau, 1832.

[30] Akang Wang et al. "Efficient Primal Heuristics for Mixed-Integer Linear Programs". In: *Proceedings of the NeurIPS 2021 Competition and Demonstration Track*. Shenzhen Research Institute of Big Data, Huawei GTS. 2021. URL: https://www.ecole.ai/2021/ml4co-competition/proceedings/2.pdf.

Sitography

[31] *Doxygen Documentation Generator*. http://www.doxygen.nl/. Accessed: May 12, 2024.

[32] *IBM ILOG CPLEX Optimization Studio*. https://www.ibm.com/it-it/products/ilog-cplex-optimization-studio. Accessed: May 10, 2024.

[33] Gerhard Reinelt. *TSPLIB - A Traveling Salesman Problem Library*. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Accessed: 2024-05-23. 1995.

[34] Tutorials Point. *Genetic Algorithms - Crossover*. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm. Accessed: May 18, 2024.