POLITECNICO DI TORINO

Master's degree in Management Engineering

Master's degree

Min-Max Electric Vehicle Routing Problem: Exact Model and Heuristic Approach

Supervisor

Prof. Fabio SALASSA

Candidate

Francesco Lucio SIMEONI

November 2025

Summary

The growing adoption of electric vehicles (EVs) represents both a challenge and an opportunity for sustainable logistics. While EVs reduce greenhouse gas emissions, they introduce new operational complexities related to limited battery autonomy and the uneven distribution of charging stations. In this context, the electric vehicle routing problem plays a central role in transportation planning.

This thesis addresses the Min-Max Electric Vehicle Routing Problem (MEVRP), a variant of the classical Vehicle Routing Problem (VRP) that focuses on minimizing the maximum distance traveled by a vehicle in the fleet. Unlike the traditional objective of minimizing total distance, the min-max criterion ensures a more balanced distribution of routes, reducing the risk of overloading certain vehicles and contributing to longer battery lifespans.

The work is developed in two main phases. First, an exact mathematical model of the problem was formulated and implemented in Python using the PuLP library, solved through a branch-and-cut approach, and tested on small-scale instances. Then, to overcome the computational limits of the exact model, a heuristic algorithm in Python was developed, capable of solving larger instances within reasonable computation times.

Experimental results show that the exact approach guarantees optimal solutions for small datasets, whereas the heuristic, while giving up absolute optimality, is able to provide high-quality solutions in significantly shorter times. The comparison between the two methods highlights the trade-offs between accuracy and scalability, offering valuable insights both from a methodological perspective and for practical applications.

The thesis concludes with a discussion of the contributions achieved, the limitations of the work, and possible future extensions, including the introduction of time windows, multi-depot settings, and real-world applications in urban logistics.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Thesis Objectives and Structure

This thesis addresses the challenge of routing electric vehicles under a min–max criterion, with the aim of analyzing the problem and proposing solution methods that balance theoretical rigor with practical applicability. The work combines an exact mathematical modeling approach, suitable for small-scale instances, with the design of a heuristic capable of handling larger and more realistic scenarios within reasonable computation times.

The thesis is organized into six chapters. **Chapter 2** reviews the relevant literature, starting from the classical Vehicle Routing Problem and moving through its green and electric variants, with a particular focus on the role of min–max objectives. **Chapter 3** introduces the mathematical formulation of the MEVRP and describes the exact approach adopted. **Chapter 4** presents the heuristic method developed to address larger instances, outlining its structure and underlying logic. **Chapter 5** reports the computational experiments, comparing the performance of the exact model and the heuristic across different instance sizes and levels of complexity. Finally, **Chapter 6** discusses the main contributions of the work, highlights its limitations, and suggests possible directions for future research.

1.1 Background and Motivation

Climate change and the increase in greenhouse gas emissions have made it urgent to develop sustainable solutions for the transportation sector, which accounts for a significant share of global pollution. Against this backdrop, many countries have adopted policies to accelerate the transition towards sustainable mobility, including incentives for zero-emission vehicles and investments in charging infrastructure.

In Europe, these efforts are framed within the **European Green Deal** and the

Fit for 55 legislative package, which set the target of reducing net greenhouse gas emissions by 55% by 2030 compared to 1990 levels. A key step in this direction is **Regulation (EU) 2023/851**, which revises CO_2 performance standards for new passenger cars and light commercial vehicles and establishes that, from **2035 onwards**, all new vehicles registered in the European Union must be zero-emission[1, 2].

These policy measures, combined with growing mobility demand driven by urbanisation and the expansion of e-commerce, make electric vehicles an increasingly central component of transportation planning. At the same time, the transition to electric mobility requires significant infrastructural investment—particularly a dense and reliable charging network. Together with the inherent limitations of battery autonomy, these aspects create new operational and decision-making challenges that call for advanced analytical and optimisation tools. One of the most promising responses is the adoption of Electric Vehicles (EVs), which reduce environmental impact by eliminating direct emissions.

However, the integration of EVs into logistics and urban transport fleets introduces new operational challenges. Compared to conventional vehicles, EVs are characterized by two main limitations:

- **Reduced battery autonomy**, which requires planning routes compatible with the vehicle's energy capacity.
- **Limited and uneven availability of charging stations**, which may affect the feasibility of routes.

These features make electric vehicle routing problems more complex than the classical Vehicle Routing Problem (VRP) [3]. The need to incorporate recharging strategies into route planning has led to the development of new problem variants, such as the Electric Vehicle Routing Problem (EVRP, Electric Vehicle Routing Problem) and, more recently, the Min-Max EVRP (MEVRP, Min-Max Electric Vehicle Routing Problem).

The adoption of a min-max criterion, in particular, allows for a more balanced distribution of routes across the fleet, reducing the maximum distance traveled by any EV. This approach not only improves fairness in vehicle utilization but also contributes to extending battery lifetime and increasing service reliability.

The problem addressed in this thesis falls within the scope of *Operations Research (OR)*, a discipline that develops mathematical models and algorithms to support complex decision-making processes in real-world contexts [4]. The VRP is one of the most widely studied problems in OR and serves as a benchmark in logistics and transportation planning. Its extensions, such as the EVRP and the MEVRP, clearly demonstrate how OR evolves to meet new requirements, in this case related to sustainability and the management of electric vehicle fleets.

In this perspective, the present work is framed within applied OR, with the aim of providing solution methods that combine mathematical rigor with computational efficiency.

1.2 Problem Statement

The problem addressed in this thesis is the Min-Max Electric Vehicle Routing Problem (MEVRP, Min-Max Electric Vehicle Routing Problem), a variant of the classical Vehicle Routing Problem (VRP). In the traditional VRP, the main objective is to minimize the total distance traveled by the fleet, thereby reducing overall transportation costs.

The MEVRP, instead, aims to minimize the maximum distance traveled by any single vehicle in the fleet. This min-max criterion ensures a balanced distribution of workloads across vehicles, preventing one vehicle from being assigned a disproportionately long route. In practice, this leads to:

- **greater fairness** in route assignment,
- **reduced battery degradation** through more uniform usage,
- **improved service reliability**, particularly in critical scenarios (e.g., urban logistics, emergency response, or surveillance tasks).

Additionally, the problem accounts for the specific constraints of electric vehicles: limited battery autonomy and the need to recharge at stations that are unevenly distributed across the network. A feasible solution must therefore guarantee that:

1. all customers or targets are visited exactly once by a vehicle,
2. each vehicle starts and ends its route at the depot,
3. no vehicle runs out of charge during the route, being allowed to stop for recharging at available stations if necessary.

The MEVRP is an **NP-hard** problem, inherited from the classical VRP [5], which makes exact solution approaches computationally expensive. While exact mathematical models can be applied to small-scale instances, larger instances require heuristic or metaheuristic methods capable of producing high-quality solutions within reasonable computation times.

1.3 Relevance and Applications

The *Min-Max Electric Vehicle Routing Problem (MEVRP)* is relevant from both a theoretical and an applied perspective. From a methodological standpoint, it extends the classical Vehicle Routing Problem (VRP) by incorporating constraints specific to electric vehicles, such as limited battery capacity and the need to recharge at designated stations. Unlike the VRP, which aims to minimize the total distance traveled by the fleet, the MEVRP focuses on minimizing the maximum distance traveled by a single vehicle. This objective leads to a more balanced distribution of routes and a more uniform utilization of resources.

From an operational perspective, adopting the min-max criterion provides several benefits:

- it balances battery usage, reducing degradation rates;
- it improves service reliability by limiting maximum travel times;
- it ensures fairer route allocation across the fleet.

The problem is of particular interest in application domains where the maximum distance, and thus the maximum service time, is more critical than the overall transportation cost [6, 7]. Notable examples include the following:

Urban distribution and last-mile delivery. With the growth of e-commerce and same-day delivery services, logistics providers must guarantee reliable and fast service across dense urban networks. Minimizing the maximum route length helps avoid situations where some vehicles are overburdened while others are underutilized, ensuring that all deliveries are completed within acceptable time windows and without exceeding battery range.

Passenger transportation. In shuttle or ride-sharing systems using electric fleets, fairness in route assignment is crucial to avoid excessive delays for passengers. By controlling the maximum route length, service quality can be improved, since no vehicle is disproportionately delayed, and users experience more consistent waiting times.

Emergency and disaster management. In critical scenarios such as medical supply distribution or disaster relief, minimizing the longest route is often more relevant than reducing total travel distance. This ensures that all target locations receive service within the shortest possible time frame, which can be vital when dealing with perishable goods, medical emergencies, or urgent relief operations.

Surveillance and patrolling operations. In multi-vehicle surveillance or security applications, the min-max criterion ensures that all areas are covered regularly and no patrol route is excessively long. This results in more homogeneous coverage and improved reliability of the monitoring system, which is particularly important in large urban areas or industrial sites.

In all these contexts, the MEVRP offers a valuable framework for designing planning strategies that explicitly account for the characteristics of electric vehicles and the distribution of charging infrastructure. By shifting the focus from global efficiency to fairness and reliability, the model addresses key challenges in sustainable mobility and modern transportation systems.

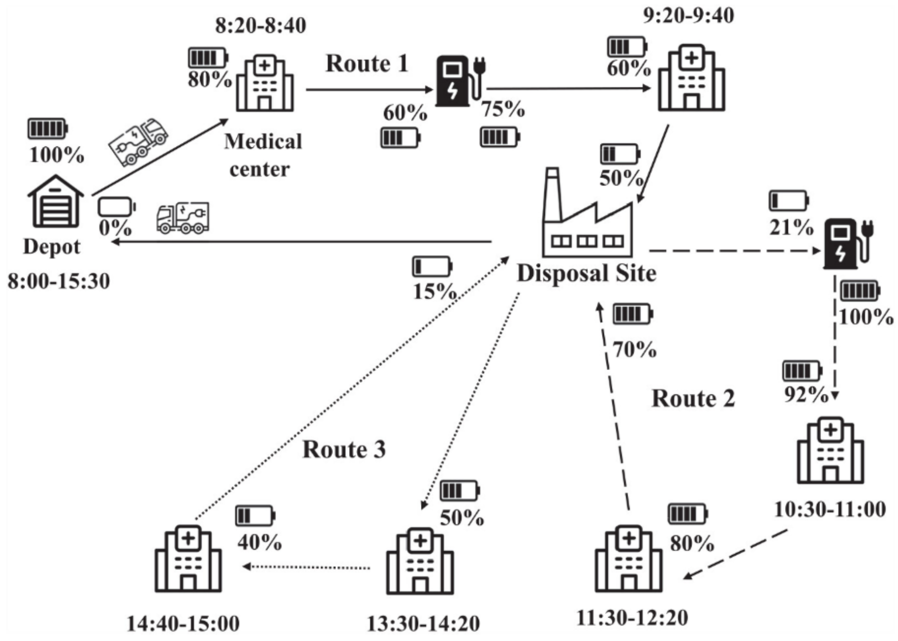

Figure 1.1: Electric vehicle routes for collecting medical waste from healthcare centers to the disposal site, with battery levels, charging points and time schedules, adapted from [3].

1.4 Methodological Approach

The analysis of the Min-Max Electric Vehicle Routing Problem (MEVRP) developed in this thesis follows a twofold methodology. In the first phase, an exact mathematical model was formulated as a mixed-integer linear programming problem and implemented in Python using the PuLP library. This formulation makes it possible to obtain optimal solutions for small-scale instances, while also serving as a reference to validate the model and study the main properties of the problem.

Given the inherent complexity of the MEVRP, which prevents exact methods from scaling to larger problem sizes, a second phase of the work relies on a heuristic approach. The proposed algorithm includes a construction phase, aimed at quickly generating feasible initial solutions, and an improvement phase, in which local refinement techniques are applied to enhance solution quality. This component was also implemented in Python, with the goal of balancing computational efficiency and solution accuracy.

Finally, the two approaches were evaluated through a computational study on instances of varying size and structure. This comparison highlights the strengths and limitations of each methodology, providing a clear picture of the trade-offs between optimality, runtime, and scalability.

Chapter 2

Literature Review

2.1 The Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is one of the most influential and widely studied problems in operations research and combinatorial optimization. First introduced by Dantzig and Ramser in 1959 [5], it can be seen as a generalization of the Traveling Salesman Problem (TSP), arising from the need to design efficient routes for a fleet of vehicles starting from a central depot to serve a set of customers with known demand.

In its general form, the VRP consists in determining a set of routes that satisfy vehicle capacity constraints and ensure that each customer is visited exactly once. The most common objective is to minimize the overall transportation cost, usually measured in terms of total distance traveled, travel time, or other relevant efficiency metrics.

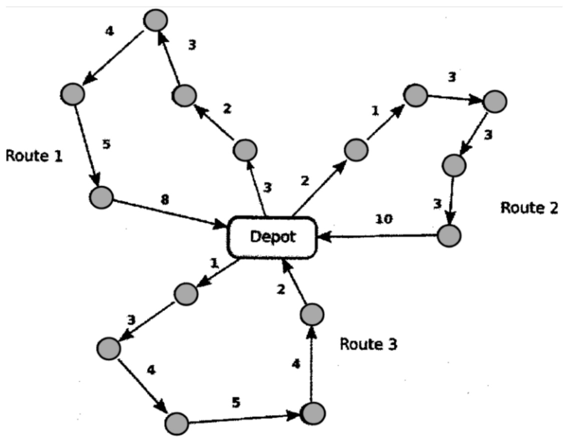

Figure 2.1: VRP graph

The mathematical formulation relies on a complete graph, where nodes correspond to the depot and customers, while edges are associated with travel costs [**Figure 2.1**]. The objective function minimizes the sum of the selected arc costs, subject to capacity and flow constraints. Despite the apparent simplicity of its definition, the VRP belongs to the class of **NP-hard (nondeterministic polynomial-time hard problem)** problems, meaning that exact approaches quickly become computationally prohibitive as the instance size increases.

Over the years, several variants of the classical VRP have been developed to reflect the complexity of real-world scenarios, such as the Capacitated VRP (CVRP), the VRP with Time Windows (VRPTW), the Multi-Depot VRP, and the Pickup and Delivery Problem (PDP) [5]. These extensions broaden the scope of applications while preserving the inherent combinatorial complexity.

The VRP has a wide range of applications, including urban freight distribution, last-mile logistics, waste collection, and the planning of public services. Optimal routing not only generates significant economic benefits but also reduces the environmental footprint of transportation systems.

The transition from the classical VRP to sustainability-oriented variants has led to the emergence of the so-called Green VRPs (**GVRP, Green Vehicle Routing Problem**) and Electric VRPs (**EVRP, Electric Vehicle Routing Problem**) [7, 6], which will be discussed in the next section.

2.2 Green VRP: EVRP and related variants

In recent years, the increasing attention to environmental sustainability has led to the development of new variants of the VRP, commonly referred to as **Green Vehicle Routing Problems (GVRP)** [7]. These models differ from the classical VRP because, in addition to minimizing operational costs, they also account for environmental factors such as reducing greenhouse gas emissions [1, 2] or incorporating alternative-fuel powered vehicles.

The literature identifies several research directions, mainly depending on the type of vehicles considered: **ICEVs (Internal Combustion Engine Vehicles)**, where the focus is on reducing fuel consumption and emissions; **AFVs (Alternative-Fuel Vehicles)**, which introduce logistical challenges related to the location of refueling stations; and **HEVs (Hybrid Electric Vehicles)**, which combine combustion and electric engines, thus mitigating the problem of limited driving range.

Among the GVRP variants, the **Electric Vehicle Routing Problem (EVRP)** plays a central role [3]. In this model, the traditional constraints of the VRP are combined with the specific features of electric vehicles: limited battery autonomy and the uneven distribution of charging stations. Route planning must therefore address not only the allocation of customers to vehicles but also the decision of whether, when, and where to recharge during the journey.

Over time, the EVRP has been extended with additional variants that increase its realism: for example, the **EVRP with time windows (EVRPTW)** [6], which imposes service time constraints for customers; models with **partial recharging**, and studies considering **mixed fleets** of EVs and conventional vehicles, aimed at analyzing transitional solutions toward more sustainable transport systems.

These extensions have made the GVRP an increasingly relevant research field, where computational complexity meets the need to design practical solutions for real-world scenarios. Along this line, the problem addressed in this thesis, the **Min–Max Electric Vehicle Routing Problem (MEVRP)**, combines the energy-related constraints of the EVRP with a min–max optimization criterion. By minimizing the maximum route length across the fleet, this approach allows for a more balanced distribution of workloads, reduces battery stress, and enhances overall service reliability.

2.3 Min–max approaches in the literature

A stream of research in vehicle routing focuses on the use of **min–max objectives**. While the classical VRP minimizes the overall routing cost (the *min–sum* criterion), the min–max approach aims at reducing the length, duration, or energy consumption of the most expensive route in the fleet. The motivation is to avoid unbalanced solutions, where one vehicle is assigned a disproportionately long or demanding route compared to the others.

This perspective is particularly relevant in applications where the maximum workload plays a critical role. Examples include urban distribution, healthcare logistics, multi-vehicle surveillance, and, most importantly, electric vehicle fleets. In these contexts, keeping the longest route under control reduces the risk of battery depletion, balances vehicle usage, and improves the reliability of the service.

From a modeling standpoint, the min–max VRP is often formulated by introducing an auxiliary variable that represents the maximum route cost, with constraints ensuring that each route does not exceed this value. This preserves the combinatorial structure of the VRP while shifting the focus towards fairness and robustness.

Several VRP variants have been studied under a min–max objective, including the capacitated VRP, the VRP with time windows, and multi-depot or heterogeneous fleet problems. In electric routing problems, the min–max criterion has a special meaning, since it directly addresses the operational limits of battery capacity and charging infrastructure [7].

Different solution methods can be found in the literature. On the one hand, **exact approaches** rely on mixed-integer linear programming and branch-and-cut methods, which can solve only small or medium-sized instances. On the other hand, **heuristic and metaheuristic approaches** adapt well-known VRP strategies—such as greedy construction, local search, tabu search, genetic algorithms, or ant colony optimization—by driving the search toward the reduction of the maximum route.

The main distinction between the **Electric Vehicle Routing Problem (EVRP)** and the **Min–Max Electric Vehicle Routing Problem (MEVRP)** lies in the optimization goal[**Figure 2.2**]:

- In the **EVRP**, the focus is still on minimizing the total cost, but with the additional constraints of electric vehicles, such as limited battery range, the need to recharge, and possibly partial recharging strategies.

- In the **MEVRP**, the same constraints are present, but the objective changes: the planner seeks to minimize the length of the longest route in the fleet. This leads to solutions that are not only energy-feasible, but also more balanced and reliable in practice.

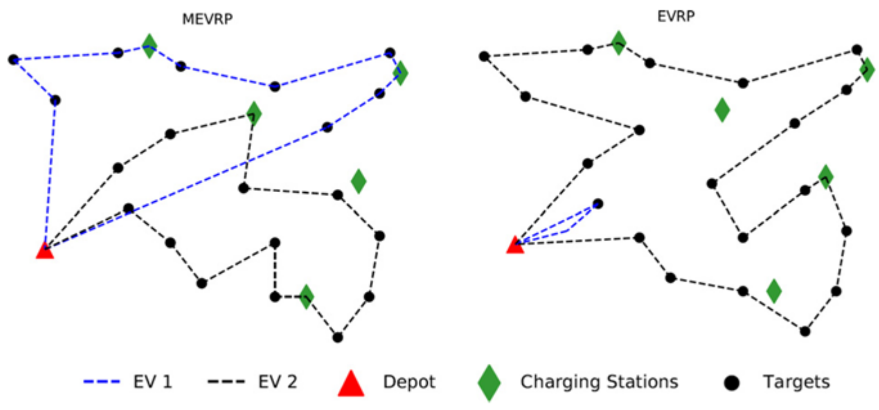

Figure 2.2: A feasible tour for two EVs visiting all the targets while visiting some charging stations for recharging: MEVRP (left) vs EVRP (right).

Overall, the evolution from EVRP to MEVRP highlights a shift from global efficiency to fairness and reliability, two aspects that are increasingly important in sustainable and real-world transportation systems.

2.4 Solution methods: exact vs heuristic/metaheuristic

Because of its combinatorial nature and computational complexity, the VRP has been tackled with a wide variety of solution strategies. Broadly speaking, these approaches fall into two categories: **exact methods**, which aim to guarantee optimal solutions, and **heuristic or metaheuristic methods**, which focus on producing good-quality solutions within reasonable computation times.

Exact methods rely on mathematical programming, in particular mixed-integer linear programming (MILP). Techniques such as *branch-and-bound*, *branch-and-cut*, and *branch-and-price* systematically explore the solution space and are able to certify optimality. These approaches are highly valuable for small- or medium-sized problems and often serve as benchmarks against which approximate methods are evaluated. However, their scalability is limited: as the number of customers grows, the computational effort increases rapidly, which makes them impractical for large real-world applications.

To address larger problem instances, the literature has proposed a wide range of **heuristics**. These are usually faster and simpler procedures, often based on constructive rules (such as the well-known Clarke & Wright savings algorithm) or on local search techniques that iteratively improve an initial solution. Heuristics do not provide guarantees of optimality, but they represent a good compromise when the goal is to find acceptable solutions within limited computing time.

Beyond these, **metaheuristics** offer more sophisticated and flexible frameworks. Algorithms such as *Tabu Search*, *Simulated Annealing*, *Genetic Algorithms*, *Ant Colony Optimization*, or *Variable Neighborhood Search (VNS)* have been widely applied to the VRP and its variants. Their main strength lies in combining diversification and intensification mechanisms, which allow them to overcome the limits of simple heuristics and to tackle very large instances. Even though they cannot guarantee the optimal solution, they usually deliver high-quality results in computation times that are compatible with real-world applications.

For electric vehicle variants such as the EVRP and the MEVRP, the challenge becomes even greater. In addition to the classical routing constraints, it is necessary to account for limited driving range, the availability of charging stations, and in some cases, partial recharging strategies. This requires either adapting existing algorithms or designing tailored ones capable of handling vehicle state of charge and integrating charging decisions directly into the routing plan.

To sum up, exact methods remain an important theoretical benchmark and are mainly useful for small problem instances, while heuristics and metaheuristics have become essential tools when dealing with realistic, large-scale cases. In the case of electric vehicles, this balance is even more delicate, since the quality of the solutions must always go hand in hand with their energy feasibility.

From this review it also becomes clear that the combination of sustainability-related constraints with min–max objectives it has been little explored so far. This is precisely the space where the **Min–Max Electric Vehicle Routing Problem (MEVRP)** fits. The next chapters focus on this problem, starting with its formal definition and then exploring both an exact model and a dedicated heuristic approach.

Chapter 3

Mathematical Model and Exact Approach

3.1 Formal Definition of the MEVRP

The *Min–Max Electric Vehicle Routing Problem (MEVRP)* is a variant of the *Electric Vehicle Routing Problem (EVRP)*, as introduced by [7], which extends the classical *Vehicle Routing Problem (VRP)* by incorporating the operational constraints specific to electric vehicles.

Unlike the conventional VRP, where the goal is to minimize the *total distance* traveled by the fleet, the MEVRP aims to *minimize the maximum distance traveled by any single vehicle*. This *min–max* objective ensures a more balanced distribution of routes among vehicles, reducing extreme workloads and promoting a fairer and more uniform utilization of the fleet. In practice, this can help mitigate battery degradation, improve reliability, and increase the overall efficiency of operations.

The problem is defined on a directed graph $G = (V, E)$, where V represents the set of nodes and E the set of arcs connecting them. The node set V is divided into three subsets:

- $D = \{d_0\}$: the **depot**, representing the starting and ending point for all vehicles;
- T: the set of **targets** (or customers) that must each be visited exactly once;
- C: the set of **charging stations** available for battery recharging during the routes.

Each arc $(i, j) \in E$ is associated with a non-negative cost c_{ij}, which is proportional to the energy or distance required to travel from node i to node j. A homogeneous fleet of electric vehicles $M = \{1, 2, \ldots, m\}$ is considered, where each vehicle has a battery capacity F_m and a linear energy consumption rate proportional to the traveled distance. All vehicles start their routes at the depot with a fully charged battery and must return to the depot after visiting their assigned targets, possibly recharging at one or more charging stations along the way.

The goal of the MEVRP is to determine:

- the subset of targets assigned to each vehicle;
- the visiting sequence of nodes (including charging stations);
- and the corresponding battery level at each point of the route;

such that:

1. every target is visited exactly once by a single vehicle;
2. each vehicle starts and ends its route at the depot;
3. no vehicle runs out of charge during its trip;
4. the maximum distance traveled by any vehicle is minimized.

Formally, the objective can be expressed as:

$$\min\; w = \max_{m\in M} \sum_{(i,j)\in E} c_{ij}\, x_{ij}^m$$

where $x_{ij}^m = 1$ if vehicle m travels along arc (i,j), and 0 otherwise. The continuous variable w represents the total distance of the longest route among all vehicles, which the model seeks to minimize.

The MEVRP therefore combines the traditional routing constraints of the VRP (flow balance, coverage, and connectivity) with the energy-related constraints of electric vehicles (battery capacity and charging requirements). This type of formulation is particularly relevant for applications where limiting the maximum service time or travel distance is more critical than minimizing total cost — for instance, in urban logistics, surveillance, emergency response, or cooperative multi-robot missions.

3.2 Sets, Parameters, and Decision Variables

To formally describe the MEVRP, the following notation is adopted, following the formulation proposed by [7]. The problem is defined on a directed graph $G = (V, E)$, where each edge $(i,j) \in E$ connects two distinct vertices $i, j \in V$. All distances and costs are assumed to satisfy the triangle inequality.

Sets

- T: set of *targets* (or customers), indexed by $t \in T$.
- $\bar{D}$: set of *charging stations*, indexed by $d \in \bar{D}$.
- $D = \bar{D} \cup \{d_0\}$: set including all charging stations and the *depot* d_0, where all vehicles start and end their routes.
- $V = T \cup D$: set of all vertices in the network (targets, charging stations, and depot).
- $E = \{(i,j) : i,j \in V, i \neq j\}$: set of directed edges connecting any two distinct nodes.
- $S \subset V$: subset of nodes used for connectivity constraints, with $\sigma^+(S) = \{(i,j) \in E : i \in S, j \notin S\}$ denoting the set of outgoing edges from S.
- $M = \{1, 2, \ldots, m\}$: set of *electric vehicles* (EVs), each initially stationed at the base depot d_0, indexed by $m \in M$.

Parameters

- c_{ij}: cost (or distance) associated with traveling an edge $(i,j) \in E$.
- f_{ij}^m: amount of energy consumed by vehicle m when traveling from node i to node j.
- F_m: battery capacity of vehicle m.
- q: a large constant, typically set equal to the number of targets $|T|$, used to activate or deactivate binary constraints.

Decision Variables

- $x_{ij}^m = \begin{cases} 1, & \text{if edge } (i,j) \text{ is traversed by vehicle } m, \\ 0, & \text{otherwise.} \end{cases}$
- z_{ij}^m: continuous variable representing the amount of energy consumed by vehicle m along edge $(i,j) \in E$.
- $y_d^m = \begin{cases} 1, & \text{if charging station } d \in D \text{ is visited by vehicle } m, \\ 0, & \text{otherwise.} \end{cases}$
- w: continuous variable representing the *maximum distance (or energy consumption)* traveled among all vehicles, which is minimized by the model.

These sets, parameters, and variables provide the formal foundation of the mixed-integer linear model presented in Section 3.3. Together, they describe both the routing and energy-related aspects of the MEVRP, enabling an integrated optimization of vehicle assignments, routes, and battery usage.

3.3 Mathematical Model

The MEVRP can be formulated as a mixed-integer linear programming (MILP) model, following the mathematical formulation proposed by [7].

Formulation

$$\min w \tag{1}$$

$$\text{s.t.} \quad w \geq \sum_{(i,j)\in E} c_{ij} x_{ij}^m \qquad \forall m \in M \tag{2}$$

$$\sum_{i\in V} x_{di}^m = \sum_{i\in V} x_{id}^m \qquad \forall d \in \bar{D},\ m \in M \tag{3}$$

$$\sum_{i\in V} x_{di}^m \leq q\, y_d^m \qquad \forall d \in \bar{D},\ m \in M \tag{4}$$

$$\sum_{i\in V} x_{id_0}^m = 1, \quad \sum_{i\in V} x_{d_0 i}^m = 1 \qquad \forall m \in M \tag{5}$$

$$\sum_{i\in V}\sum_{m\in M} x_{ij}^m = 1, \quad \sum_{i\in V}\sum_{m\in M} x_{ji}^m = 1 \qquad \forall j \in T \tag{6}$$

$$x^m(\sigma^+(S)) \geq y_d^m \qquad \begin{aligned} &\forall d \in S \cap \bar{D},\\ &S \subset V \setminus \{d_0\},\\ &S \cap \bar{D} \neq \emptyset,\\ &m \in M \end{aligned} \tag{7}$$

$$\sum_{j\in V} z_{ij}^m - \sum_{j\in V} z_{ji}^m = \sum_{j\in V} f_{ij}^m x_{ij}^m \qquad \forall i \in T,\ m \in M \tag{8}$$

$$z_{ij}^m \leq F_m x_{ij}^m \qquad \forall (i,j) \in E,\ m \in M \tag{9}$$

$$z_{di}^m = f_{di}^m x_{di}^m \qquad \forall i \in T,\ d \in D,\ m \in M \tag{10}$$

$$x_{ij}^m \in \{0,1\}, \quad y_d^m \in \{0,1\}, \quad z_{ij}^m \geq 0 \qquad \forall (i,j) \in E,\ d \in \bar{D},\ m \in M \tag{11–12}$$

Model Interpretation

- Objective (1) minimizes w, the maximum travel distance among all vehicles.
- Constraint (2) ensures that w is at least as large as the distance traveled by each individual vehicle.
- Constraint (3) enforces flow conservation at charging stations: every arrival must correspond to a departure.
- Constraint (4) activates the binary variable y_d^m if and only if vehicle m visits charging station d.
- Constraint (5) guarantees that each vehicle starts and ends its route at the depot.
- Constraint (6) ensures that every target is visited exactly once and by a single vehicle.
- Constraint (7) provides connectivity and eliminates subtours using cut-set inequalities, written explicitly as in the original formulation.
- Constraint (8) defines the energy flow variable z_{ij}^m, which tracks battery consumption along routes.
- Constraint (9) enforces the maximum battery capacity for each vehicle.
- Constraint (10) models recharging: when a vehicle leaves a charging station, its battery is reset to full capacity.
- Constraints (11–12) define the variable domains: routing and charging variables are binary, while energy flow variables are continuous and non-negative.

This formulation integrates routing and energy constraints into a single optimization framework. It guarantees feasibility with respect to vehicle autonomy and allows a fair workload distribution across the fleet by minimizing the longest route.

3.4 Implementation in Python using PuLP

The mathematical model of the *Min–Max Electric Vehicle Routing Problem (MEVRP)*, described in the previous sections, was implemented in Python using the open-source library `PuLP` [8]. This library provides a high-level modeling interface for linear and integer programming problems, allowing the direct definition of sets, variables, constraints, and the objective function within a flexible and intuitive programming environment.

The sets, parameters, and decision variables introduced in Section 3.2 were represented in Python using lists and dictionaries. The problem instance was created through the `LpProblem` class, specifying a minimization objective. The binary variables x_{ij}^m and y_c^m, together with the continuous variables z_{ij}^m and w, were defined using the function `LpVariable.dicts()`. The objective function and the constraints were then added iteratively, following the structure of the mathematical formulation presented in Section 3.3.

A simplified code excerpt is shown below:

Listing 3.1: Simplified code excerpt for the MEVRP implementation in PuLP.

```
from pulp import *

# Decision variables
x = LpVariable.dicts("x", ((i, j, m) for i, j in Edges for m in
    Electric_Vehicles), cat="Binary")
z = LpVariable.dicts("z", ((i, j, m) for i, j in Edges for m in
    Electric_Vehicles), lowBound=0)
y = LpVariable.dicts("y", ((d, m) for d in Charging_stations for m in
      Electric_Vehicles), cat="Binary")
w = LpVariable("w", lowBound=0)

# Minimization model
problem = LpProblem("MEVRP", LpMinimize)

# Objective
problem += w
```

The model was solved directly within `PyCharm`, a Python development environment, by calling the `model.solve()` command. When no external solver is specified, `PuLP` automatically uses the open-source solver `CBC` (*Coin-or Branch and Cut*), which is integrated by default within the library. CBC performs the actual optimization process: it takes the MILP formulation built by PuLP and applies a branch-and-cut algorithm to find the optimal solution. This approach allows binary and continuous variables to be handled efficiently, guaranteeing exactness for small- and medium-sized instances.

During the testing phase, a few basic solver parameters were configured, such as the maximum computation time and the verbosity level of the output, in order to monitor the optimization process and evaluate solver convergence.

After the optimization, the values of the decision variables were retrieved directly from the PuLP objects. The routes of each vehicle were reconstructed by analyzing the binary variables x_{ij}^m, while the energy consumption along each edge was obtained from the corresponding z_{ij}^m values. The results were then used to verify the energy feasibility of the solutions and to analyze the routes returned by the model through the printed distances, paths, and energy consumption data. This information also served as the basis for the comparative evaluation between the exact method and the heuristic approach described in the next chapter.

The complete Python implementation, including input data management, constraint generation, and result analysis functions, is provided in **Appendix A**.

3.5 Exact Method: Branch and Cut

The mathematical model described in Section 3.3 was solved using an exact approach based on the *branch-and-cut* algorithm. This method combines the *branch-and-bound* strategy with the generation of additional valid inequalities (known as "cuts"), which progressively tighten the feasible region of the problem. By doing so, the algorithm ensures that the global optimum can be identified whenever full convergence is achieved.

In the context of the MEVRP, the problem formulation involves an exponential number of connectivity constraints, required to guarantee that each route forms a single continuous path between the depot, the charging stations, and the assigned targets. Including all these constraints explicitly at the beginning would be computationally inefficient. As proposed by [7], this issue can be addressed by adopting a dynamic strategy: some constraints are initially relaxed and then added only when they are violated by an intermediate solution. This mechanism, typical of branch-and-cut algorithms, allows the model to be solved more efficiently, reducing both memory usage and computation time while still ensuring optimality.

During the optimization phase, the solver gradually explores different versions of the problem, each characterized by additional constraints that restrict the search space (*branching*). At the same time, when the solver detects solutions that do not fully satisfy the integrity or connectivity conditions, it automatically introduces new inequalities (*cuts*) to exclude them. In this way, the model is progressively refined until the algorithm isolates the truly optimal solution.

In this work, the model implemented in `PuLP` was solved using the open-source solver `CBC` (*Coin-or Branch and Cut*), which internally manages both the branching process and the generation of cuts. During computation, the solver reports intermediate information such as the best feasible solution found so far, the current lower bound, and the corresponding optimality gap. If the solver terminates with status `Optimal`, the gap is equal to zero; otherwise, the reported value represents the percentage difference between the best feasible solution and the best known theoretical bound. This metric provides a direct measure of how close the current solution is to the true optimum.

The Python implementation employs CBC through the PuLP interface, as shown in the following code excerpt:

Listing 3.2: Implementation of the exact method with CBC and time limit check.

```python
from pulp import PULP_CBC_CMD, LpStatus, value
import time, os, re

# CBC solver with time limit and log file
solver = PULP_CBC_CMD(timeLimit=300, logPath="cbc_log.txt", msg=False
    )

t0 = time.perf_counter()
status = problem.solve(solver)
solve_time = time.perf_counter() - t0

# Check the solver log for time limit
log_text = open("cbc_log.txt", "r", errors="ignore").read()
hit_time_limit = bool(re.search(r"(Stopped on time limit|Exiting on
    maximum time)", log_text, re.I))
derived_status = "TimeLimit" if hit_time_limit else LpStatus[problem.
    status]

# Report main solver results
print(f"Solver status: {derived_status}")
print(f"Objective value (w): {value(problem.objective):.3f}")
print(f"Runtime: {solve_time:.2f} seconds")
```

The branch-and-cut algorithm therefore provides an exact solution framework for the MEVRP, capable of obtaining provably optimal solutions for small and medium-sized problem instances. However, due to the combinatorial complexity and the exponential growth of the search tree, this approach becomes computationally demanding for larger instances a limitation that motivates the development of a heuristic method, which is discussed in the next section.

3.6 Computational Results on Small Instances

To evaluate the performance of the mathematical model and the exact solution approach, a series of computational experiments were conducted on small-scale instances of the MEVRP. The test instances were generated using a custom Python function, `generate_instance()`, which automatically creates the coordinates of the depot, targets, and charging stations within a predefined grid. In all experiments, the battery capacity was fixed at $F = 100$, and the energy consumption rate per unit distance was kept constant at 0.8.

3.6.1 Instance Generation

For each configuration — that is, for each number of targets — five independent instances were generated using the `generate_instance()` function. The grid was defined as a 100×100 area, with the depot fixed at the center position $(50, 50)$. The targets and charging stations were randomly distributed within this grid to simulate different spatial configurations while maintaining comparable scales across instances. The battery capacity ($F = 100$) and the energy consumption rate per unit distance (0.8) were kept constant throughout all experiments.

For each configuration, the reported values correspond to the average objective value ($F.O.$), runtime, and optimality gap computed over five independent runs. A maximum time limit of 300 seconds was imposed for each execution. In larger instances, the solver reached this limit, stopping the search before completing the optimization process.

The complete datasets containing the detailed results of all individual runs are provided in **Appendix C**, while Table 3.1 below summarizes the average results for each configuration.

3.6.2 Numerical Results and Discussion

Table 3.1: Average results of the exact method (CBC solver) with a time limit of 300 seconds.

Instance Parameters			**Average Results**		
n_{target}	n_{cs}	n_{evs}	Avg. F.O.	Avg. Runtime (s)	Avg. Gap (%)
5	2	2	124.72	2.16	0.00
6	2	2	176.87	33.02	0.00
7	2	2	166.99	158.87	7.02
8	2	2	168.39	176.87	10.70
9	2	3	147.10	300.00	31.72

The results show that for smaller instances (up to six targets), the model consistently achieves the optimal solution within very short computation times. As the problem size increases, the runtime grows rapidly, reaching the 300-second time limit for instances with eight and nine targets. In these cases, the solver terminates with a `TimeLimit` status and reports a positive *gap*, representing the percentage difference between the best feasible solution found and the best known lower bound on the optimal value. In simpler terms, the gap measures how far the current solution is from the true optimum: a gap of zero indicates optimality, while positive values imply that the solver stopped before full convergence.

The increasing runtime and gap values are consistent with the combinatorial complexity of the problem, as the number of variables and constraints grows rapidly with the number of targets. Nevertheless, the exact method demonstrates strong computational performance, providing optimal solutions for small instances and near-optimal ones for moderately sized problems.

For larger instances, where computation times become prohibitive and full optimality can no longer be guaranteed, an alternative approach was adopted. The next chapter introduces a **heuristic strategy** designed to obtain high-quality solutions in significantly shorter computation times.

Chapter 4

Heuristic Approach

4.1 Motivation

Even for medium-sized instances, the exact approach requires considerable computation time or fails to find a solution within the imposed time limit. This limitation motivates the development of a **heuristic method**, capable of producing good-quality solutions within much shorter computational times.

The goal is not to completely replace the exact model, but rather to **complement** it: the exact formulation provides an optimal reference for small-scale instances, while the heuristic allows us to tackle **larger and more realistic scenarios**, where exact resolution becomes impractical.

From an operational perspective, the development of a dedicated heuristic addresses three main needs:

- **Scalability** – The algorithm must maintain reasonable computation times as the number of targets, vehicles, and charging stations increases.

- **Energy feasibility** – Every solution must respect the vehicle battery constraints and ensure that it is always possible to reach either a charging station or the depot.

- **Solution quality** – The resulting routes should be well-balanced in length, keeping the maximum route distance (min–max objective) as low as possible.

In this sense, the heuristic represents an effective compromise between **accuracy** and **computational efficiency**, providing feasible solutions in short times and making the approach applicable to operational contexts where rapid decision-making is essential.

4.2 Algorithm Description

The heuristic developed for the Min–Max Electric Vehicle Routing Problem is designed to efficiently produce feasible and high-quality solutions within short computational times. The algorithm follows a two-phase structure: a *construction phase*, where an initial feasible solution is generated, and an *improvement phase*, which iteratively refines that solution through local search operations. This design reflects classical heuristic frameworks commonly employed in Vehicle Routing Problems (VRPs), where greedy initialization and neighborhood-based improvement moves are widely adopted [9].

The overall objective is to minimize the maximum route length among all vehicles (the min–max criterion), while ensuring that every route remains feasible with respect to the vehicles' energy autonomy and recharging constraints.

4.2.1 Construction Phase

The problem instances used in this phase are generated through the `generator.py` module (introduced in Chapter 3), which creates the spatial configuration of targets, charging stations, and the depot within a predefined grid. The same parameter settings are used as in the mathematical model — in particular, battery capacity and energy consumption rate per unit distance are kept fixed — in order to guarantee full consistency between the exact and heuristic analyses.

The construction phase aims to build a feasible initial solution that respects all energy constraints. The approach adopted is *greedy*: targets are considered one by one and inserted into the routes of the available vehicles following a locally optimal criterion.

The procedure can be summarized as follows:

1. All targets are sorted in increasing order of their distance from the depot.
2. Each target is then assigned to one of the available vehicles, starting from the one whose current route has the shortest total length.
3. After each tentative insertion, the algorithm checks whether the route remains feasible in terms of battery autonomy. If the vehicle cannot reach the next node or return to a charging station, one or more charging stations are automatically inserted in the most suitable positions.
4. If no feasible assignment exists for a given target, the instance is declared *infeasible.*
5. Otherwise, the process continues until all targets are served, producing a fully feasible initial solution that satisfies both routing and energy constraints.

This phase ensures that all vehicles can complete their assigned routes while respecting the battery limitations, and provides the starting configuration for the subsequent improvement phase.

4.2.2 Improvement Phase

Starting from the feasible initial solution, the algorithm performs an iterative improvement process to reduce the objective value (i.e., the maximum route length). At each iteration, two vehicles are randomly selected, and one of two local search moves is applied:

- **Relocate:** a target is moved from one route to another (or to a different position within the same route);
- **Swap:** two targets belonging to different vehicles exchange their positions.

After each move, the affected routes are reconstructed and checked again for energy feasibility. A move is accepted only if:

1. the resulting solution remains feasible under the energy constraints, and
2. the maximum route length is reduced compared to the previous solution.

This iterative process continues until no further improvement is found or a termination condition is met.

4.2.3 Remarks

The proposed heuristic provides an effective balance between solution quality and computational efficiency, combining a constructive and an improvement phase within a unified framework. Its design ensures that all intermediate solutions remain feasible, as the algorithm automatically introduces charging stations or depot returns whenever required by the vehicle's energy state. This integration of feasibility control within the search process allows the method to achieve robust and realistic solutions in very short computation times.

Building upon this conceptual structure, the following section discusses the Python implementation of the heuristic, outlining its key components and the main design choices adopted during development.

4.3 Python Implementation

The heuristic developed for the *Min–Max Electric Vehicle Routing Problem* was entirely implemented in Python, following a modular structure that reflects the two main phases described previously: the construction phase and the improvement phase.

The implementation is based on two main scripts:

- `generator.py`, used to generate test instances by defining the positions of targets, charging stations, and the depot on a predefined grid;
- `heuristic.py`, which contains the full implementation of the heuristic algorithm, including the construction of the initial solution and the subsequent improvement phase based on local search moves.

The main function, `heuristic(data, patience, p_swap)`, takes as input the instance generated by `generator.py` and returns the final routes along with the corresponding objective value. Its execution is supported by several auxiliary functions, including:

- `enforce_battery_strict()`, which checks the energy feasibility of each route and automatically inserts charging stations whenever the battery level becomes insufficient;
- `build_feasible_start()`, which generates the initial solution following a *greedy* logic, assigning targets to vehicles based on their current route lengths;
- `dist()`, `route_length()` and `obj()`, used for computing distances and evaluating the objective function.

Two main parameters control the behavior of the improvement phase:

- `patience`, which specifies the maximum number of consecutive non-improving iterations before the algorithm stops;
- `p_swap`, which defines the probability of performing a *swap* move instead of a *relocate* move at each iteration.

The implementation also includes an option for printing detailed distance and energy consumption summaries, which can be useful for verification and debugging purposes.

At the end of each execution, the algorithm provides a detailed report that includes:

- the initial and final solutions produced by the heuristic;
- the total length of each vehicle's route;
- the objective value (maximum route length);

- the total runtime.

The modular structure of the code allows easy testing on multiple instances while maintaining full consistency with the parameters used in the mathematical model. The complete source code of the heuristic, including the auxiliary functions for feasibility checking and energy management, is provided in **Appendix D**.

4.4 Performance on Test Instances

To assess the performance of the developed heuristic, a series of experiments was carried out on problem instances of increasing size, with the number of targets ranging from 8 to 240. This gradual increase was designed with a dual purpose: first, to observe the behavior of the heuristic on small and medium-sized problems, and second, to analyze how its efficiency and stability evolved with problem complexity. The upper limit of 240 targets was chosen because, beyond this size, the total runtime started to exceed 300 seconds — the same time limit used for the mathematical model discussed in Chapter 3.

Each configuration was tested on five independent instances generated using the same procedure described earlier, ensuring methodological consistency between the exact and heuristic approaches. The reported values represent the average results across the five runs for each case. All experiments were conducted with constant energy parameters: a battery capacity $F = 100$ and an energy consumption rate of 0.8 per unit of distance. The `patience` parameter was set to 100,000 iterations, a value found sufficient to guarantee stable convergence of the local search phase.

Table 4.1: Heuristic performance on test instances (average over 5 runs per case).

#Targets	#CS	#EVs	Initial F.O.	Final F.O.	Runtime (s)
8	3	2	335.79	263.41	6.54
15	4	3	352.34	254.36	7.42
30	6	6	409.84	187.92	9.25
50	8	10	374.59	182.43	14.76
75	13	15	358.13	163.16	23.19
100	20	20	385.84	162.86	32.88
200	40	40	367.84	145.55	168.29
240	40	45	384.01	144.39	324.87

The results presented in Table 4.1 represent average values computed over five independent runs for each configuration. The complete datasets and numerical tables used for these averages, including the individual results for each test instance, are reported in Appendix E.

The results show that the heuristic consistently improves the initial greedy solution while maintaining reasonable computational times. For small instances (8–15 targets), the average improvement ranges between 20% and 30%, with runtimes below 10 seconds. As the problem size increases, the algorithm becomes more effective: for medium-sized instances (30–100 targets), the improvement grows to between 50% and 60%, with execution times remaining under one minute. Even for large instances (200–240 targets), the heuristic maintains stable behavior, achieving improvements above 60% with an average runtime of approximately five minutes.

These findings demonstrate the *scalability* of the heuristic: computation times increase almost linearly with instance size, while solution quality remains high. The algorithm's strength lies in the combination of a fast constructive phase and an efficient local search mechanism that progressively balances the workload among vehicles, reducing the maximum route length (min–max objective) at each iteration.

From the energy perspective, all obtained solutions are fully feasible. The function `enforce_battery_strict()` effectively ensures that each vehicle always has an energy-sustainable route. In particular, when the remaining battery level is insufficient to reach the next target — or when, after reaching it, the vehicle would not have enough energy to continue towards a charging station or return to the depot — the algorithm automatically selects, among the existing charging stations, the nearest reachable one and dynamically inserts it into the route. This "look-ahead" mechanism guarantees that every vehicle can complete its route without exhausting its battery, closely reflecting realistic operational conditions.

Another important observation concerns the *stability of the results*. The variability between different instances of the same configuration is minimal, confirming the reliability of both the constructive and improvement phases. Even when changing the random seed, the heuristic consistently produces solutions of comparable quality, indicating robust performance.

Overall, the heuristic proves to be an *effective, scalable, and reliable* approach, capable of delivering high-quality, energy-feasible solutions within short computation times. While it does not guarantee global optimality, it provides a strong practical alternative to the exact model for medium-to-large instances, where mathematical optimization becomes computationally expensive. A more detailed quantitative comparison between the two approaches is presented in Chapter 5.

Chapter 5

Computational Results and Comparison

5.1 Overview of the Comparative Analysis

This chapter presents a comparative analysis between the exact model and the heuristic algorithm developed for the *Min–Max Electric Vehicle Routing Problem* (MEVRP). While the previous chapters examined each approach separately, focusing respectively on the formulation and optimality of the mathematical model and on the efficiency of the heuristic procedure, the goal here is to evaluate their performance side by side on a common set of instances.

The purpose of this comparison is twofold: first, to measure how close the heuristic solutions are to the optimal (or best-known) ones obtained through exact optimization; and second, to quantify the computational savings achieved by the heuristic approach. This makes it possible to assess not only the quality of the results, but also the practical efficiency of both methods when applied under comparable conditions.

To ensure a fair and consistent comparison, both algorithms were tested on **identical problem instances**, generated through a controlled random process. A fixed initialization parameter (*seed*) was introduced in the instance generator so that each test configuration could be reproduced exactly. This allows the analysis to focus purely on the algorithmic differences rather than on random variations in the spatial distribution of targets or charging stations.

5.2 Experimental Setup and Instance Generation

To make the comparison between the two approaches as fair and consistent as possible, a controlled mechanism was introduced in the data generation process. Both the exact model and the heuristic algorithm were tested on the same instances, created in a deterministic way by fixing a random seed. This ensures that the two methods start from identical spatial configurations and that any observed difference in performance depends only on the algorithmic behavior rather than on random variations in the input data.

In the instance generator (`generator.py`), an optional argument was added to allow control over the random seed used by Python's pseudo-random number generator. The implementation is shown below:

Listing 5.1: Implementation of the seed parameter in the instance generator.

```python
def generate_instance(n_targets=8, n_cs=3, n_evs=2, grid_size=100,
    seed=None):
    if seed is not None:
        random.seed(seed)
```

When a value is specified for the `seed` parameter, the spatial distribution of all nodes (depot, targets, and charging stations) is exactly reproduced in every execution. In practice, this means that both algorithms face the same instance layout, which allows for a direct and meaningful comparison between their results.

During testing, different seed values were used to generate distinct yet reproducible scenarios. In both models, the seed value was simply passed as an argument when creating the instance, as shown below:

Listing 5.2: Example of instance generation with a fixed seed.

```python
data = generate_instance(n_targets=8, n_cs=3, n_evs=2, grid_size=100,
    seed=60)
```

This setup made it possible to perform a one-to-one comparison between the heuristic and the exact model on the same data, ensuring full reproducibility of the experiments.

The test instances were created with an increasing number of targets (from 6 to 9), while the number of charging stations and electric vehicles was either kept constant or slightly varied. This configuration offered a good balance between problem complexity and computational effort: small enough for the exact model to find or approximate the optimal solution within the given time limit, but large enough to highlight the performance differences between the two approaches.

The exact model was solved using the *CBC* solver within *PuLP*, with a time limit of 300 seconds per instance. The heuristic was executed with a number of iterations (`patience`) set to 100 000. This configuration was chosen to ensure convergence and stability in the obtained solutions. However, additional tests were carried out by progressively reducing the number of iterations, in order to assess whether comparable results could be achieved in shorter computation times.

5.3 Results and Comparison

To directly compare the performance of the exact model and the developed heuristic, a representative test case was selected among those analyzed. Specifically, the results refer to the instance generated with `seed = 60`, while keeping constant the number of electric vehicles ($n_{evs} = 2$) and charging stations ($n_{cs} = 3$). This configuration makes it possible to evaluate how both approaches behave as the problem size increases, while maintaining the same spatial distribution of nodes.

Tables 5.1 and 5.2 report, respectively, the results obtained with the *exact model* and the *heuristic*, as the number of targets increases from 6 to 9. For each case, the table lists the objective value (expressed as the maximum route length), the execution time, and the percentage gap computed with respect to the lower bound (*best bound*) returned by the solver.

The *best bound* represents the best theoretical estimate of the optimum available at the termination of the optimization process. When the solver reaches the predefined time limit, the reported gap indicates the relative distance between the best feasible solution found and this lower bound. Therefore, the gap does not necessarily represent the distance from the true optimum of the problem but provides a quantitative measure of the solution quality with respect to the best theoretical approximation known.

Table 5.1: Exact model results for seed = 60 ($n_{evs} = 2$, $n_{cs} = 3$)

Targets	**Objective Value**	**Runtime (s)**	**Gap (%)**
6	126.03	22.97	0.00
7	138.14	121.65	0.00
8	167.70	300.00	24.27
9	179.54	300.00	37.78

Table 5.2: Heuristic results for seed = 60 ($n_{evs} = 2$, $n_{cs} = 3$)

Targets	**Objective Value**	**Runtime (s)**	**Gap (%)**
6	163.37	4.45	22.86
7	167.74	4.54	17.65
8	186.95	5.36	32.07
9	202.42	6.10	44.81

For small instances (6–7 targets), the *exact model* is able to determine the optimal solution within reasonable times. However, as the number of nodes increases, the computational time grows rapidly, reaching the maximum limit of 300 seconds with residual optimality gaps above 35%. The *heuristic*, on the other hand, maintains extremely low computation times—always below 7 seconds—and produces feasible solutions of good quality, with objective values close to those of the exact model and gaps that remain within an acceptable range. In particular, the heuristic gap varies between approximately 17% and 45%, increasing proportionally with the instance complexity.

The *heuristic*, on the other hand, maintains extremely low computation times—always below 7 seconds—and produces feasible solutions of good quality, with objective values close to those of the exact model and gaps that remain within an acceptable range. In particular, the heuristic gap varies between approximately 17% and 45%, increasing proportionally with the instance complexity.

To better illustrate these results, Figures 5.1 and 5.2 show how the objective value and the runtime evolve as the number of targets increases, for the same instance configuration (seed = 60, $n_{evs} = 2$, $n_{cs} = 3$). The graphical trends confirm that the heuristic consistently achieves near-optimal solutions in a fraction of the computation time required by the exact model.

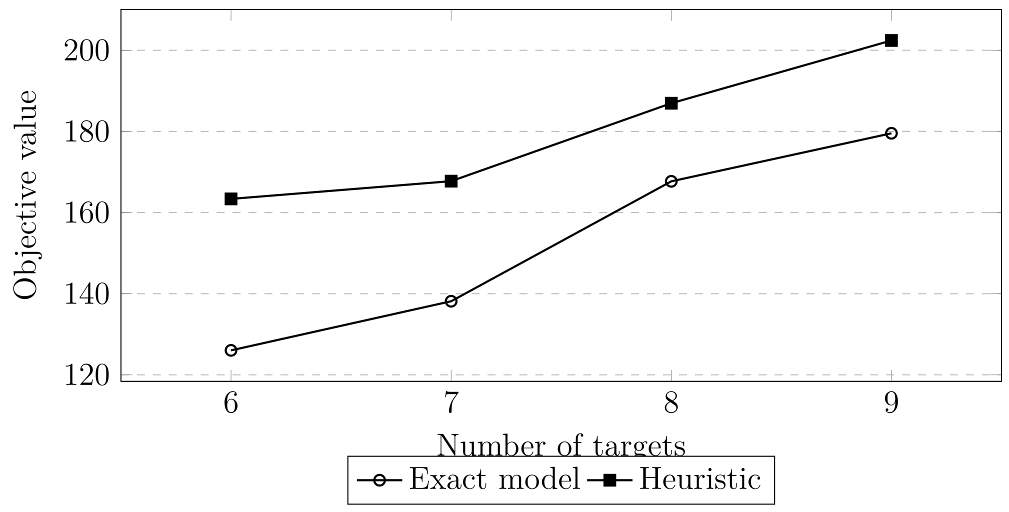

Figure 5.1: Objective value vs number of targets for the exact model and heuristic (seed = 60).

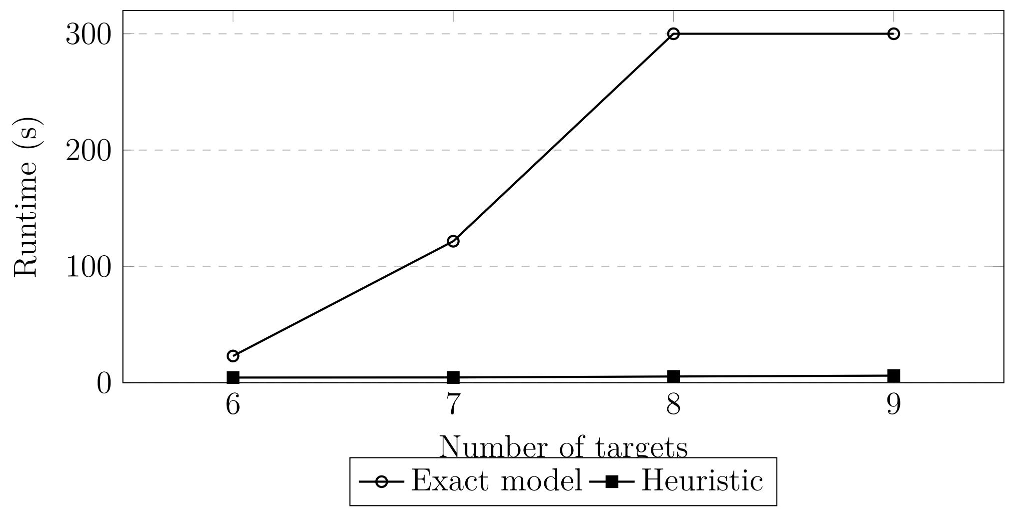

Figure 5.2: Runtime vs number of targets for the exact model and heuristic (seed = 60).

These results confirm the complementary nature of the two approaches: the exact method guarantees optimality but at the cost of significantly higher runtimes, whereas the heuristic provides near-optimal solutions in a fraction of the time. In practical applications, where rapid response is often more important than absolute optimality, the heuristic therefore represents an effective and well-balanced strategy.

5.4 Effect of the Iteration Budget on the Heuristic

To assess the efficiency of the heuristic, an additional analysis was carried out by varying the number of iterations (`patience`), which defines the maximum number of consecutive non-improving moves before the algorithm stops. In the previous experiments, this parameter was set to 100 000, allowing for a broad exploration of the solution space and highly stable results.

During the testing phase, further experiments were performed by modifying the number of iterations, which can be adjusted according to the complexity of the instance under consideration. It was observed that, in several cases, the heuristic achieved the **same objective values** reported earlier even with a considerably smaller number of iterations. In particular, when the limit was reduced to 10 000, the execution time decreased from about 4–6 seconds to approximately 0.4–0.5 seconds, without any deterioration in solution quality.

This result demonstrates that the heuristic converges very quickly toward high-quality solutions, even with a reduced iteration budget. For small and medium-sized instances, therefore, a lower `patience` value is sufficient to obtain stable results, while higher values may be useful only for more complex problems. More generally, by tuning the number of iterations according to the problem context and verifying that no further improvements are achieved beyond a certain threshold, it is possible to identify an **optimal balance between accuracy and computation time**, thus creating an additional opportunity for performance optimization.

Overall, these findings confirm the robustness and computational efficiency of the proposed approach, which enables an effective trade-off between solution quality and runtime through a simple adjustment of the iteration parameter. Looking ahead, further improvements to the heuristic could focus on more flexible construction strategies or enhanced local search operators, in order to further reduce the gap with respect to the exact solutions. These aspects will be discussed in more detail in the final chapter.

Chapter 6

Conclusions

6.1 General Overview

This thesis addressed the optimization of electric vehicle routing problems, with the objective of identifying efficient solutions that comply with both energy and operational constraints. To this end, a dual solution strategy was developed: an **exact mixed-integer linear programming (MILP) formulation** and a **dedicated heuristic procedure**, designed to extend the applicability of the method to larger and more complex instances.

The exact approach allowed the computation of optimal or near-optimal solutions for small-scale problems, providing an important benchmark for the experimental phase. However, as the problem size increased, the combinatorial complexity and the energy-related constraints made the exact model computationally demanding, confirming the need for more scalable approaches.

In this perspective, the proposed heuristic represented an **effective compromise between solution quality and computational time**, being able to produce good-quality solutions within a few seconds, even for medium-sized instances. Its modular structure and the possibility to tune the search parameters also enabled stable performance and flexible control over computational efficiency.

Overall, the work demonstrated that a balanced integration between exact and heuristic approaches can provide a solid and practical framework for tackling complex routing problems, while preserving both modeling rigor and operational efficiency.

6.2 Heuristic Improvements and Further Refinements

Although the proposed heuristic has proven to be effective in providing feasible and high-quality solutions within short computational times, several aspects could be further improved to enhance its performance and robustness.

A first direction concerns the **adaptive management of search parameters**. In the current implementation, parameters such as the maximum number of iterations or the `patience` threshold are manually set before execution. Although tuning these values already allows control over the trade-off between exploration and exploitation, introducing **adaptive mechanisms** capable of dynamically adjusting them according to the convergence behaviour could further improve efficiency. For instance, the algorithm could automatically increase the patience threshold when progress slows down, or reduce it once the solution stabilizes, thereby achieving a more balanced and responsive search process.

A second opportunity for improvement lies in the **integration of local search and metaheuristic operators**. Embedding techniques such as Tabu Search, Simulated Annealing, or Variable Neighborhood Search could strengthen the ability of the heuristic to escape local minima and explore the solution space more effectively, especially in large-scale or highly constrained instances.

From a computational standpoint, future developments may also involve the **parallelization of the algorithm**, allowing multiple search processes or initial configurations to be evaluated simultaneously. This strategy would significantly reduce the overall computation time while maintaining a wide diversity of explored solutions, thus improving both efficiency and robustness. Alternatively, the heuristic could be embedded within a **hybrid framework** that combines exact and heuristic components, exploiting the precision of the exact model for partial evaluations while preserving the flexibility and speed of the heuristic search.

Finally, applying the method to **real-world case studies**—for instance, involving real energy consumption data, heterogeneous vehicles, or time-dependent charging constraints—would allow for a more comprehensive assessment of the heuristic's practical applicability and scalability.

6.3 Broader Perspectives

The research presented in this thesis contributes to the wider field of sustainable transportation and electric mobility, offering both methodological and practical insights into the optimization of energy-constrained vehicle routing problems. By combining the rigor of an exact mathematical formulation with the flexibility of a heuristic approach, this work has shown how hybrid and computationally efficient methods can effectively deal with the increasing complexity of modern logistics systems.

Beyond its methodological contribution, the study underlines the value of heuristic approaches as practical decision-support tools, especially in real-world contexts where time efficiency and feasibility often matter more than formal optimality. In this sense, the proposed framework can be seen as a step towards bridging the gap between theoretical optimization and operational planning, supporting the deployment of electric fleets and energy-aware logistics strategies.

Looking to the future, the continuous evolution of electric vehicle technology, charging infrastructure, and data availability will open new opportunities for research and application. The methods developed in this work may serve as a foundation for more adaptive and data-driven optimization models, capable of supporting smarter, more sustainable, and energy-efficient routing solutions in the years to come.

Appendix A

Exact Model Implementation in Python

The following listing shows the complete Python implementation of the exact optimization model for the Min–Max Electric Vehicle Routing Problem (MEVRP). The model was formulated and solved using the `PuLP` library, based on the data generated by the `generator.py` function. It includes all decision variables, constraints, and post-solution reporting procedures.

Listing A.1: Python implementation of the exact MEVRP optimization model.

```python
from pulp import *
from generator import generate_instance

# data instance generation
data = generate_instance(n_targets=9, n_cs=2, n_evs=3, grid_size=100)

Targets = data["Targets"]
Charging_stations = data["Charging_stations"]
depot = data["Depot"]
Electric_Vehicles = data["Electric_Vehicles"]
Battery_capacity = data["Battery_capacity"]
consumption_value = data["Consumption_rate"]
costs = data["Costs"]

# derived sets
D = Charging_stations + [depot]
Vertices = Targets + Charging_stations + [depot]
Edges = [(i, j) for i in Vertices for j in Vertices if i != j]

# energy on arc (i,j) for EV m = rate * distance(i,j)
consumption_rate = {
    (i, j, m): consumption_value * costs[i, j]
    for i, j in Edges for m in Electric_Vehicles
}

q = len(Targets)

# Decision variables
x = LpVariable.dicts("x", ((i, j, m) for i, j in Edges for m in
    Electric_Vehicles), cat="Binary")
z = LpVariable.dicts("z", ((i, j, m) for i, j in Edges for m in
    Electric_Vehicles), lowBound=0)
y = LpVariable.dicts("y", ((d, m) for d in Charging_stations for m in
     Electric_Vehicles), cat="Binary")
w = LpVariable("w", lowBound=0)

# Minimization model
problem = LpProblem("MEVRP", LpMinimize)

# Objective
problem += w

# Constraints
for m in Electric_Vehicles:
    problem += lpSum(costs[i, j] * x[i, j, m] for i, j in Edges) <= w

for d in Charging_stations:
    for m in Electric_Vehicles:
        problem += lpSum(x[d, i, m] for i in Vertices if i != d) ==
    lpSum(x[i, d, m] for i in Vertices if i != d)
        problem += lpSum(x[d, i, m] for i in Vertices if i != d) <= q
     * y[d, m]

for m in Electric_Vehicles:
    problem += lpSum(x[depot, i, m] for i in Vertices if i != depot)
   == 1
    problem += lpSum(x[i, depot, m] for i in Vertices if i != depot)
   == 1

for t in Targets:
    problem += lpSum(x[i, t, m] for i in Vertices if i != t for m in
    Electric_Vehicles) == 1
    problem += lpSum(x[t, j, m] for j in Vertices if j != t for m in
    Electric_Vehicles) == 1

from itertools import combinations
for m in Electric_Vehicles:
    Vertices_wo_depot = [v for v in Vertices if v != depot]
    for r in range(2, len(Vertices_wo_depot)):
        for S in combinations(Vertices_wo_depot, r):
            for d in set(S).intersection(Charging_stations):
                delta_plus_S = [(i, j) for i in S if j not in S for j in Vertices if i != j]
                if delta_plus_S:
                    problem += lpSum(x[i, j, m] for (i, j) in delta_plus_S) >= y[d, m]

for i in Targets:
    for m in Electric_Vehicles:
        problem += (
            lpSum(z[i, j, m] for j in Vertices if i != j)
            - lpSum(z[j, i, m] for j in Vertices if j != i)
            == lpSum(consumption_rate[i, j, m] * x[i, j, m] for j in Vertices if j != i)
        )

for i, j in Edges:
    for m in Electric_Vehicles:
        problem += z[i, j, m] <= Battery_capacity * x[i, j, m]

for d in D:
    for m in Electric_Vehicles:
        for j in Vertices:
            if j != d:
                problem += z[d, j, m] == consumption_rate[d, j, m] * x[d, j, m]

# ---------- Solve + reporting ----------
import time, re, os
from pulp import PULP_CBC_CMD, value, LpStatus

LOG_PATH = "cbc_log.txt"
TIME_LIMIT = 300
solver = PULP_CBC_CMD(timeLimit=TIME_LIMIT, logPath=LOG_PATH, msg=False)

t0 = time.perf_counter()
status = problem.solve(solver)
solve_time = time.perf_counter() - t0

log_text = ""
if os.path.exists(LOG_PATH):
    with open(LOG_PATH, "r", errors="ignore") as f:
        log_text = f.read()

def infer_time_limit_from_log(text):
    return "time limit" in text.lower()

def parse_gap_from_log(text):
    import re
```

```python
    NUM = r'([+-]?\d+(?:\.\d+)?(?:[eE][+-]?\d+)?)'
    m_gap = re.findall(rf"Gap:\s*{NUM}", text)
    if m_gap:
        try:
            g = float(m_gap[-1])
            return g if g > 1 else g * 100
        except Exception:
            return None
    return None

hit_time_limit = infer_time_limit_from_log(log_text)
derived_status = "TimeLimit" if hit_time_limit else LpStatus[problem.status]

obj_val = value(problem.objective)
print(f"Status: {derived_status}")
print(f"Objective value (w): {obj_val:.3f}")
print(f"Runtime: {solve_time:.2f} sec")
```

Appendix B

Instance Generation Code

The following Python function was used to generate the test instances employed in the computational experiments described in Chapter 3 and Chapter 4. The function randomly creates the positions of targets and charging stations within a 100×100 grid, while fixing the depot at the center coordinates $(50, 50)$. Distances are computed as Euclidean values, and both the battery capacity and the energy consumption rate are kept constant.

Listing B.1: Python function for instance generation used in the computational experiments.

```python
import random
from math import sqrt

def generate_instance(n_targets=50, n_cs=5, n_evs=2, grid_size=100):
    depot = 0
    targets = list(range(1, n_targets + 1))
    cs = list(range(n_targets + 1, n_targets + 1 + n_cs))

    # fixed depot in the center of the grid
    coordinates = {depot: (50, 50)}
    used_coords = {(50, 50)}

    # generate unique coordinates
    def generate_unique_coord():
        while True:
            cell = (random.randint(0, grid_size), random.randint(0,
   grid_size))
            if cell not in used_coords:
                used_coords.add(cell)
                return cell

    # random targets
    for t in targets:
        coordinates[t] = generate_unique_coord()

    # random charging stations
    for c in cs:
        coordinates[c] = generate_unique_coord()

    # compute Euclidean distances
    all_nodes = [depot] + targets + cs
    costs = {(i, j): round(sqrt((coordinates[i][0]-coordinates[j][0])**2 +
                                (coordinates[i][1]-coordinates[j][1])**2), 2)
             for i in all_nodes for j in all_nodes if i != j}

    return {
        "Targets": targets,
        "Charging_stations": cs,
        "Depot": depot,
        "Electric_Vehicles": list(range(1, n_evs + 1)),
        "Battery_capacity": 100,
        "Consumption_rate": 0.8,
        "Coordinates": coordinates,
        "Costs": costs
    }
```

Appendix C

Exact model results

The following tables summarize the outcomes of the exact model runs for increasing problem sizes (n_{target} from 5 to 9). Each column reports the optimal objective value (F.O.), runtime in seconds, relative optimality gap (%), and the solver status.

Table C.1: Exact model results grouped by instance size.

$n_{\textbf{target}} = 5,\ n_{\textbf{cs}} = 2,\ n_{\textbf{evs}} = 2,\ F = 100$

F.O.	Runtime (s)	Gap (%)	Status
145.42	3.93	0	Optimal
122.73	1.63	0	Optimal
159.01	3.22	0	Optimal
91.94	1.49	0	Optimal
104.52	0.53	0	Optimal

$n_{\textbf{target}} = 6,\ n_{\textbf{cs}} = 2,\ n_{\textbf{evs}} = 2,\ F = 100$

F.O.	Runtime (s)	Gap (%)	Status
119.65	11.08	0	Optimal
192.51	86.07	0	Optimal
187.58	16.32	0	Optimal
241.06	35.05	0	Optimal
143.56	16.57	0	Optimal

$n_{\textbf{target}} = 7,\ n_{\textbf{cs}} = 2,\ n_{\textbf{evs}} = 2,\ F = 100$

F.O.	Runtime (s)	Gap (%)	Status
139.27	47.10	0	Optimal
85.58	90.72	0	Optimal
201.90	300.00	7.78	TimeLimit
184.62	56.51	0	Optimal
223.58	300.00	27.33	TimeLimit

$n_{\textbf{target}} = 8,\ n_{\textbf{cs}} = 2,\ n_{\textbf{evs}} = 2,\ F = 100$

F.O.	Runtime (s)	Gap (%)	Status
128.92	22.85	0	Optimal
225.27	300.00	22.58	TimeLimit
198.53	300.00	30.92	TimeLimit
124.46	147.21	0	Optimal
164.76	114.29	0	Optimal

$n_{\textbf{target}} = 9,\ n_{\textbf{cs}} = 2,\ n_{\textbf{evs}} = 3,\ F = 100$

F.O.	Runtime (s)	Gap (%)	Status
132.56	300.00	26.75	TimeLimit
159.06	300.00	40.89	TimeLimit
97.04	300.00	31.87	TimeLimit
184.52	300.00	33.73	TimeLimit
162.33	300.00	25.34	TimeLimit

Appendix D

Heuristic Implementation (Python)

The following listing reports the full Python implementation of the heuristic used for the MEVRP. It includes the feasibility enforcement with charging-station insertions, the greedy feasible start, and the local-improvement loop with `swap`/`relocate` moves.

```python
import random
from math import sqrt
from generator import generate_instance

# Euclidean distance
def dist(a, b):
    return sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def route_length(route, coords):
    return sum(dist(coords[route[i]], coords[route[i+1]]) for i in
    range(len(route)-1))

def obj(routes, coords):
    return max(route_length(r, coords) for r in routes.values())

#  Given a base route [depot, t1, t2, ..., depot] WITHOUT CS,
# insert CS (or depot) when the battery is not enough.
# Always keeps all targets: if needed, returns to depot, recharges, and
    continues.
# CS can change at each reconstruction (greedy choice of nearest reachable
    CS).
def enforce_battery_strict(base_route, coords, depot, cs_list, cap, rate,
                           distM=None, energyM=None, minEnergyToCS=None,
                           sorted_cs_by_dist=None):
    if not base_route:
        return [], True

    use_tables = (distM is not None) and (energyM is not None)
    cs_set = set(cs_list)

    def d(a, b):
        return distM[(a, b)] if use_tables else ((coords[a][0]-coords[b][0])**2 + (coords[a][1]-coords[b][1])**2) ** 0.5

    def e(a, b):
        return energyM[(a, b)] if use_tables else rate * d(a, b)

    route = [base_route[0]]
    battery = float(cap)

    # Use index to find out if Depot comes after 'nxt'
    for idx in range(1, len(base_route)):
        nxt = base_route[idx]
        next_is_depot = (idx == len(base_route) - 1)   # the last one on the base_route is the Depot

        safety = 0
        SAFETY_LIMIT = len(cs_list) + 3

        while True:
            safety += 1
            if safety > SAFETY_LIMIT:
                return [], False

            cur = route[-1]
            need = e(cur, nxt)

            # (i) if I don't reach 'nxt' now, I enter a reachable CS
            if need > battery + 1e-9:
                best_cs = None
                if sorted_cs_by_dist is not None:
                    for c in sorted_cs_by_dist[cur]:
                        if c != cur and e(cur, c) <= battery + 1e-9:
                            best_cs = c
                            break
                else:
                    cand = [c for c in cs_list if c != cur and e(cur, c) <= battery + 1e-9]
                    best_cs = min(cand, key=lambda c: d(cur, c)) if cand else None

                if best_cs is None:
                    return [], False

                route.append(best_cs)
                battery -= e(cur, best_cs)
                battery = float(cap)
                continue  # recalculate towards 'nxt'

            # (ii) LOOK-AHEAD
            # - If nxt is a TARGET and not the last before Depot -> must reach a CS after arrival
            # - If nxt is the final TARGET (before Depot) -> must reach Depot after arrival
            if (nxt not in cs_set) and (nxt != depot):
                residual = battery - need
                if next_is_depot:
                    req_after = e(nxt, depot)   # just get to the Depot
                else:
                    if minEnergyToCS is not None and nxt in minEnergyToCS:
                        req_after = minEnergyToCS[nxt]
                    else:
                        req_after = min(e(nxt, c) for c in cs_list)

                if residual + 1e-9 < req_after:
                    # enter CS BEFORE going to 'nxt'
                    best_cs = None
                    if sorted_cs_by_dist is not None:
                        for c in sorted_cs_by_dist[cur]:
                            if c != cur and e(cur, c) <= battery + 1e-9:
                                best_cs = c
                                break
                    else:
                        cand = [c for c in cs_list if c != cur and e(cur, c) <= battery + 1e-9]
                        best_cs = min(cand, key=lambda c: d(cur, c)) if cand else None

                    if best_cs is None:
                        return [], False

                    route.append(best_cs)
                    battery -= e(cur, best_cs)
                    battery = float(cap)
                    continue  # try again towards 'nxt'

            # (iii) ok: go to 'nxt'
            route.append(nxt)
            battery -= need
            if (nxt in cs_set) or (nxt == depot):
                battery = float(cap)
            break

```

```python
    # sanity: Depot only at the head and tail
    if route[0] != depot or route[-1] != depot:
        return route, False
    if any(node == depot for node in route[1:-1]):
        return route, False

    return route, True

def format_route(route, depot, cs_list):
    cs_set = set(cs_list)
    parts = []
    for n in route:
        if n == depot:
            parts.append("Depot")
        elif n in cs_set:
            parts.append(f"CS({n})")
        else:
            parts.append(str(n))
    return " -> ".join(parts)

def print_dist_summaries_HEU(targets, cs_list, depot, coords):
    print("\nPairwise distances between targets:")
    for i in targets:
        for j in targets:
            if i < j:
                print(f"Distance {i}-{j} = {dist(coords[i],
    coords[j]):.2f}")

    print("\nDepot  Target distances:")
    for t in targets:
        print(f"Depot-{t} = {dist(coords[depot], coords[t]):.2f}")

    print("\nTarget  Charging station distances:")
    for t in targets:
        for c in cs_list:
            print(f"Target {t}-CS({c}) = {dist(coords[t], coords[c]):.2f}")

    print("\nDepot  Charging station distances:")
    for c in cs_list:
        print(f"Depot-CS({c}) = {dist(coords[depot], coords[c]):.2f}")

def build_feasible_start(evs, targets, depot, cs_list, coords, cap, rate,
                         distM, energyM, minEnergyToCS, sorted_cs_by_dist):
    """
    Builds a FEASIBLE initial solution without shuffling:
    - Sorts targets by distance from the depot;
    - Attempts to attach each target to the vehicle with the shortest route,
      validating each time with enforce_battery_strict (which inserts CS if
    necessary).
    Returns (assign, routes) or (None, None) if impossible.
    """
    assign = {m: [] for m in evs}
    routes = {m: [depot, depot] for m in evs}

    def route_len(r):
        return sum(distM[(r[i], r[i+1])] for i in range(len(r)-1))

    targets_sorted = sorted(targets, key=lambda t: distM[(depot, t)])

    for t in targets_sorted:
        placed = False
        # test vehicles in order of shortest route
        for m in sorted(evs, key=lambda mm: route_len(routes[mm])):
            tentative = assign[m] + [t]
            base_route = [depot] + tentative + [depot]
            r, ok = enforce_battery_strict(
                base_route, coords, depot, cs_list, cap, rate,
                distM, energyM, minEnergyToCS, sorted_cs_by_dist
            )
            if ok:
                assign[m] = tentative
                routes[m] = r
                placed = True
                break
        if not placed:
            return None, None  # cannot insert t into any vehicle

    return assign, routes

#  heuristic
def heuristic(data, patience=60, p_swap=0.5, show_distances=True):
    # patience: number of consecutive non-improving moves before stopping
    # p_swap:   probability of using SWAP move (otherwise RELOCATE)
    import time
    t_start = time.perf_counter()
    targets = data["Targets"]
    cs_list = data["Charging_stations"]
    depot = data["Depot"]
    evs = data["Electric_Vehicles"]
    coords = data["Coordinates"]
    cap = data["Battery_capacity"]
    rate = data["Consumption_rate"]

    # --- SPEEDUP: precompute distances and energies (include diagonals
    i==j) ---
    all_nodes = [depot] + targets + cs_list

    distM = {(i, j): (0.0 if i == j else dist(coords[i], coords[j]))
```

```python
             for i in all_nodes for j in all_nodes}

    energyM = {(i, j): rate * distM[(i, j)]
               for i in all_nodes for j in all_nodes}

    # minimum energy to reach at least one CS from each TARGET
    minEnergyToCS = {t: min(energyM[(t, c)] for c in cs_list)
                     for t in targets}

    # CS sorted by distance from each node (fast pick of a reachable CS)
    sorted_cs_by_dist = {n: sorted(cs_list, key=lambda c: distM[(n, c)])
                         for n in all_nodes}
    # ------------------------------------------------------------------------

    # === FEASIBLE START (greedy, no shuffle) ===
    assign, routes = build_feasible_start(
        evs, targets, depot, cs_list, coords, cap, rate,
        distM, energyM, minEnergyToCS, sorted_cs_by_dist
    )
    if assign is None:
        print("Infeasible initial assignment (greedy)  no feasible start found.")
        if show_distances:
            print_dist_summaries_HEU(targets, cs_list, depot, coords)
        return {}, float("inf")

    best_obj = obj(routes, coords)

    # Initial solution print
    print("Initial feasible solution")
    for m in evs:
        s = format_route(routes[m], depot, cs_list)
        print(f"EV {m}: {s}")
    print("Initial objective value:", round(best_obj, 2))
    print("\n")

    no_improve = 0
    while no_improve < patience:
        m1, m2 = random.sample(evs, 2)
        do_swap = (random.random() < p_swap)

        if do_swap:
            if not assign[m1] or not assign[m2]:
                no_improve += 1
                continue
            t1 = random.choice(assign[m1])
            t2 = random.choice(assign[m2])

            assign_new = {m: lst[:] for m, lst in assign.items()}
            i1 = assign_new[m1].index(t1)
            i2 = assign_new[m2].index(t2)
            assign_new[m1][i1], assign_new[m2][i2] = assign_new[m2][i2],
    assign_new[m1][i1]

            new_routes = dict(routes)
            r1, ok1 = enforce_battery_strict([depot] + assign_new[m1] +
    [depot],
                                             coords, depot, cs_list, cap,
    rate,
                                             distM, energyM, minEnergyToCS,
    sorted_cs_by_dist)

            r2, ok2 = enforce_battery_strict([depot] + assign_new[m2] +
    [depot],
                                             coords, depot, cs_list, cap,
    rate,
                                             distM, energyM, minEnergyToCS,
    sorted_cs_by_dist)

            if not (ok1 and ok2):
                no_improve += 1
                continue
            new_routes[m1] = r1
            new_routes[m2] = r2

        else:
            if not assign[m1]:
                no_improve += 1
                continue
            t = random.choice(assign[m1])

            assign_new = {m: lst[:] for m, lst in assign.items()}
            assign_new[m1].remove(t)
            assign_new[m2].append(t)

            new_routes = dict(routes)
            r1, ok1 = enforce_battery_strict([depot] + assign_new[m1] +
    [depot],
                                             coords, depot, cs_list, cap,
    rate,
                                             distM, energyM, minEnergyToCS,
    sorted_cs_by_dist)

            r2, ok2 = enforce_battery_strict([depot] + assign_new[m2] +
    [depot],
                                             coords, depot, cs_list, cap,
    rate,
                                                        distM, energyM, minEnergyToCS,
sorted_cs_by_dist)

            if not (ok1 and ok2):
                no_improve += 1
                continue
            new_routes[m1] = r1
            new_routes[m2] = r2

        new_obj = obj(new_routes, coords)
        if new_obj < best_obj:
            assign = assign_new
            routes = new_routes
            best_obj = new_obj
            no_improve = 0
        else:
            no_improve += 1

    # ------- Final diagnostics & reporting for the heuristic -------
    cs_set = set(cs_list)

    def fmt_node(n):
        if n == depot:
            return "Depot"
        if n in cs_set:
            idx = cs_list.index(n) + 1
            return f"CS({idx})"
        return str(n)

    def route_length_from_coords(route):
        return sum(dist(coords[route[i]], coords[route[i+1]]) for i in
range(len(route)-1))

    if not routes or any(not r or len(r) < 2 for r in routes.values()):
        print("\nSolution infeasible for the given instance/parameters.")
        if show_distances:
            print_dist_summaries_HEU(targets, cs_list, depot, coords)
        return routes, float('inf')

    print("\nFinal solution (heuristic)")
    any_capacity_violation = False
    per_vehicle_lengths = []

    for m in sorted(routes.keys()):
        r = routes[m]
        if not r or len(r) < 2:
            continue

        print(f"EV {m}: " + " - ".join(fmt_node(n) for n in r))

        print("Leg  From        To                Dist     Energy
    Batt_before  Batt_after  Recharge_to  Batt_after_rech")
    print("------------------------------------------------------------------------------------

        F = float(cap)
        battery = F
        cum_viol = False

        for k in range(len(r) - 1):
            i, j = r[k], r[k+1]
            dij = dist(coords[i], coords[j])
            eij = rate * dij

            batt_before = battery
            batt_after = batt_before - eij
            if batt_after < -1e-9:
                cum_viol = True

            recharge_to = ""
            batt_after_rech = ""
            if (j in cs_set) or (j == depot):
                recharge_to = fmt_node(j)
                battery = F
                batt_after_rech = f"{battery:10.2f}"
            else:
                battery = batt_after

            print(f"{k:3d}  {fmt_node(i):<10} -> {fmt_node(j):<10}
    {dij:7.2f}  {eij:8.2f}  "
                  f"{batt_before:12.2f}  {batt_after:11.2f}
    {recharge_to:>11}  {batt_after_rech:>16}")

        rl = route_length_from_coords(r)
        per_vehicle_lengths.append(rl)
        print(f"Route length (EV {m}): {rl:.2f}\n")
        any_capacity_violation = any_capacity_violation or cum_viol

    max_len = max(per_vehicle_lengths) if per_vehicle_lengths else 0.0
    print(f"Vehicles: {len(evs)}")
    print(f"Targets:  {len(targets)}")
    print(f"Heuristic objective (max route length): {max_len:.2f}")
    print(f"best_obj tracked:                       {best_obj:.2f}")
    if abs(max_len - best_obj) > 1e-6:
        print("Note: obj mismatch between print and tracker; double-check
    route lengths.")

    if any_capacity_violation:
        print("WARNING: infeasible under energy model (cumulative battery violated).")
    else:
        print("Energy feasibility check passed (cumulative).")

    if show_distances:
        print("\nPairwise distances between targets:")
        for i in targets:
            for j in targets:
                if i < j:
                    print(f"Distance {i}-{j} = {dist(coords[i], coords[j]):.2f}")

        print("\nDepot  Target distances:")
        for t in targets:
            print(f"Depot-{t} = {dist(coords[depot], coords[t]):.2f}")

        print("\nTarget  Charging station distances:")
        for t in targets:
            for c in cs_list:
                print(f"Target {t}-CS({c}) = {dist(coords[t], coords[c]):.2f}")

        print("\nDepot  Charging station distances:")
        for c in cs_list:
            print(f"Depot-CS({c}) = {dist(coords[depot], coords[c]):.2f}")

    runtime = time.perf_counter() - t_start
    print(f"\nHeuristic runtime (wallclock): {runtime:.2f} seconds")
    return routes, max_len

#  run example
data = generate_instance(n_targets=240, n_cs=40, n_evs=45, grid_size=100)
routes, best_obj = heuristic(data, patience=100000, p_swap=0.5, show_distances=False)
```

Listing D.1: Python heuristic for the MEVRP with battery-feasibility enforcement

Appendix E

Detailed Heuristic Test Data

This appendix reports the detailed numerical results for each test instance used to compute the average values presented in Chapter 4 (Table 4.1). Each table corresponds to a different instance configuration, showing the initial and final objective values and the runtime for each of the five runs. All experiments were performed with fixed parameters $F = 100$ and an energy consumption rate of 0.8 per unit of distance.

Table E.1: Detailed results for instances with 8 and 15 targets.

8 Targets, 3 CS, 2 EVs			**15 Targets, 4 CS, 3 EVs**		
Init. F.O.	**Final F.O.**	**Time (s)**	**Init. F.O.**	**Final F.O.**	**Time (s)**
321.50	258.72	5.84	432.95	275.99	7.97
383.82	287.05	6.63	310.48	193.13	6.62
389.72	294.25	6.03	331.50	274.60	7.59
341.66	267.04	8.64	332.04	302.21	7.69
242.24	210.01	5.54	354.73	225.87	7.24

Table E.2: Detailed results for instances with 30 and 50 targets.

30 Targets, 6 CS, 6 EVs			**50 Targets, 8 CS, 10 EVs**		
Init. F.O.	**Final F.O.**	**Time (s)**	**Init. F.O.**	**Final F.O.**	**Time (s)**
382.67	186.45	8.99	374.54	183.36	15.70
438.76	208.04	8.63	427.94	179.04	13.57
420.75	184.75	10.09	414.91	194.45	14.06
409.06	157.97	9.62	315.35	164.43	14.17
397.98	202.39	8.92	340.23	190.87	16.31

Table E.3: Detailed results for instances with 75 and 100 targets.

75 Targets, 13 CS, 15 EVs			**100 Targets, 20 CS, 20 EVs**		
Init. F.O.	**Final F.O.**	**Time (s)**	**Init. F.O.**	**Final F.O.**	**Time (s)**
350.95	151.76	25.67	361.89	156.18	34.37
383.39	190.01	17.87	401.95	182.89	27.47
342.78	157.81	27.33	432.71	143.20	41.20
362.07	157.48	23.52	361.44	159.97	28.09
351.45	158.76	21.57	371.19	172.07	33.28

Table E.4: Detailed results for instances with 200 and 240 targets.

200 Targets, 40 CS, 40 EVs			**240 Targets, 40 CS, 45 EVs**		
Init. F.O.	**Final F.O.**	**Time (s)**	**Init. F.O.**	**Final F.O.**	**Time (s)**
384.37	141.11	219.36	365.78	137.89	392.84
354.11	142.16	145.36	379.02	148.49	294.72
366.95	151.30	135.30	393.50	142.70	308.70
394.07	146.79	157.34	384.89	152.48	235.43
339.68	146.40	184.10	396.86	140.39	392.67

Bibliography

[1] *Fit for 55: delivering the EU's 2030 Climate Target on the way to climate neutrality.* URL: https://ec.europa.eu/commission/presscorner/detail/en/IP_21_3541

[2] *Regulation (EU) 2023/851 of the European Parliament and of the Council of 19 April 2023.* URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32023R0851

[3] Kai Huang, Fei Xie, Shi-An Shen, Jun Li, and Zhiwei Wang. «Electric Vehicle Routing Problem Considering Charging Time and Battery Degradation». In: *Transportation Research Record* 2677.12 (2023), pp. 1087–1099. URL: https://journals.sagepub.com/doi/abs/10.1177/03611981231207096.

[4] *Introduzione alla Programmazione Lineare.* URL: https://www.di.unito.it/~locatell/didattica/ro1/intropl-sl-bf.pdf

[5] *Vehicle routing problem — Wikipedia, The Free Encyclopedia.* URL: https://it.wikipedia.org/wiki/Vehicle_routing_problem

[6] S. F. Ghannadpour, A. Noori-Daryan, H. Yousefi, and A. Sadeghi. «A multi-depot green vehicle routing problem with time windows, heterogeneous fleet, and split delivery: Mathematical modeling and solution approach». In: *International Journal of Production Economics* 227 (2020), p. 107618. URL: https://www.sciencedirect.com/science/article/abs/pii/S0925527320302607.

[7] S. Hassan, A. F. Alqaser, M. Alhaj, and M. Alhazmi. «Efficient algorithms for electric vehicles' min–max routing problem». In: *Operations Research Perspectives* 10 (2023), p. 100231. URL: https://www.sciencedirect.com/science/article/pii/S2666412723000107.

[8] Stuart A Mitchell, Michael Mason, et al. «PuLP: A linear programming toolkit for Python». In: *The University of Auckland, Auckland, New Zealand* (2011). Available at https://coin-or.github.io/pulp/.

[9] Zhongzheng Liu, Zhiguang Cao, Jie Zhang, and Andrew Lim. «Learning to Solve Vehicle Routing Problems: A Comprehensive Review». In: *arXiv preprint arXiv:2303.04147* (2023). Accessed: October 2025. URL: `https://arxiv.org/pdf/2303.04147`.