Lancaster University

The Electrical Vehicles Routing Problem

Blaise Petit

35913359

Supervisor

Ahmed Kheiri

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science Business Analytics

in the

Lancaster University Management School
Department of Management Science

September 2022

Declaration of Authorship

I, **Blaise Petit**, hereby declare that this thesis entitled, **"The Electrical Vehicles Routing Problem"**, is all my own work, except as indicated in the text.

The report has been not accepted for any degree and it is not being submitted currently in candidature for any degree or other reward.

Signed: ***Blaise Petit***

Date: **September 19, 2022**

Abstract

The VRP is a classical variation of the TSP and are both two *NP-hard* problems of optimisation, which means that there are no know algorithms to solve them with a polynomial complexity. It is a simplistic base problem to describe delivering challenges of delivering companies. In a worldwide context of measures against climate change, private companies of delivering are changing their fleet of thermal vehicles for EVs to stop the emission of CO2 for last mile deliveries.

Consequently, a new evolved formulation of the VRP is now becoming critical: the EVRP. The idea of minimising total travel distance remains the same, but in that case the battery capacity of the vehicles is limited and they need to charge their battery to charging stations, making the problem much harder to solve. The goal of that dissertation is to provide the best solving method to solve the EVRP instances given in the CEC-12 Competition 2020 on Electric Vehicle Routing Problem. The proposed algorithm is a heuristic algorithm based on local search with an evolving way of picking new operators and additional parameters to escape local optima. The heuristic algorithm comes along with two constructive algorithms that would build starting solutions.

The performance of the local search algorithm will then be compared to the optimal value found using MIP. To finish the dissertation, comments to make the model more accurate, ideas on how to improve the efficiency of the algorithm and ways of speeding up the Python implementation are discussed in the last chapter.

Acknowledgements

I am really thankful toward my supervisor, Dr Ahmed Kheiri, for introducing me into that fascinating topic and helping me with heuristic details.
I am also having a thought for everybody that supports me during the period of writing this dissertation, family and classmates.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

EV	Electrical Vehicles
CO2	Carbon Dioxide
GGE	Greenhouse Gas Emissions
VRP	Vehicles Routing Problem
TSP	Traveling Salesman Problem
CVRP	Capacited Vehicles Routing Problem
ECVRP	Electrical Capacited Vehicles Routing Problem
EVRP	Electrical Vehicles Routing Problem
MIP	Mixed Integer Programming
RAM	Random Access Memory
JIT	Just In-time Compiler

Chapter 1

Introduction

In a recent survey conveyed by Statista, it has been found out that 44% of Western customers are expecting their parcel to be delivered in 2 days or less. The amount of online shopping and deliveries is expected to skyrocket by 30% in the incoming years. Consequently, the market of last miles delivery is growing and delivering company are facing a logistic issue to deliver everybody without making the cost too expensive. In the past the only concern was the ability to deliver parcels on time to everybody with the lowest total distance, delivering companies were formulating that problem as CVRP at that time.

In the current context of fight against global warming, delivering companies are trying to find a compromise between the growing demand and the need to reduce carbon dioxide emission. The reason to choose to reduce greenhouses gas emission are many: to avoid huge tax on emission, ethic line of the brand, etc. In every case, it is becoming one of the key issues they decide to focus on while planning their strategy for the near future. In this dissertation, we will develop the impact on logistic method one of the chosen solutions to tackle the problem of carbon dioxide emission: changing the fleet for electrical vehicles. Most of the biggest logistic companies such as FedEx, UPS, DHL and others are already making the shift to EVs, especially for last mile delivery. Electrical vehicles are not emitting any gas or particles while running, but they are having three particularly annoying drawbacks: their battery autonomy is limited, it is very hard to find an electrical charging station and the charging time is long. The problem of charging battery is really getting improved from year to year: for example Tesla is currently running over 35k+ supercharger worldwide, supporting up to 250kW charge, which means that electrical vehicles can be half charged in approximately 15 minutes.

However, the autonomy of the battery is struggling to get improves, new technology of battery are currently being developed, but it is taking time and the percentage of progress expected are never any close to be 100%. Taking all these new issues into account, it was necessary to upgrade the CVRP to the Electrical Capacited Vehicles Routing Problem (ECVRP) and to find methods to solve it: the previous methods are no longer guarantee to find good solution, but even worse solutions are very unlikely to be feasible due to the battery problem. The next part of literature review will show the evolution of VRP formulation to reach the new project this dissertation will be dealing with: EVRP.

Chapter 2

Literature review

2.1 The evolution of TSP to CVRP

The first ever optimisation problem about routing was the TSP, defined for the first time somewhere in the middle of the 19^{th} century was already a *NP-Hard* optimisation problem. It is in the middle of the 20^{th} century that the TSP evolved to a logistic problem of delivery as Dantzig and Ramser [1].
At the beginning there was no formulation and no way of finding exact solutions for this problem, the first heuristic algorithm to manage to produce good solution was the constructive algorithm proposed by Clarks and Wrights in 1964 [2]. It was based on the concept of starting from tour with unique visit, and computing the overall saving of merging two routes to always choose the move maximising the savings: it is one of the first "greedy" algorithm used on CVRP and it will be a base for a greedy algorithm for EVRP later in the dissertation.
The first exact method that was able to solve CVRP came almost 20 years later and was published by Laporte et al. [3]. It was based on a cutting plane approach of the relaxation of the linear solution of a fully integer model.
Then from the early 2000's both the exact solving and the heuristic evolved in parallel: the exact solving was solving bigger and bigger instances in less time, but the speed and the size of the maximum instance was remaining in favour of the heuristic algorithm. Exact method were adding inequalities to try to make always better cuts to solve the problem faster, it is the case in the work of Fukasawa et al. (2006) [4] and Baldacci et al. (2008) [5]. On the other hand, it is mostly meta heuristics method that were developed with Tabu search (Brandao [6]), Simulated annealing (Banos et al. [7]), genetic algorithm

(Vidal et al. [8]), etc...
In parallel a lot of variations of the CVRP have been studied like CVRP with time window, CVRP with stochastic demand, etc.. but the only variation that we will focus on today is the ECVRP (also known as EVRP).

2.2 Global warming becomes a concern and EVRP emerges

The EVRP is a relatively new formulation that aims at providing good solutions of CVRP while taking into account the limitation of the battery of EVs in order to tackle climate change. The interest of that subject is really skyrocketing since 2013 according to the graph from [9] Fig.2.1:

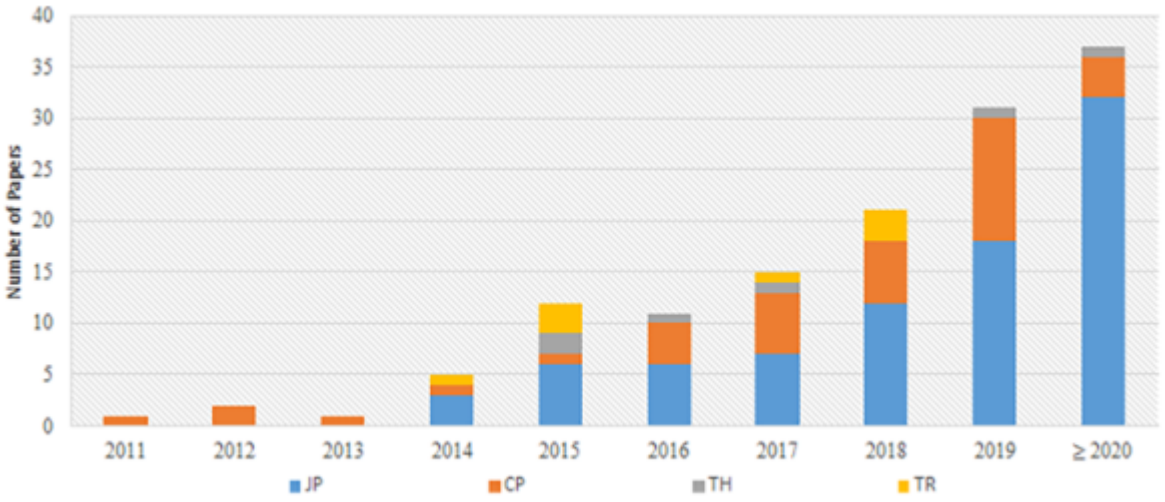

FIGURE 2.1: Number of papers on EVRP by year

However, most of the paper focus only one specific side of the topic: either its the model of battery consumption (Goeke Schneider [10]), or it is to add duration time limit (Montoya et al. [11]) or simply to focus on the meta heuristic quality of solution. In this dissertation the focus will be put not only on formulation and on model, but also link to real life situation and on the quality of the meta heuristic solving algorithm.

Chapter 3

Problem description

In this dissertation, I decided to follow the rules set for the CEC-12 Competition on Electric Vehicle Routing Problem [12], so every of the following elements are influenced by these rules. This chapter will focus on explaining the objective function and the constraints of the EVRP, as well as presenting the format of the 17 available instances. It will be concluded by the exact formulation which allows solver like *Gurobi* to give optimal solutions and value.

3.1 The Problem

As intended according to the introduction, the scope of the dissertation will be focused on the EVRP. The EVRP is a complexified version of the CVRP to take into account battery consumption constraints. The CVRP is well famous to be *NP-Hard*, consequently all its variations including EVRP are also NP-hard according to [13]. As consequence, we cannot except any solving algorithm to have a lower complexity than polynomial time, but the results will be shown to be actually worse later in the dissertation.

The EVRP, like the CVRP, is described using 2D graphs with the following items:

- ***A depot*** and its coordinates
- ***n customers*** to serve and the set of n associated coordinates
- ***m charging stations*** that charges the battery of every EVs visiting them and the set of m associated coordinates

- $\forall i \in \{1, 2, .., n\}$, $d_i \geqslant 0$, d_i is the **associated demand with the i^{th} customers**
- A fleet of ***minimum k EVs***, with a capacity to deliver of C and a battery capacity of Q

There are a lot of possible small variations on the constraints (fixing the number of vehicles or not, allowing vehicules to serve multiple route or not, etc..), but in this dissertation I chose to stick to the formulation given in the CEC-12 Competition on Electric Vehicle Routing Problem [12]. EVRP is an evolution of CVRP that aims to reduce the GGE using EV without reducing the delivering quality of delivering using thermal vehicles. The only emission of CO2 associated with the use of EVs is the cost of the production of the electrical energy used by the vehicles [14]. Consequently, both the cost of delivery and the quantity of GGE are highly correlated with the total traveled distances, thus the objective function that has been chosen to be **minimise** is the total distances travelled by the fleet of EVs.

According to [12], a solution is a set of routes (one route for one EV of the fleet) that is respecting the following rules:

- Every route **must start and end** with **the depot**;
- All clients **must be served** but **only once**;
- During its tour in its planned route, the EV **must never fall out of battery or fall short of capacity to deliver customers**;
- The depot is considered as a charging station too, and there is **no limit of use of each charging station**.

The last assumption made is to assume that the consumption of electrical energy from the battery is linear with the distance travelled. So I will use h the given coefficient of linear consumption: it means that if d_{ij} is the distance between the node i and the node j, the energy consumed by this travel would be: $Energy_{consumed} = h \times d_{ij}$

3.2 Instances used as benchmark

The instances used in this dissertation are the 17 ECVRP instances given in for the CEC-12 contest, which have been build from the famous existing CVRP instances of

E.UCHOA in [15]. M.MAVROVOUNIOTIS is proposing those 17 augmented instances in [16]. All the file used as instances are following the format convention of VRPlib with *.evrp* extension. They are named following the format E/X - n - k, with E instances with known optimal value, X instances with unknown optimal value, n the number of customers and k the minimum number of EVs used in the solution (as indication).

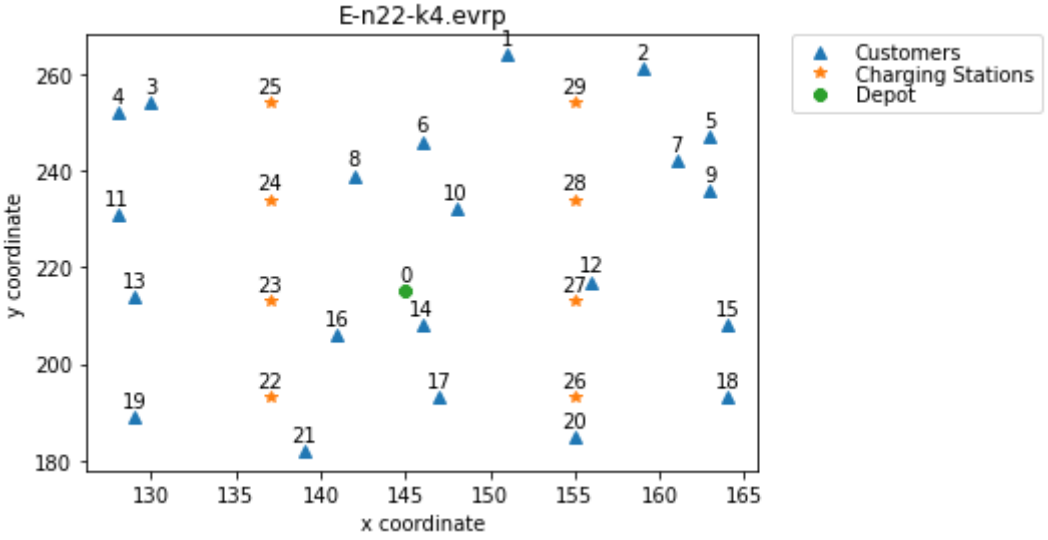

FIGURE 3.1: Example of the smallest instance

In Fig.3.1 above, we can see the 22 customers as blue triangles, the 8 charging stations as orange stars and the depot as green circles inside the 2D coordinates axis.

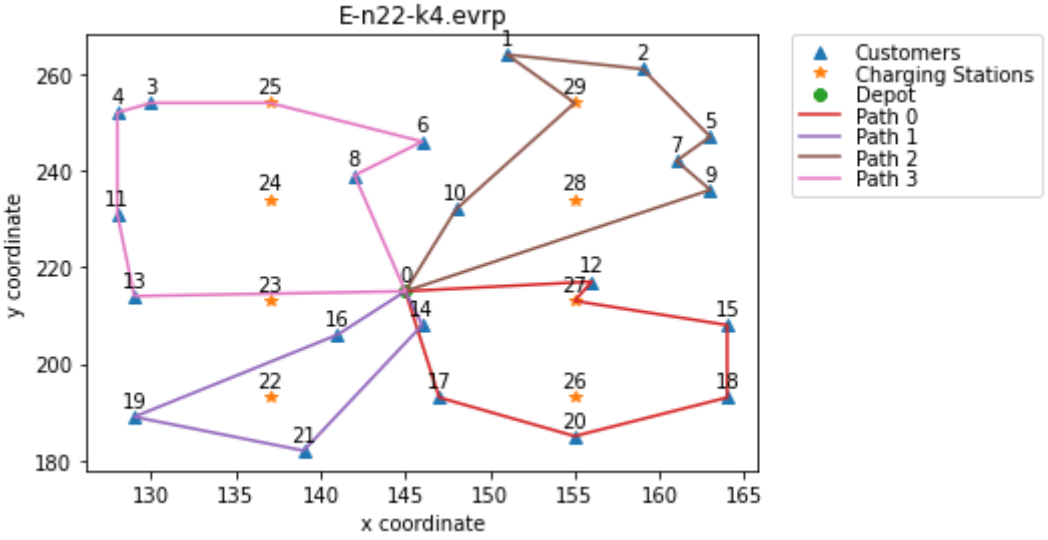

FIGURE 3.2: Example of a feasible solution (here it is optimal)

In Fig.3.2 we have a nice overview of what could be a feasible solution. Here, the displayed solution is actually the optimal one, we can see it has 4 routes as indicated in the file name ($k = 4$). We can also notice that charging stations 25, 27 and 29 are visited to guarantee battery feasibility of tour.

It is also interesting to talk about the algorithm used to convert CVRP instances into EVRP instances described in [16]. The first step is to define the maximum battery capacity of each EV, named Q. It is defined to be proportional to $\bar{d}$, the maximum distance between the depot and a customer: that way the battery is designed to allow to travel from the depot to any point, which makes it easy to end tour since the battery is large enough to come back from almost every possible customers. Consequently we have the following first equation:

$$Q = \gamma \cdot \bar{d} \tag{3.1}$$

The γ sign used in 3.1 represents a parameter set by the user: too little, the battery will be too small to allow to reach every customers (typically when $\gamma \leqslant 1$), but too big, the battery will be enough to make tours without visiting charging stations. It has to be set carefully to create the most realistic instances possible.

The second step to convert CVRP into EVRP is to assume that every charging station covers an area of radius R proportional to the battery capacity Q we just define. The ultimate goal is to put the minimum number of charging stations to guarantee feasibility, so the area of action of charging stations must overlap the least possible. We then define the following equation:

$$R = \phi \cdot Q \tag{3.2}$$

The ϕ sign used in 3.2 is another parameters that should be set by the user. The lower it is, the more charging stations would be necessary to cover the whole map, but if it is too high charging stations radius of action would overlap and it would be a waste.

After this initial set up, it is possible to use the properties of convexity and MIP to implement charging stations at the right positions to guarantee that the new generated EVRP instance will be feasible. The chosen parameters have been $\gamma = 2$ and $\phi = 0.125$, a chose of consequent battery capacity but fewer amount of charging stations: this is

the best balance found to approach the reality and also guarantee that battery will be a problem in the routing problem (otherwise it would virtually remain a CVRP).

3.3 The mathematical formulation

Even if in this dissertation I chose to solve the problem using various heuristic algorithms, it is still interesting to formulate the problem mathematically in a way that it can be solved using MIP, even if the computing time will rapidly skyrocket with the number of customers n. According to the description of the problem drawn so far and the paper of M.MAVROVOUNIOTIS [12] describing the rule of the contest, a working mathematical formulation for MIP would be based on the following notation: a fully connected weighted graph $G = (V, A)$ where $V = \{0 \cup I \cup F'\}$ is a set of nodes. It is important to note that:

- I denotes the set of customers (the set is size n)
- Let F be the set that denotes the charging stations in the instances. F' is the set of β_i copies of F (it is a mathematical trick to allow multiple visits of each charging stations). So we have $|F'| = \sum_{i \in F} \beta_i$
- $\{0\}$ denotes the depot (the coordinates are not necessarily $(0, 0)$)
- h denotes the coefficient of linear electrical consumption
- d_{ij} represents the distance between node i and node j, we also have by definition $d_{ij} \geq 0$
- Consequently the electrical energy consumed to travel from node i to node j is equal to hd_{ij}

In the chosen formulation there are three different parameters, even though we are only choosing the tours ($x_{ij} = 0$ if we do not want to travel from node i to node j in that tour, or $x_{ij} = 1$ if we want the travel from node i to j to be part of the current tour). We also have to add two parameters: u_i and y_i, respectively representing the remaining capacity of good to deliver when arriving to node i and the remaining battery capacity when arriving to node j. We assume that EVs are leaving the depot fully loaded ($u_0 = C$) and fully charged ($y_0 = Q$). These values have to be set as parameters because their values depends on the solution and cannot be known before hand during the formulation

phase. This makes the formulation a bit trickier but it is the only way of taking those very important constraints into account in the MIP formulation. Finally, the chosen MIP is the following:

$$\min \sum_{i \in V, j \in V, i \neq j} d_{ij} x_{ij} \tag{3.3}$$

$$\sum_{j \in V, i \neq j} x_{ij} = 1, \forall i \in I \tag{3.4}$$

$$\sum_{j \in V, i \neq j} x_{ij} \leq 1, \forall i \in F' \tag{3.5}$$

$$\sum_{j \in V, i \neq j} x_{ij} - \sum_{j \in V, i \neq j} x_{ji} = 0, \forall i \in V \tag{3.6}$$

$$u_j \leq u_i - b_i x_{ij} + C(1 - x_{ij}), \forall i \in V, \forall j \in V, i \neq j \tag{3.7}$$

$$0 \leq u_i \leq C, \forall i \in V \tag{3.8}$$

$$y_j \leq y_i - h d_{ij} x_{ij} + Q(1 - x_{ij}), \forall i \in I, \forall j \in V, i \neq j \tag{3.9}$$

$$y_j \leq Q - h d_{ij} x_{ij}, \forall i \in F' \cup \{0\}, \forall j \in V, i \neq j \tag{3.10}$$

$$0 \leq y_i \leq Q, \forall i \in V \tag{3.11}$$

$$x_{ij} \in \{0, 1\}, \forall i \in V, \forall j \in V, i \neq j \tag{3.12}$$

Let's have a quick overview of the usefulness of each of the previous equations:

- Eq.3.3 is the objective function, minimising the total traveled distance

- Eq.3.4 is assuring that every clients are visited exactly once
- Eq.3.5 allows every charging stations to be visited at least once, which allows the algorithm to fix battery problem to reach a feasible solution
- Since we are working with a connected graph, it is important to verify the flow equality at each node with Eq.3.6
- Eq.3.7 and Eq.3.8 works together to guarantee that the demand of client (b_i) is fulfilled (Eq.3.7) and that the value of the remaining capacity u_i remains within a feasible range (Eq.3.8)
- The logic remains the same for battery with Eq.3.9, Eq.3.10 and Eq.3.11, however Eq.3.9 and Eq.3.10 have been separated in two to take into account the case of reaching a charging station or the depot and charging the battery
- The last equation (Eq.3.12) defines the binary variables x_{ij} representing doing the travel from i to j if $x_{ij} = 1$ or not doing it if $x_{ij} = 0$

This chapter was mostly focusing on describing the grounding rules of the EVRP, as long as introducing the format of the instances (following VRPlib standards). The exact mathematical formulation has been presented at the end as it will be useful to solve the instances exactly, the optimal value will then be use as lower bound to benchmark the heuristic methods.

Chapter 4

Solving the instances: a Heuristic method

This chapter will extensively discuss the details of all the algorithm used to find first solution, but also to make local search. The algorithm will then be benchmark in the next Chapter (chapter 5).

4.1 Short introduction about exact solving

When possible, it is always considered first to solve an optimisation problem using an exact method. There is one major benefit in using exact method solving: there is no need to try to assess the quality of the found solution since it is optimal. It can also be useful to evaluate quality of solution of heuristic method since it allows to know the optimal solution. Using Gurobi solver on the previous MIP we finally got the following results which would be useful for later benchmarks:

Instance name	**Optimal value**
E-n22-k4	384.68
E-n23-k3	573.13
E-n30-k3	511.25
E-n33-k4	869.89
E-n51-k5	570.17
E-n76-k7	723.37
E-n101-k8	899.89

Table 4.1: Table of optimal value by instance

Unfortunately, using a solver like *Gurobi* for MIP is only possible for the smallest instances, the complexity of the algorithm that solves is higher than polynomial since the EVRP is $NP - hard$, which means than it is in practice almost impossible to solve instances bigger than a hundred of clients in a reasonable time using a personal computer (it is possible to solve them using servers specialised in computing power for 20 hours or so but it is totally out of the scope of this dissertation).

4.2 Handling the instance and useful framework

I made the choice to work with **Python 3.9** with **Spyder 5.2** as IDE. Python is a high level language that allows a very fast implementation of the basic framework we would need to work on solving instances with heuristic methods:

- Everything that would be necessary to read the VRPlib format used for instances
- A *feasibility* function that evaluate the feasibility of the chosen solutions: are they any violations in battery capacity? in carrying capacity? Are all clients visited only once?
- A *cost* function that would evaluate the total traveled distance of solutions to evaluate how great and close to optimally they are
- One or multiple *generate first solution* functions to have a starting point for the local search algorithm to find good solutions
- A *heuristic solve* function that starts from the given solution and try to find an optimal one using a custom local search that will be described later in the dissertation

4.2.1 Handling VRPlib format using *open*

The format used in the *.evrp* file is the consensus reached in the VRPlib. A example is given in the following figure:

```
Name: Checking instance
COMMENT: Smallest instance for checking
TYPE: EVRP
OPTIMAL_VALUE: 384.678035
VEHICLES: 4
DIMENSION: 7
STATIONS: 1
CAPACITY: 6000
ENERGY_CAPACITY: 94
ENERGY_CONSUMPTION: 1.20
EDGE_WEIGHT_FORMAT: EUC_2D
NODE_COORD_SECTION
1 145 215
2 130 254
3 128 252
4 146 246
5 142 239
6 128 231
7 129 214
8 137 254
DEMAND_SECTION
1 0
2 800
3 1400
4 400
5 100
6 1200
7 1300
STATIONS_COORD_SECTION
8
DEPOT_SECTION
1
-1
EOF
```

FIGURE 4.1: Example of the format of VRPlib in *.evrp* file for instances

In Fig.4.1, we can notice that the only thing that will vary is the number of line of clients and charging stations, however they are given by DIMENSION and STATIONS so I just implemented a parser that is going through all lines of the instances, storing all parameters in a dictionary of parameters, and creating list of clients, a list of charging station as well as a dictionary having id of nodes as *key* and coordinates as *value*.

Another thing that will be later needed is the distance between nodes to compute battery consumption and to evaluate the cost, so we compute them one time in a matrix size $n \times n$ called *dist array* and it looks like following:

	0	1	2	3	4	5
0	0	49.366	48.0833	41.7852	40.7185	36.7151
1	49.366	0	8.544	23.2594	25.9422	20.8087
2	48.0833	8.544	0	29.8329	32.28	14.5602
3	41.7852	23.2594	29.8329	0	2.82843	33.7343
4	40.7185	25.9422	32.28	2.82843	0	35.3553
5	36.7151	20.8087	14.5602	33.7343	35.3553	0

FIGURE 4.2: Example of the format of the matrix dist array

We can see in Fig.4.2 that the diagonal is equal to 0 since the distance from one node to the same node is null, and we also see that the matrix is symmetrical since by definition $d_{ij} = d_{ji}$.

4.2.2 The *cost* function

The requirements to compute the *cost* function is to have the matrix of distance between each node (defined as *dist array* previously) and to have a solution (a set of at least one non empty tour). I chose to represent solution by a dictionary using ID of the tour as *key* and a list of visited node as *value* (it represents a tour). A solution with two tours would have the following form:

Key	Type	Size	Value
0	list	4	[0, 1, 2, 0]
1	list	4	[0, 3, 4, 0]

FIGURE 4.3: Example of a solution with two tours

In Fig.4.3 we can see a solution made by two tours. The first tour is going from depot (0), to client 1, then client 2 and going back to depot (0). The second tour is going from depot (0), to client 3, then client 4 and finally going back to depot (0). The *cost* here must be computed using the following formula:

$$cost = d_{01} + d_{12} + d_{20} + d_{03} + d_{34} + d_{40} \tag{4.1}$$

4.2.3 The *feasibility* function

The *cost* function is mandatory to evaluate solution, however if we do not check the feasibility it is pointless: if we are visiting half the clients we will of course have a lower cost but it is not an interesting solution. The *feasibility* solution will have to check multiple things to assure that the solution is feasible:

- There must be no **violation of battery capacity**, which means that the sum of hd_{ij} between charging stations in every tour of the solution must never exceed Q (the maximum battery capacity when fully charged)
- There must be no **violation of capacity**, which means that the sum of b_i in every tour of the solution must not exceed C (the maximum capacity when fully loaded)
- All clients must be visited **once**

We are using the following notation: $v_{battery}$ is equal to the total amount of battery energy excess in the solution, $v_{capacity}$ is equal to the total of capacity excess in the solution, and $v_{clients}$ is the total of clients that are visited more than once. I then decided to use the following expression as feasibility equation:

$$feasibility = (v_{battery} + v_{capacity} + 10v_{clients}) \tag{4.2}$$

In Eq.4.2, we are establishing the feasibility to be the balance between the three type of violations: battery violation, capacity violation, and multiple visit violation (it has been multiplied by 10 for scaling purposes). Note that the problem of not visiting clients have been omitted since the starting point we will define later always visit every clients, and the algorithm we will use are always keeping the same number of customers.

With all these functions it is now possible to evaluate a solution, possible at every iterations of a local search heuristic algorithm. However, we need to find a way of having a starting consistent solution to start the local search and try to reach optimally.

4.3 Constructive algorithm to find an initial solution

We know that the the quality of the first solution in a local search algorithm is important: it is always better if the solution is anywhere close to being feasible, and having the

smallest cost possible. It is also very an important criteria that this solution could be consistently found to guarantee a consistent starting point. This is why the first starting solution I will use is based on a constructive algorithm based on nested conditions to guarantee as much as possible feasibility: we will see that it does not guarantee to visit all clients, so the remaining clients would be added to the last tour because it will be very important to have all clients in our solution for the local search algorithm. The proposed constructive algorithm is the following, it will be explained in further details:

Algorithm 1 Pseudo code of the constructive algorithm

Require: *cust_list* the list of customers
charging_station_list the list of charging stations
demand_dict the dictionary of demand for every clients
dist_array the matrix of distance between nodes
Ensure: Finding a **most likely to be feasible** solution. In any case every clients will be visited and the solution will be close to feasible.
available_move $\leftarrow True$ {There are remaining available moves}
non_visited_clients $\leftarrow$ *cust_list* {The list of non visited clients is all clients}
visited_clients $\leftarrow$ [] {The list of visited clients is empty}
dict_sol $\leftarrow$ {The dictionary of tours representing the solution is empty}
tour $\leftarrow$ [0] {The current tour is empty and will get updated}
curr_tour_id $\leftarrow$ 0 {The first tour index is 0}
while $len($*visited_clients*$) <$ *nbr_cust* **and** *available_move* **do**
  ordered_list_of_candidates $\leftarrow$ List of all possible tour candidates ordered by cost {More details will be provided in description}
  if *ordered_list_of_candidates* has at least one value **then**
    tour $\leftarrow$ *best_candidates* {Tour is updated}
    dict_sol[*curr_tour_id*] $\leftarrow$ *tour*
    visited_clients and *non_visited_clients* are updated following the previous move
  else
    if *tour* $=$ [0] **then**
      available_move $\leftarrow False$ {No moves are available}
    else
      tour $\leftarrow$ tour finished with a special algorithm to make it feasible
      dict_sol[*curr_tour_id*] $\leftarrow$ *tour*
      curr_tour_ind $\leftarrow$ *curr_tour_ind* $+ 1$ {We start the next tour}
      tour $\leftarrow$ [0] {New tour is reseted}
    end if
  end if
end while
if not available_move **then**
  dict_sol[*curr_tour_ind* $- 1$] $\leftarrow$ all remaining non served clients
else
  tour $\leftarrow$ tour finished with a special algorithm to make it feasible
  dict_sol[*curr_tour_id*] $\leftarrow$ *tour*
end if

The overall idea of the algorithm proposed above (1) is to use property of the problem to construct a solution starting from empty tour and always doing the best move, following a greedy logic. However, the constraints is that, we cannot guarantee the feasibility of the solution by doing that method, it is possible to be stuck with a client that is too far to be visited according to battery constraints in the hardest instances: in that case the clients are artificially added to the last tour, the solution is not feasible but would still be usable for local search. We start from empty solution, and we construct tours one by one, always adding all the possible non visited clients to the tail of the route. Then we check 2 things: if the capacity is violated the tour is instantly deleted, but if the battery is violated, the algorithm try to insert a charging stations in the best place of the current tour to make it feasible and to lower as much as possible the cost. This is how list of possible tour is computed, then the cost of each tour is also computed, and tour are ordered so we keep the one with the lowest cost. I also tried to make this part random, selecting a random tour from the 1^{st} cost to the n^{th} cost, but it only made the code significantly longer without bringing significant improve. It is important to keep in mind that the first solution has to be **easy and fast to compute**, and the quality of it is secondary: the local search is very powerful to turn slightly non feasible solution into feasible solutions anyway. To finish a route, we add the final return to the depot and check the battery feasibility: if it is not feasible, we had the best charging station in the best places in the tour until the tour is feasible, if it fails to find a good spot for some reasons (some instances have tough configuration), in that case we keep the tour infeasible. Further results of that algorithm tested on the seventh first instances will be discussed in 5.

4.4 Using CVRP solutions as basis to general initial solution of EVRP

In this dissertation we will also cover another approach of first solution generation, in order to compare both and see which one gives the best results. The idea of this one is to solve the EVRP as CVRP using exact solving if possible: as a result we would have a non feasible solution with no violation of capacity and no violation of number of clients, only the battery side of things would be faulty. Then an algorithm that will be described below is used to fix every tour of the solution by adding the best charging stations, and this method tends to produce interesting results that we would have time to comment in the results part of the dissertation.

The method use to solve CVRP is based on a branch-and-price approach described in [17] and implemented by Romain Montagné and David Torres Sanchez in the **VRPy** library for python. The choice of branch-and-price may be slightly outdated, but it has the advantage of being able to find feasible solutions quite fast starting from the well-known Clark and Wright algorithm [2]. Another idea would be too use a new general solver published in 2020 [18] implementing a reviewed version of branch-cut-and-price. However, the solver was hard to install and to interface with python so I chose the simplest option as benchmark, keeping in mind the possibility of improve. Once the CVRP is solved, the following algorithm is applied to every tour to try to fix the battery problem of each tour 2:

Algorithm 2 Pseudo code of the constructive algorithm

Require: tour a tour from CVRP solution
Ensure: tour will be very likely to be fixed to respect battery capacity
while tour battery not OK or max iterations (10) reached **do**
1. Browse all nodes of the tour and find **the place where battery fall short**
2. Insert all possible charging stations at that place
3. Keep only the tour with the lowest cost

end while
RETURN tour {The tour battery is fixed except if the while loop stopped for max iterations}

4.5 Heuristic algorithm of local search to improve the solution

4.5.1 Overall presentation of the algorithm

Now that we manage to simple starting solutions, the goal is to implement a local search algorithm that would both turns non feasible into feasible solutions, and then improve the solution with a goal to reach the solution the closest to optimally as possible. The overall idea is to implement different operators which will make some random change on solution (but always keeping the number of clients the same in order to not diverge from feasibility). Those operators will then be selected randomly for each iteration, the *cost* and the *feasibility* of the new solution will be evaluate at each iteration, and the solution will be kept following certain conditions (the most basic one is when the new solution's cost is better than the best of all time, however there are also acceptance mechanism that accepts worsening moves to escape local optima). In order to evaluate *cost* and *feasibility* at the same time, we define a now *Cost* with the following formula:

$$Cost = cost + K \times feasibility, K > 10000 \tag{4.3}$$

By choosing $K > 10000$ in Eq.4.3, we guarantee the absolute priority of *feasibility* over *cost* since a solution with good cost but non feasible will be 10000x worse than a solution with poor cost but feasible. Keeping this in mind, the following algorithm is the pseudo-code of the local search algorithm (it will of course be explained in further details right after):

Algorithm 3 Pseudo code of the local search algorithm

Require: Starting solution named *sol*
  Threshold other move named *threshold*
  Percentage tolerance worsening move named *percentage_tolerance*
  The number of different operators named $nrb\_operator = 14$
  Max iterations named *max_iterations*
Ensure: Starting will be made feasible if not and improved
  $Cost \leftarrow Cost(sol)$
  $new\_sol \leftarrow sol$
  $best\_sol \leftarrow sol$
  $best\_cost \leftarrow cost$
  $list\_utility \leftarrow [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$
  $i = 0$
  while $i \leq max\_iterations$ **do**
    if $i - i\_last\_best \geq 0.05 \times max\_iterations$ **then**
      $list\_utility \leftarrow [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$ {Reset the chance of taking an operator}
    end if
    1. Pick a random item in *list_utility*. The index picked correspond to the operators that will be applied
    2. $sol \leftarrow operators(sol)$ {The operator is applied to the current solution}
    if The random number is over *threshold* **then**
      Continue {It has the effect of doing at least one other operator before evaluation}
    end if
    $cost\_new \leftarrow Cost(sol)$
    $i \leftarrow i + 1$
    if $cost\_new < best\_cost$ **then**
      $i\_last\_best \leftarrow i$
      $list\_utility \leftarrow list\_utility+$ index of the operator used
      $best\_cost \leftarrow cost$
      $best\_sol \leftarrow sol$
    end if
    if $cost\_new \leq cost$ **or** $cost\_new \leq best\_cost + percentage\_tolerance \times best\_cost$ **then**
      $cost \leftarrow new\_cost$
      $new\_sol \leftarrow sol$
    else
      $sol \leftarrow new\_sol$
    end if
  end while
  RETURN solution

The overall idea of the local search algorithm 3 is to select randomly an operator among the 14 available, to make a move on the current solution, to evaluate its new cost, and to keep it if it an improving moves. However, there are some more subtle mechanisms:

- There is a list of utility containing all index of operators, and every time an operator leads to an improved solution, we had that operator's index in the list. The new operator is always picked at random in that list, which means the algorithm tends to favour the operators that have already gave the best results

- However, operators are not always design for the same purposes: some may be better at making the solution feasible, some other may be better when we are very far from optimally, and some others may be good when the solution gets close to optimally. To avoid being stuck using wrong set of operators, when the algorithm does not accept any moves for 5% of the total number of iterations, the list of indexes of operators is reset to allow the algorithm to escape that situation.

- In the case of pure hill climbing algorithm (ie accepting **only non worsening moves**). However here the algorithm is making a difference between the all time best solution, and the current best solution. When an old time best solution is found, it is updated, but the current solution could accept worsening moves if it is $\leq +best\_sol \times percentage\_tolerance$

- The last mechanism to avoid the algorithm to be stuck is a mechanism of executing randomly two operators in a row before evaluating the solution. The algorithm is picking a random number between 0 and *threshold*, if the number is 0 another operator is applied to the solution before evaluating its cost.

The clear things that appears is that the ability of the program of improving solutions and escaping local optima is directly related to the set of parameters *percentage_tolerance* and *threshold* chosen by the user. The graphics of cost over iterations will be presented and commented in 5

4.5.2 Presentation of the 14 operators

There are 14 different operators doing moves inside tours of the solution or between tours of the solution. The moves can be very small and do not disturb the solution a lot, or be almost a complete re-writing of a part of the solution: this is made to be able to make small changes to converge to optimally and make great improvement at the beginning, but to escape local optima and reach optimally it will then be needed more important moves in operators. The operators will be ranked from approximately the one making the smallest moves to the one making the biggest changes in the solution.

The first operator is called swap visits inside tour:

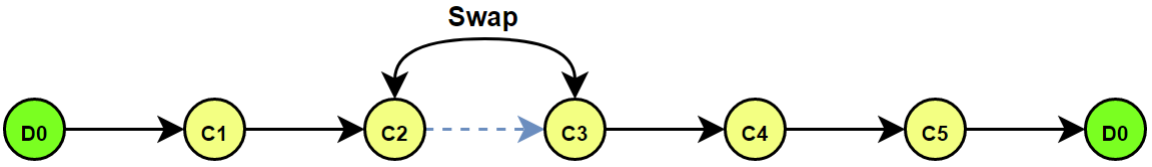

FIGURE 4.4: Operator: swap visits inside tour

In Fig.4.4 it is clearly explained what the operator does: it is selecting a tour randomly in a solution, and then it is swapping two visits inside this tour.

The second operator is called swap visits between tours:

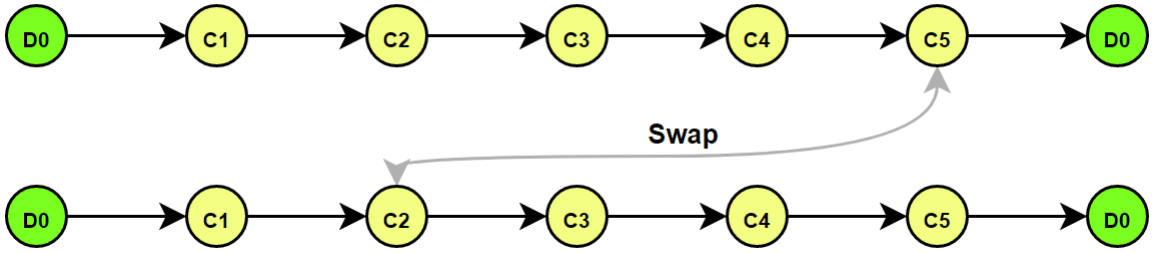

FIGURE 4.5: Operator: swap visits between tours

In Fig.4.5 the mechanism of the operator is explained: it is selecting two tours among the solution at random, and then it is randomly swapping two visits across the two tours. This operator can fail to be executed in case there is only one tour in the solution, in that case it is simply return the solution as it was.

The third operator is the lightest one, it is called moving one visit:

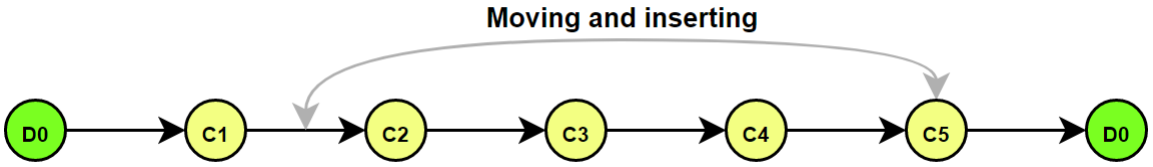

FIGURE 4.6: Operator: moving one visit

Moving one visit is the lightest operators presented in Fig.4.6, it is choosing a tour at random and it is just moving one visit in a random location of the selected tour.

The fourth operator is named inserting to another tour:

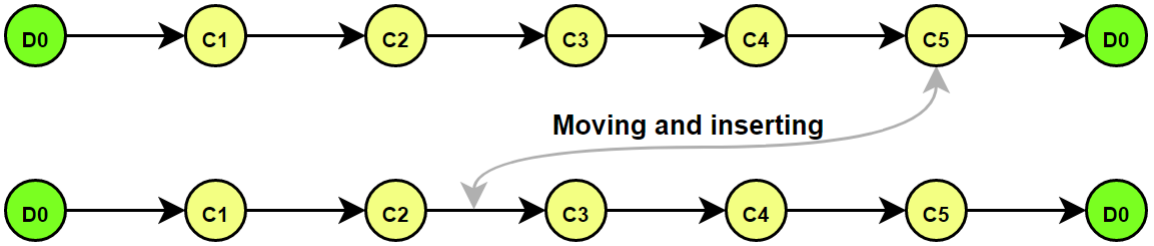

FIGURE 4.7: Operator: inserting to another tour

This is the first operators that is changing the number of visits by tour: it is randomly selecting a visits from one tour, and inserting it in a random place in another tour (Fig.4.7). In the special case where the first selected tour is only made by depot-visit-depot, the tour will be deleted to avoid the solution containing depot-depot tour. This is once again an operator that needs a solution with at least two routes.

The fifth operator is inserting to a new tour:

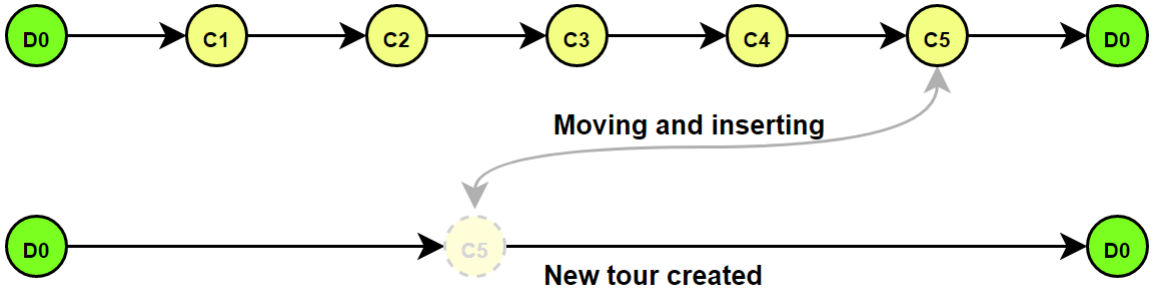

FIGURE 4.8: Operator: inserting to a new tour

This operators described in Fig.4.7 is similar to the previous one, but it is always inserting the chosen visits in a new tour, ie depot-visit-depot. It has the same issues with taking an item for an existing tour than the previous one, however this operator does not require to have two tours in the used solution.

The sixth operator is the first operator to deal with insertion of charging stations, which is mandatory to tackle battery issues in most EVRP. It is named adding station clever:

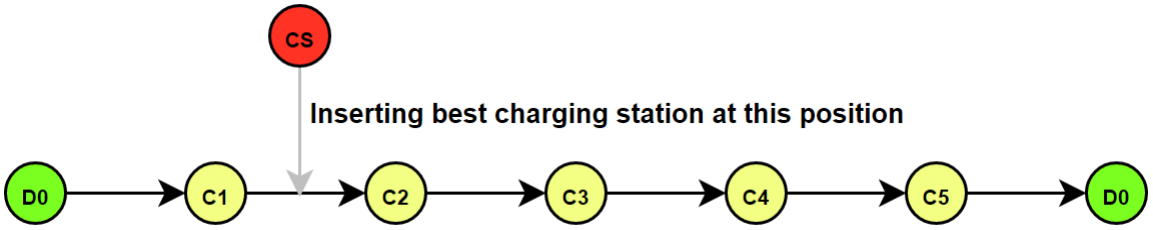

FIGURE 4.9: Operator: adding station clever

In Fig.4.9, the operator shown is adding the station that is making the smallest increase of cost in a random spot of a random tour of the solution.

The seventh operator is very similar to the first one as it is also oriented to insert charging stations, however the method is slightly different. It is called adding station random:

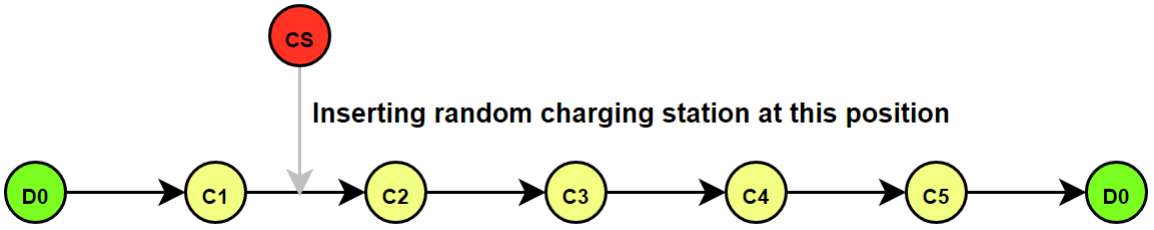

FIGURE 4.10: Operator: adding station random

In Fig.4.9, the operator shown is adding a random charging station in a random spot of a random tour of the solution.

The eight operator is here to balance the two previous one as it is the removing station operator:

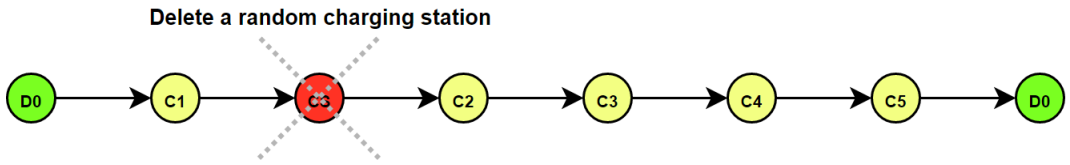

FIGURE 4.11: Operator: removing station

This operators randomly tries 10 times to pick a tour with at least one charging stations: if it fails, the solution remains unchanged, if it succeed, one of the charging station of the tour is randomly removed (Fig.4.11)

The ninth operator is introducing a new type of move and is called reversing:

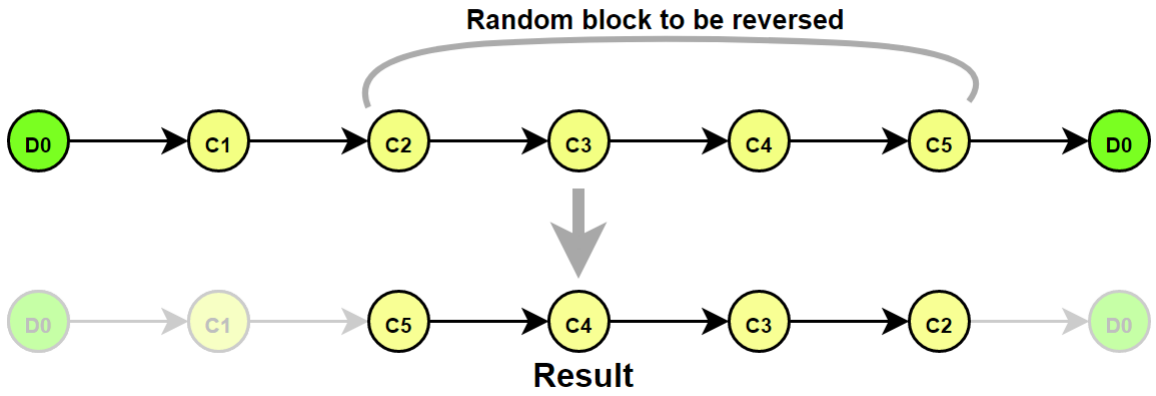

FIGURE 4.12: Operator: reversing

The reversing operator (Fig.4.12) is selecting a random tour among the solution, and then it is reversing a piece of that tour (the selected route has to be bigger than just depot-visit-depot or the operator cannot do anything). This operator does not change the overall cost of the set of visits that are reversed but it may solve feasibility problem for battery consumption: changing the order may allow to visit a clients that is closer to a charging station.

The tenth operator is one of the most disruptive operator and is named reverse inserting:

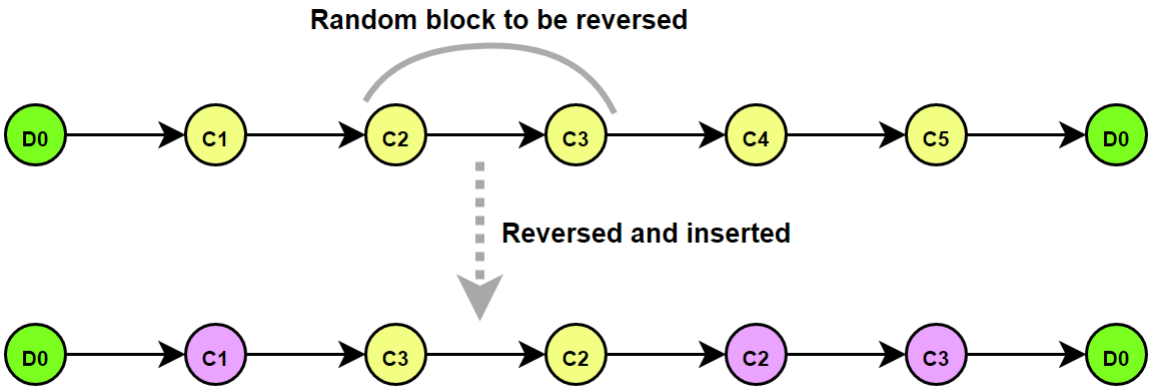

FIGURE 4.13: Operator: reverse inserting

This operator is selecting a random tour in which it selects a random set of visits. Then, this set of visits is reversed and is inserted to another random tour (Fig.4.13). This is a complicated operator since it requires both a solution with at least two tours, a first tour with more visits than just depot-visit-depot, and it also requires to handle the case where the left tour is too small and should be deleted: it has the potential to make massive change on the solution if the algorithm got stuck in local optima.

The eleventh operator is an evolution of the tenth one but paradoxically it is less problematic to implement: reverse swapping:

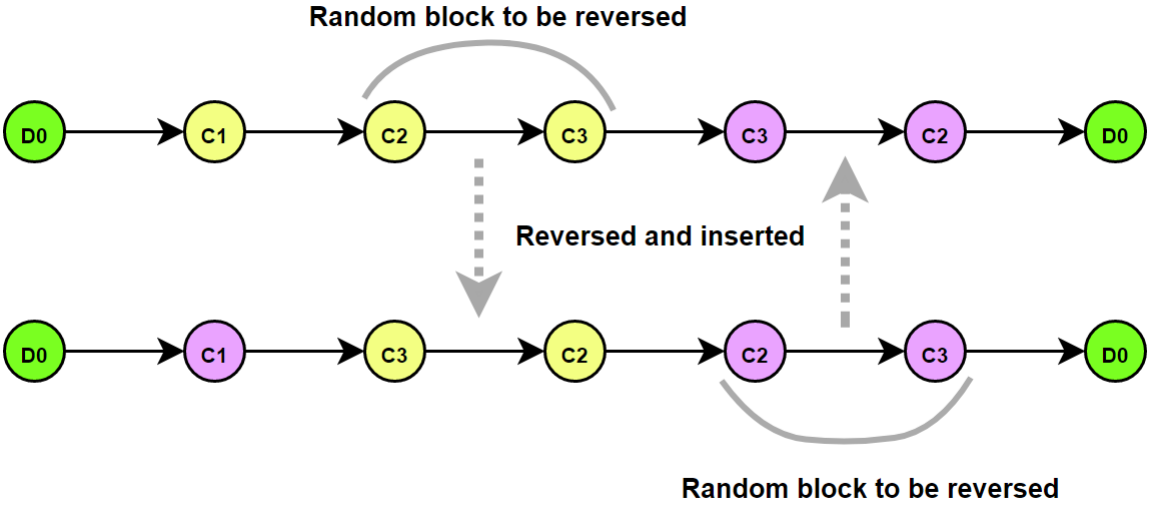

FIGURE 4.14: Operator: reverse swapping

The difference with the previous one is shown on Fig.4.14: two tours are selected, two set of visits are reversed and are then swap between tours. It no longer has the issue of emptying a tour, however, both tours needs to have more than one visit outside of depot.

The operator number 12 is the same as the previous one, without any action of reversing the selected set of clients, it is named swapping blocks:

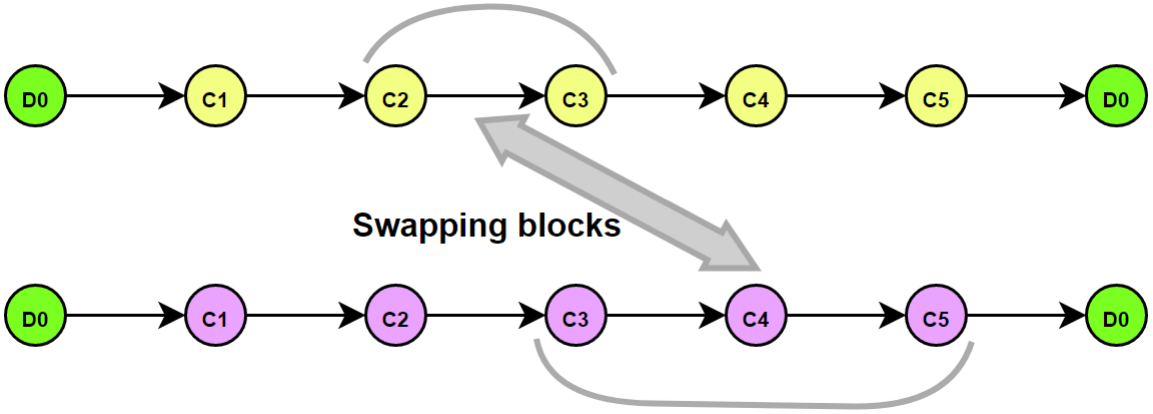

FIGURE 4.15: Operator: swapping blocks

The swapping blocks operator described in Fig.4.15 does not introduce any problem in the implementation, apart from the need of having a solution with at least two tours.

The thirteenth operator is like the tenth operator but it is not reversing the set of visit selected; it is called inserting block:

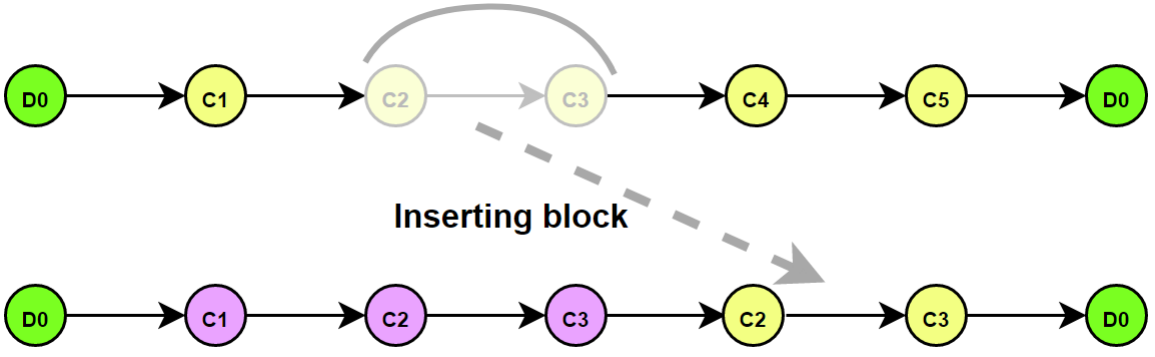

FIGURE 4.16: Operator: inserting block

This operator is taking a set of visits from on tour and inserting it in a random place of another tour (Fig.4.16). It requires both two routes and to have a special care in case the first route became empty and needs to be removed.

The last operator has a different philosophy from the previous one: it has been made to introduce chaos among on tour in order to escape local optima. It has been named ruin and recreate:

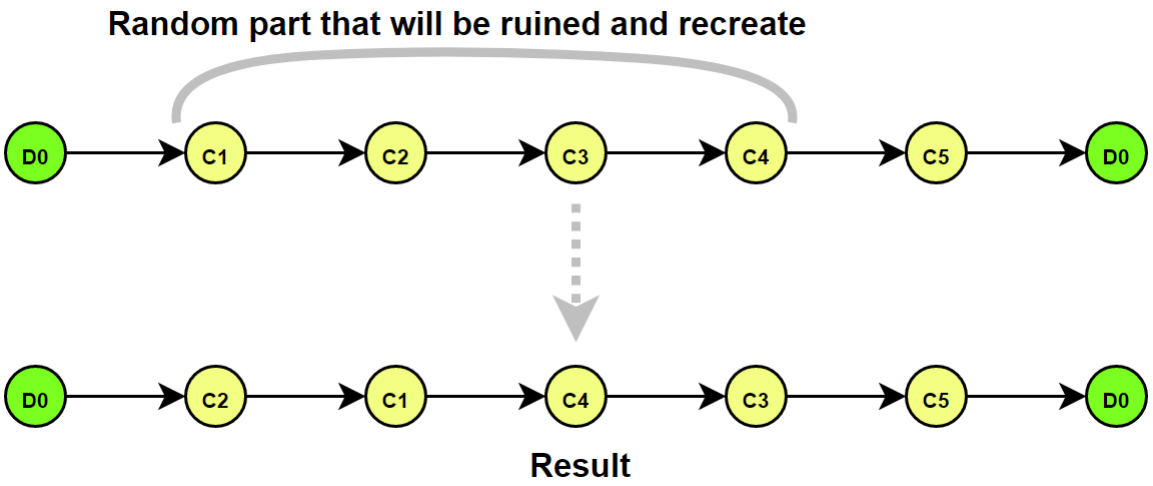

FIGURE 4.17: Operator: ruin and recreate

There is not special care for this operator as it is keeping the number of element of the selected tour the same, it is just shuffling the order as presented in Fig.4.17

It is important to notice that all the operators, even the one modifying the number of visits among a tour, are never modifying the overall number of client's visits. Indeed, the initial solution have been built to include all clients, so the number of clients is already optimal, it is just the order of visits that needs to be changed. The number of charging stations can change but it is balanced so it can go up or down. The overall utility of every operators in the runs I have made on different instances will be commented in the next Chapter (5).

This chapter was mostly focusing on describing the two different approach to generate starting solutions (greedy constructive heuristic and battery fixing on already solved CVRP solution) and its mechanism (the algorithm used to find the best place to insert the best charging station in an already existing tour). Then the overall local search algorithm have been presented along with its two parameters *threshold* and *percentage_tolerance* that will be set in the next chapter. To finish the chapter, the 14 different operators are presented with their goal and the obstacles I had to overpass for the implementation in the code.

Chapter 5

Results and performance of the heuristic method on benchmark instances

There are two important things to notice before going into the details of the results of all the presented algorithm and they are both related to computational time. All the experiments and runs of algorithms have been conveyed using my personal computer, a Desktop running Windows 10 Family with an Intel Core i5-4570 @3.20GHz with 6Mo of cache as CPU and 16Go of DDR3 RAM. The *Python* environment used is *Python 3.9* using the default interpreter of *Spyder 5.2*. Consequently, the computing power was limited, in order to have the time to run the algorithm a significant number of time I had to reduce the number of max iterations from $n \times 25000$ to a fix amount of 100000. The second decision I had to take is to focus only on the 7 instances of which we know the optimal values: it is better to benchmark an algorithm when we know a lower bound, and these are also smaller instances, which guarantee a wall time that fits the need of the experiments (minutes rather than hours). The goal of this part is to benchmark the full solving algorithm and both try to set the best parameters, but also analyse its behaviour and possibly understand the possibility of improvement remaining.

5.1 Comparison of first solution methods

In Chapter 4 two methods to find initial solutions are presented: the greedy constructive method, and fixing battery issue of CVRP solutions found by branch-and-price. Before looking at the actual cost results it is important to notice that the wall time of the greedy method is increasing rapidly with n, however for these small instances it is only taking few seconds, while solving the CVRP problem was always taking few minutes (fixing the battery problem is negligible compared to the time of solving CVRP). The results have been gathered in the following table:

Instance Name	**Cost Greedy Method**	**Cost CVRP Augmented**	**Optimal Cost**
E-n22-k4	484.35	385.39	384.68
E-n23-k3	730.91	654.27	573.13
E-n30-k3	573.78	516.37	511.25
E-n33-k4	1046.39	845.01	869.89
E-n51-k5	754.51	574.76	570.17
E-n76-k7	1113.56	767.45	723.37
E-n101-k8	1387.83	935.13	899.89

TABLE 5.1: Benchmark of first solution costs

The results gathered in Table5.1 are all affirming the same thing: solving CVRP and then fixing the battery issues is giving a better solution in term of costs for all the instances. The instances were that supremacy is the most obvious are: E-n76-k7 and E-n101-k8, the biggest instances of the 7 selected. However, the solution given for E-n33-k4 failed to fix the battery issue, this is why the cost is lower than the lower bound given by the optimal cost: the found solution is not feasible. The conclusion is that the greedy method is faster consistently gives a solution that is respecting the needed format, but it is often failing to approach optimality. On the other hands, fixing the CVRP solution is taking more time and do not fix the problem of failing to fix battery to reach optimality. The question that needs to be assessed now is: what is the influence of the proximity to optimality for the algorithm of local search: it will be answered later in this chapter.

5.2 Setting the parameters of the local search algorithm

In Chapter 4 the conclusion was that there were two parameters to set on the local search algorithm:

- *percentage_tolerance*: the percentage of the best currently known solution that defines the limit of acceptance of worsening moves: it will be accepted if it is less than the best cost known + that percentage of the best cost known.
- *threshold*: the algorithm draw a random value between 0 and *threshold* and when the drawn number is 0 the algorithm do another operator before evaluating the solution. The higher *threshold* is, the lower the chance of doing multiple operators before evaluating the solution.

The first instinct that I had on this parameters is that they are controlling the ability of the problem of rapidly changing and accept worsening solutions: if they are set too high there is a huge of diverging from optimality, but if they are too low the risk is to get stuck in a local optima. The methodology I have used to set this parameters was to try 5 sets of parameters 10 times on the smallest instance (E-n22-k4). The starting solution is the solution given by the greedy algorithm, assuming that the starting solution has no influence on the behaviour of the algorithm depending on its parameters. A set of parameters is given by $(threshold, percentage\_tolerance)$. The monitored metrics are the following:

- The average cost of found solutions
- The standard deviation of cost of found solutions
- A graphic of cost over iterations on average
- The average wall time
- The standard deviation of the wall time

I obtained the following results:

Parameters	**Avg Cost**	**Std Cost**	**Avg Time (s)**	**Std Time (s)**
(10, 0.05)	400.95	11.71	33.83	14.85
(10, 0.1)	511.43	44.07	108.18	28.07
(10, 0.01)	396.97	10.99	17.21	2.37
(5, 0.01)	392.01	7.97	17.75	4.53
(2, 0.01)	396.79	11.77	22.01	6.22

TABLE 5.2: Table of performance of different set of parameters

In the previous table (5.2) we can notice that there is no trade off to address between the performance of cost and the performance of time: the set of parameters (5, 0.01) is finding the lowest cost in average with the lowest standard deviation, which means that it is consistent over numerous runs. Even if the wall time is slightly bigger and more eager to vary than the set of parameters (10,0.01), we will put the focus on the performance of cost over the time since it is not significantly faster. In addition we can notice than the results are logical: increasing the chance of doing multiple operators allows to reach lower cost of solution if it is paired with a very selective condition of acceptance of worsening moves (0.01). If the condition of accepting worsening move is too easy (0.1) but the chance of making strong changes too low (10), then the algorithm is stuck and never manage to converge toward optimality.

We can also take a look at the graph of cost over iterations (not the best cost but the current cost, which means we can see the moment the algorithm is accepting worsening moves):

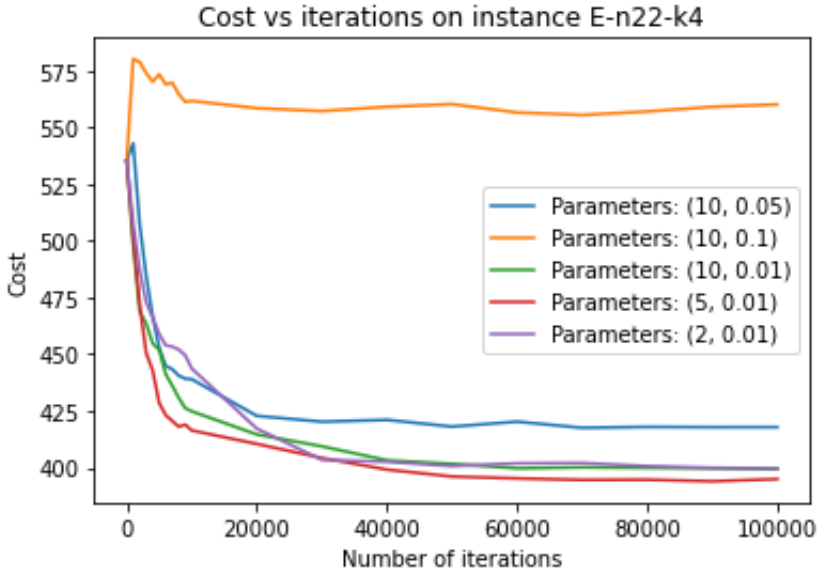

FIGURE 5.1: Average cost over iterations for E-n22-k4 for different set of parameters

This graph (Fig.5.1 that is presenting the cost over iterations for a different set of parameters is validating the previous observations: the set of parameters (5, 0.01) is the one reaching the lowest cost on average, but it is also the one that is consistently faster to converge than the other set of parameters. We can see even better on that graph that the set of parameters (10, 0.05) is not finding a way of converging toward optimality as explained before, the final solution is even worse than the starting solution.

The final decision is to keep $threshold = 5$ and $percentage\_tolerance = 0.01$ for the rest of the benchmarks since it performed better.

5.3 Influence of the starting point on the local search

Now that the parameters have been set, we can compute the result of the algorithm with such parameters on the 7 instances that we know the optimal cost. It will also be necessary to compare the results depending on the algorithm chosen as starting solution. All the statistics that will be presented (average and standard deviation) have been computed over 10 independent runs, which has been considered to be a good compromise between the time taken for computation and the reliability of the results. The starting solutions are always the same since they have been built using **deterministic** ways: consequently it is not cost efficient to recompute them for every runs of the algorithm. This is why I used the *Python* library named *Pickle* to serialise the solution (they are dictionaries in my code, and every element of the dictionary is a tour). I then wrote a *open pickle* function that allowed me to open the starting solution for every run instead of computing them again and again. The following table is summarizing the key statistics that I obtained after 10 runs for each starting points type for each of the 7 instances:

	From Greedy		**From CVRP to EVRP**		**Optimal value**
Instance Name	**Avg Cost**	**Std Cost**	**Avg Cost**	**Std Cost**	**Cost**
E-n22-k4	395.15	6.43	384.82	0.28	364.68
E-n23-k3	573.32	2.57	586.32	22.25	573.13
E-n30-k3	514.7	10.89	511.95	0.82	511.25
E-n33-k4	889.31	25.89	873.3	1.96	869.89
E-n51-k5	582	10.88	575.27	4.12	570.17
E-n76-k7	790.66	17.03	733.54	14.53	723.37
E-n101-k8	1019.57	31.77	5303.33	8625.59	899.89

TABLE 5.3: Benchmark of the algorithm using two different starting points

In the previous table (5.3) there are plenty of interesting observations to make. First we can notice that, on most instances, the average cost starting from the augmented CVRP solution is better, and the standard deviation is lower: it will find an explanation when we will look at the cost vs iterations graphs. However, for the instance E-n23-k3, the initial solution provided by the augmenter CVRP is not feasible and the algorithm somehow struggled to make it feasible, it is explaining why in that case the greedy solution as starter is outperforming it. The configuration is even worse on instance E-n101-k8: the algorithm cannot consistently make the starting solution from augmented

CVRP feasible, which leads to a very high standard deviation (the cost is computed via the formula 4.3). Speaking of the performance itself, it is possible to aim for optimality on average for the first three instances using this local search algorithm with this set of parameters, however the algorithm is struggling to reach a better solution when it is already in the range of 10% of the optimal solution (ex: instance E-n101-k8).

Even if starting from the CVRP augmented seem to be the best idea, it is important to look at cost vs iterations graphs as they are bringing another perspective on the table. On a classical instances with two feasible starting points, we would obtain the following graph (instance E-n22-k4):

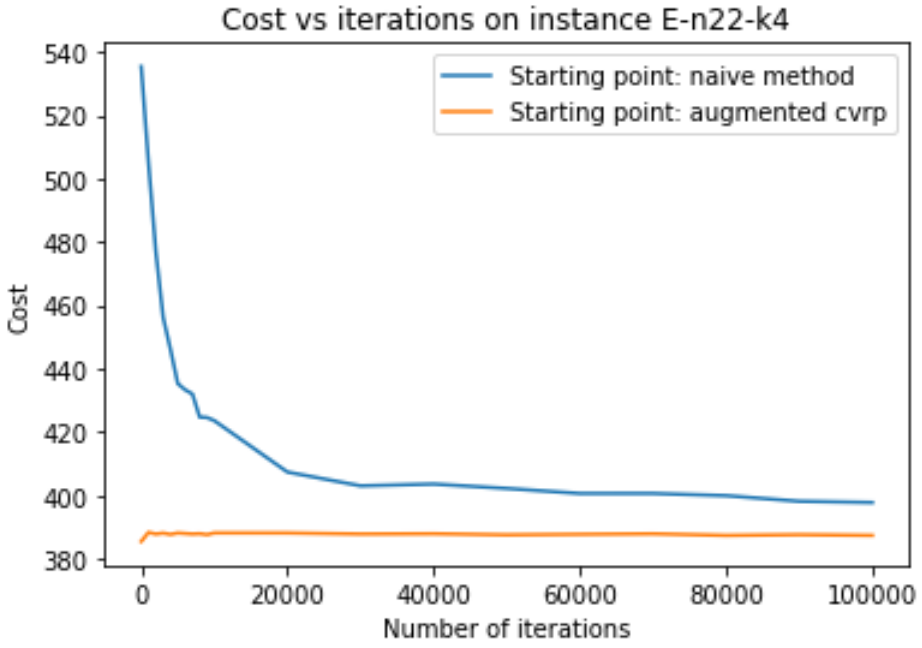

FIGURE 5.2: Cost vs iterations for E-n22-k4 comparing two starting points

On the previous figure (Fig.5.2), we are facing the most classic case: most of the improvement from the greedy solutions are made in the first 20000 iterations and then the algorithm is struggling to reach optimality. The augmented CVRP starting point gives a better result, however we can see that the algorithm is not able to optimise it at all: the local search is inefficient on such solutions.

Another case that is likely to happen is the case where the augmented CVRP solution is not feasible, it is the case with instance E-n23-k3:

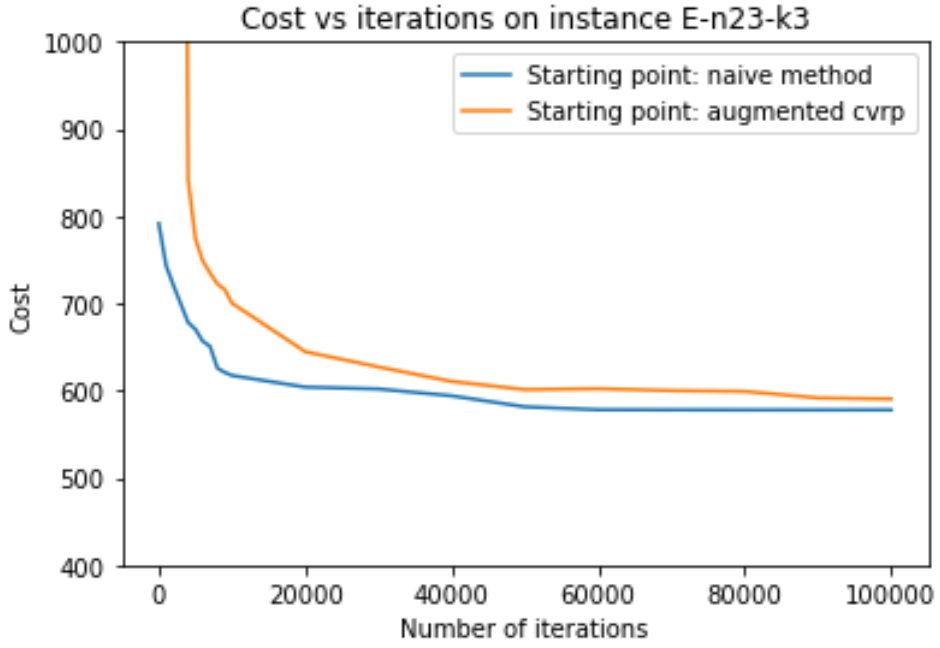

FIGURE 5.3: Cost vs iterations for E-n23-k3 comparing two starting points

But in that case exposed in Fig.5.3, the algorithm manage to make the solution feasible, however it is not managing to find a solution as good as it has found with the greedy naive method.

To finish, the very worst scenario is the case where the algorithm is not even able of consistently making the starting solution feasible, it can be the case starting from a non feasible solution with a bigger number of customers n, in instance E-n101-k8:

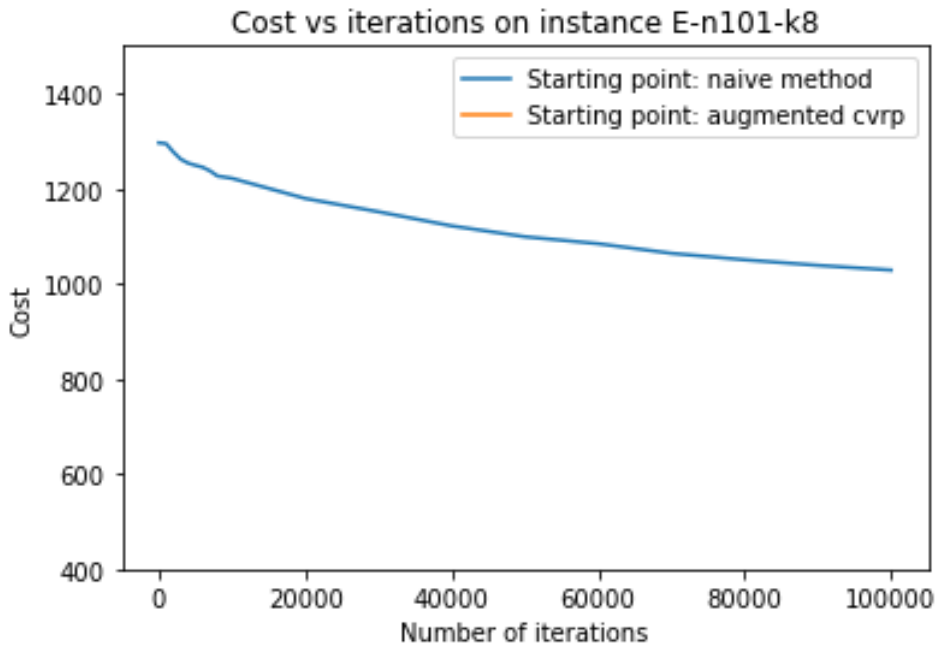

FIGURE 5.4: Cost vs iterations for E-n101-k8 comparing two starting points

We do not even see the line corresponding to augmented CVRP on Fig.5.4, in that case the algorithm is completely failing to make the solution feasible on average (it managed to do it on some runs, but it cannot be seen on average because the cost is penalised by a factor 10000 when the solution is not feasible). To summarise, the behaviour of the local search is consistent when the first solution given is the greedy solution: it always manages to make it feasible and then it tries to improve the cost. However, when the first solution is from CVRP augmented, the average cost is better but the local search is never able to improve it (this explains why the standard deviation is so low), and in some worst case scenario when the starting point is not feasible the local search is totally unable to converge to good solutions. In conclusion, the greedy algorithm remains the best starting point to work along with the local search I implemented.

5.4 Analyse of the utility of each operator

Currently in the algorithm the probability of selecting an operator depends on the previous improving moves: at the beginning each operator has the same probability of being

selected, and this probability is increasing if the operator has made a change that improved the solution previously. In order to do so, we also keep track of the *utility* of each operator: the score given corresponds to the number of time the specified operator lead to an accepted new solution. However it is important to notice that the choice have been made to only count the last operator in case of multiple operator in a raw, it would have been better to create a category of grouped operators to have more accurate results but it was hard to implement in the code. The utility have been measured on two instances: E-n22-k4 and E-n30-k3, using the same set of optimal parameters as before, computing statistics on 10 runs once again. For the first instance the utility scores obtained are reported below:

Operator Name	**Avg utility**	**Std utility**
swap visits inside tour	8.1	6.0
swap visits between tours	0.8	1.0
swap visits inside route	5.4	4.5
moving one visit	2.1	1.7
inserting to another tour	0.0	0.0
inserting to a new tour	0.4	0.5
adding station clever	0.0	0.0
adding station random	13.6	3.8
removing station	8.8	7.8
reversing	1.5	1.1
reverse inserting	0.2	0.4
ruining and recreate	1.6	1.5
inserting block	0.8	0.9
reverse swapping	1.2	1

TABLE 5.4: Utility of each operator for instance E-n22-k4

Keeping in mind that the utility function is only taken into account for the last operator before evaluation, we can take a look at the Table 5.4 and draw conclusions. The first conclusion is that we can clearly distinguish two category of operators: most of them are never called or less than 2 times in average while the other are having much higher utility score. Among the list of operator with a very low utility score, we observe inserting to another tour, inserting to a new tour, adding charging station clever, swap visits between tours, inserting block... The standard deviation of the utility score of those operators is really low, meaning that those operators are consistently not good at making improving moves. We observe that the operator with the highest utility scores are the operators making small moves (adding or moving only one visit at a time) among only one tour: it means that initial solutions tend to have the right composition of tour, however the order and the charging stations often need to be fixed. The score of utility of adding

charging station and removing stations is both high, it is possible that adding charging stations has a high utility score to reach feasibility, and then we have so remove some of them once the tour have been shuffled to reduce the total traveled distance (it is an hypothesis of the behaviour). The standard deviation of the utility cost of operator with a high score is also significantly higher, which lead me to think that depending on the configuration of solution we reached after the previous list of applied operator, one operator could be very relevant or totally irrelevant to make improvement (regardless of the proximity to the optimality).

To confirm the previous results, the experiment have also been conveyed on the E-n30-k3 instances (Table 5.5), and the results are exactly following the same logic, so I concluded that regardless of the shape of the instance, the pattern of solving could be considered the same, which may be a good information to improve the performance of the local search algorithm even further.

Operator Name	**Avg utility**	**Std utility**
swap visits inside tour	10.1	8.6
swap visits between tours	1.0	1.1
swap visits inside route	4.3	4.3
moving one visit	1.4	1.8
inserting to another tour	0.0	0.0
inserting to a new tour	0.0	0.0
adding station clever	0.0	0.0
adding station random	9.5	3.7
removing station	15.3	9.45
reversing	0.8	0.9
reverse inserting	0.2	0.4
ruining and recreate	1.8	1.7
inserting block	0.9	1.1
reverse swapping	0.8	0.8

TABLE 5.5: Utility of each operator for instance E-n30-k3

In this part we first started by comparing the results of the first solution's methods alone, then the goal was to identify the best possible set of parameters for the local search algorithm. Once those parameters were set, it was possible to benchmark the algorithm using both method as starting solutions, and I concluded that the algorithm was very efficient to make solution feasible, however it was sometimes struggling to improve the solution any further toward optimality. After discovering that the CVRP augmented method was not a good starting point because it was either not feasible enough or already too close to optimality, I finally analyse the utility score of every

operator: it is mostly the "lighest" one that are used, but there are not used consistently which may be an explanation why the algorithm is struggling to get closer to the optimal value: the disruptive operators are too chaotic and can only work once in a while with a specific configuration, and other operators are not changing enough to rich the optimal solution.

Chapter 6

Critics and work left to do

In the previous chapter (5) I have shown that the local search algorithm with the right set of parameters using a greedy constructed solution as starting point was consistently able to make solutions feasible and to improve it to reach 10% or less than the optimal value. However, there were still some remaining issues, questionable assumptions or possibility of improvement that will now be discuss in this chapter.

6.1 Possible improvement of the utility scores used

The first thing I wanted to discuss is the way the operator are selected to make moves. The current system is starting from a list with all the index from 1 to 14 representing the 14 different available operators: at the stage the algorithm is randomly selecting one of the index of the list, and this will be the used operator. There is also 20% chance that another move is selected according to the parameters we set for the algorithm. Then, we evaluate the solution, and if it is kept because it has all the requirements, the utility scores of the last operator used increases by one, and one more instance of its index is added to the list of index: the new probability of choosing that operator becomes $\frac{2}{15} \approx 0.13$ while the probability of all the other remains $\frac{1}{15} \approx 0.07$ etc... So probability of picking one operator over another depends a lot on the start: at the beginning it is easier to make an improving move, and then the operator will be favored for the rest of the run. This is why when the algorithm fails too many time in a row to improve the current solution, the list of index got reset to equality of probability in order to limit the influence of the start on the way operators are selected.

If this way of working has shown qualities and has prove its ability to make solutions feasible and better, it is clear that there are a lot of problems that may explain why it is sometimes too hard to improve solution even further:

The current way of counting utility score is ignoring combination of operators, it would be more accurate to count combination as combination, it could possible give the idea to add a fusion of both operators as a new operator and that could significantly improve the capacity to reach optimal value.

The current way of giving weight to operator is not giving consistent results: there are finite combinations of operators that allow to reach optimality, and the algorithm can be rapidly stuck because the impact of one good move is too big at the beginning and too flat at the end: the more index there are in the list, the less impact a good moves have and vice versa. It would be something to study to improve the quality of the solutions given by the algorithm.

Operators are varying from very small changes among the same tour to ground breaking changes between different tours, so the problem does not come from the operators that are able to converge to any optimal solution if they are called in the right orders (they are actually able to create every possible solutions). The current way of calling operator is rely too much on the random number generation, and it would be too long to try every possible distribution of the operators to evaluate which one performs the best: this is why very recent work (2022) are mentioning using ML to set the parameters of heuristic to improve drastically the solutions found ([19]. The other problem is the acceptance of worsening move and/or the way of computing the cost. The current algorithm is penalizing so much non feasible solutions that the only worsening moves accepted are the one keeping the solution feasible: it means that the algorithm is more likely to get stuck in a local optimum. However, we have to be careful when giving the algorithm the ability to accept non feasible solutions, it has to be only under certain circumstances or we will struggle to find final feasible solutions.

6.2 Criticising the generality of instances and benchmark

I have discussed above all the possible ways to keep on improving the actual algorithm, especially in order to be able to really improve the solution even further. But another problem needs to be solved: how representative of the reality are the instances used

in the competition? In case there is a detail that differs from reality, would it have a significant impact on the already built algorithm or will the performance would be still assumed to remain the same?

The main critic we can formulate about the given EVRP instances is that there are a bit artificial in the way of selecting battery capacity and of positioning charging stations. The battery capacity is set to allow the EV to move from the depot to the farthest client without falling short of battery, however this may not always be the case in real life, battery are limited and depending on the unit of distance used on instances it could be totally unrealistic. The second problem is that the charging stations are placed on the map to guarantee that there is at least one feasible solution, however in real life situation, especially now for early adopters, it is very likely that available charging stations are not at the right place, and that no feasible solution is even existing.

The algorithm have been built assuming that there would be one or two visits to charging stations and always a feasible solutions, but in real life this scenario is very unlikely to append. However, it may not be a problem since the algorithm is **flexible** thanks to its set of parameters and the way of selecting operators that can be changed: it is very likely that only by changing those parameters to fit the actual need the algorithm would be easily adapted to solve real life situations, so we can consider that the current algorithm would be **relevant anyway**.

6.3 Model of the battery consumption: too simplistic?

Another important problem of the model when we would want to solve real case scenario is the model of battery consumption: assuming that the consumption is linear is too ideal, it is supposed to depend a lot on the acceleration of the vehicles, the total weight of it (which vary a lot depending on the remaining load), the slope of the road and the speed (among the other).

The model I will explain in the dissertation is partially based on the work of Zhang et al.(2018) [20]. Using the second law of Newton we know that:

$$\sum \vec{F_{ext}} = M \cdot \vec{\ddot{x}} \tag{6.1}$$

In the following formula we will adopt the following notation:

- $\ddot{x}$ is the acceleration of the vehicles (in m/s^2)
- $\dot{x}$ is the speed of the vehicles (in m/s)
- **M** is the weight of vehicles (in kg) (it is fixed but it should vary depending on load)
- **g** is the gravitational constant (9.81 m/s^2)
- θ_r is the angle of slope of the route (in rad)
- θ_s the angle of the air on the car (in rad)
- **A** is the vehicles surfaces facing the force of the air (in m^2)
- C_r is the rolling resistance coefficient (no unit)
- C_d is the dragging resistance coefficient (no unit)
- ρ is the air density (1.2 kg/m^3)

According to the Fig.6.1 that has been taken from [21], there are 3 main forces that are applied to the EV: the weight force decomposed according to the slope angle, the air resistance and the drag resistance.

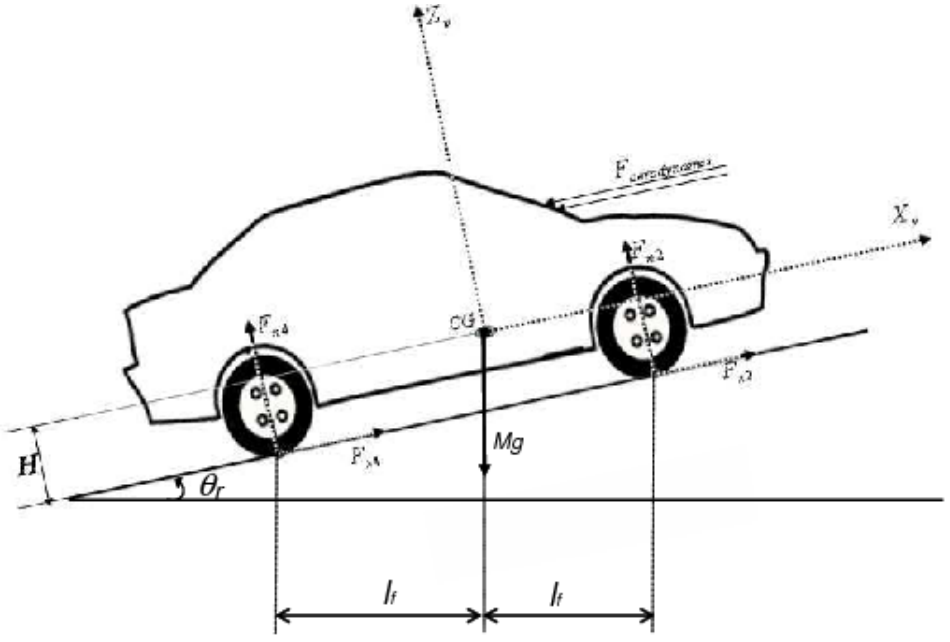

FIGURE 6.1: Schema of the force applied on the EV

Applying the second law of newton keeping in mind that $Power = Force \times speed$ we got the following equation on the $\vec{x}$ axis:

$$P_t = M(t)\ddot{x}\dot{x} + M(t)\dot{x}g\sin\theta_r + 0.5C_dA\rho\dot{x}^3 + M(t)gC_r\cos\theta_s \tag{6.2}$$

One first estimation on the instant power used for by an EV is given in the Eq.6.2 (the energy will be computed by taking the integral of that instant power over the time of travel). We can also take into consideration the yield of 0.9 of electrical vehicles. The important things to notice are that the weight **M** can make that power vary a lot, and the weight will vary from simple to double depending of the load. Assuming constant speed is a wrong assumption, consequently the speed will also be a big factor. We may also notice that for $theta_r$ between π *rand* and 2π *rad* the slope is negative and the force of weight is actually helping the EV to consume less energy. In any case, the logical conclusion is that linear consumption of the distance is not a reasonable approximation and the model of battery consumption would need to be changed when trying to solve real case scenario to avoid the model to create solution that are feasible in theory but not in practice.

6.4 Speeding up the code using JIT

One problem that was not presented until now of the code is purely technical: it is not related to the pseudo code of the local search algorithm, but only to the python implementation. *Python* is an interpreted programming language which makes it unfortunately much slower than compiled programming language: this the reason why I was not able to run the algorithm the right amount of maximum iterations and not on the biggest instances. In this section I wanted to discuss all the technical possibilities that we have to make the code running faster without changing any features of it.

The first thing to explore anyway would be to profile the code over one run: a very good library in python that does that task is called *Line Profiler*. After a run the profiler is giving the wall time of every line of the code, along with the number of time each line has been called and a percentage of the total wall time. Most of the time it allows to identify **bottleneck** on the code, there are then multiples way of solving them to improve drastically the speed of the code.

The first thing that could be tried is trying not to evaluate part of the code that we already know the solution: for example the *feasibility* function and the *cost* function are called at every evaluation, which means that it will make a significant improve to improve them. It is possible to speed those two functions by only computing the tour that has been influenced by the operator, since we already know the cost and the feasibility of the other tour from the previous evaluation.

The second thing, that will be even more adapted to my concrete case will be to pre-compile the operators in order to make every call of operator much faster: it will have the effect of making the cost much faster (I already noticed than depending on which operator is called the most the speed of the code could be very different). The problem of *Python* is that variable assignment is more flexible, however it also makes the code slower: when the type of the variables is already assigned, and that the size of list is fixed instead of using python list, it allows the computer to manage RAM allocation much better and to make the code significantly smarter. This also explains why the code can get slower and slower over iterations: the RAM may be overloaded over iterations. The first solution would be to rewrite the code of every operators in C, then compiled it and call it directly in *Python*. It is a very effective method, but it requires skill in low level programming which are not easily in the reach of anybody. So the solution I chose is using a Just In-time Compiler (JIT) called *Numba*. This library adds a new decorator to *Python* which allows the user to rewrite the code specifying the type of every variables in advance to allow the JIT to compile the function once and then call the speed up version of that function: it is quite easy to use since it is based on python's syntax and it is much quicker to learn than C.

The last possible thing is even more technical but it can in theory be used to speed up part of the code. It is called code parallelism, it is based on the idea that a CPU has multiple physical and logical cores that can run in parallel. It means that multiple actions of the code could be run in parallel using threads in order to speed up the code, but all those actions have to be independent. The principle of this is illustrated in the Fig. 6.2 below. Fortunately, evaluating functions (*cost* and *feasibility*) can be run in parallel since every tour of the solution are independent. The good news is that just in time compiling could be coupled with parallelism coding to make the code even faster.

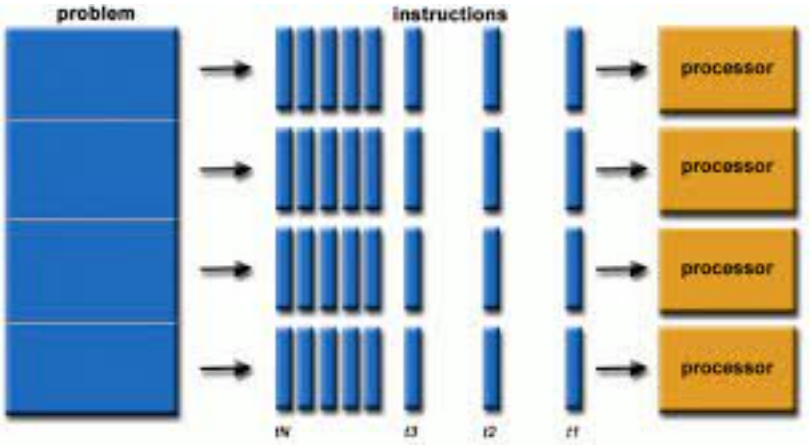

FIGURE 6.2: Principle of code parallelism

Implementing all those tricks would make the cost much faster and would make any further work on improving the code much faster and convenient, it will be possible to try the algorithm even on the biggest instances.

Chapter 7

Conclusion

The goal of this dissertation was to propose a consistent and efficient method of solving EVRP from the CEC-12 Competition. It is especially relevant in the context of making effort against climate change, since EVs are not emitting any CO2, it is better for the overall climate and also for the near environment of cities that already have a very polluted air.

It starts with a literature review in Chapter 2 that is presenting the evolution of VRP as well as the recent effort to take the environmental side of things into account in recent papers: however the overall literature is still lacking papers about EVRP, both the clear formulation but also effective way of solving it.

The Chapter 3 is dedicated to the formulation of the EVRP: it is explained there all the proximity between EVRP and CVRP but the problem of battery consumption is also described very well. New constraints appear with the need to visit charging stations every time that the battery fall short, and it leads to a complicated MIP formulation with the current level of battery as variable since it is not fixed and cannot be used as a parameters of the problem.

It is in Chapter 4 that the local search heuristic method is entirely described, along with the two constructive methods used to build starting solution.

In Chapter 5 we discovered that the given algorithm was able to consistently make solution feasible and improve the cost, but it is hard to escape local optimum and to reach optimality since the penalty on non feasible solutions lead the algorithm to accept none of the non feasible solution. The greedy algorithm has been proven to give the best

starting solution, and the global algorithm is able to reach 10% of the optimal solution on the first 7 instances.

The last chapter, Chapter 6, is dedicated to express all the critics toward the model, as well as the explanations and the possible way of improving issues. The implementation of the charging stations is proven to be too artificial, however it may not be a problem since the algorithm is quite flexible with the parameters that can be set. It is also very important to use a more realistic battery consumption model since the linear approximation is very strong: it will not be an issue for the performance of the algorithm but it will modified the optimal solution to stick to the reality. Finally, some tips to make the Python implementation faster are also discussed.

Appendix A

Code Listings

```
### Import libraries ###

### Import libraries ###

import time
import copy
import pickle
import math as m
import numpy as np
import pandas as pd
import random as rd
import matplotlib.pyplot as plt
from itertools import permutations
from networkx import DiGraph, from_numpy_matrix, relabel_nodes,
    set_node_attributes
from vrpy import VehicleRoutingProblem

class parse_instance:

    def __init__(self,
                 path_evrp_dir = "C://Users//Blaise//Business Analytics//Term
    3//Solving Electrical Vehicles Routing
    Problem//Solving-EVRP-MSCI-Business-Analytics-//evrp-benchmark-set",
                 instance_name = "E-n22-k4.evrp"):

        ### Parsing first line to collect instance parameters ###
```

```
        # Opening the EVRP instance
        self.__instance_name = instance_name
        f = open(path_evrp_dir + "//" + instance_name, "r")

        # Creating a dictionary to store instance parameters
        self.__dict_param = dict()

        # Reading the parameters of the instance one by one
        self.__dict_param["Name"] = f.readline()[6:-2]
        self.__dict_param["Comment"] = f.readline()[9:-1]
        self.__dict_param["Type"] = f.readline()[6:-2]
        #self.dict_param["Opt_val"] = float(f.readline()[14:-2])
        self.__dict_param["Opt_val"] = f.readline()[14:-2]
        self.__dict_param["Nbr_EV"] = int(f.readline()[10:-2])
        self.__dict_param["Dim"] = int(f.readline()[10:-2])
        self.__dict_param["Nbr_stat"] = int(f.readline()[9:-2])
        self.__dict_param["Capacity"] = float(f.readline()[9:-2])
        self.__dict_param["Energy_cap"] = float(f.readline()[16:-2])
        self.__dict_param["Energy_cons"] = float(f.readline()[20:-2])
        self.__dict_param["Edge_weight_format"] = f.readline()[20:-2]

        self.__nbr_nodes = self.__dict_param["Dim"] +
    self.__dict_param["Nbr_stat"]

        ### Parsing coordinate of nodes ###
        f.readline()
        self.__dict_coord = {}

        for ind in range(0, self.__nbr_nodes):
            l = f.readline().split()
            self.__dict_coord[ind] = (int(l[1]),int(l[2]))

        ### Parsing demand of clients ###
        f.readline()
        self.__dict_demand = {}
        self.__list_cust = []
        self.__coord_cust_x = []
        self.__coord_cust_y = []

        for ind in range(0, self.__dict_param["Dim"]):
            l = f.readline().split()
```

```
            demand = int(l[1])
            if demand > 0:
                self.__list_cust.append(ind)
                self.__coord_cust_x.append(self.__dict_coord[ind][0])
                self.__coord_cust_y.append(self.__dict_coord[ind][1])
            self.__dict_demand[ind] = demand

        ### Parsing index of charging stations ###
        f.readline()
        self.__list_stat = []
        self.__coord_stat_x = []
        self.__coord_stat_y = []

        for ind in range(self.__dict_param["Nbr_stat"]):
            s = f.readline()
            self.__list_stat.append(int(s)-1)
            self.__coord_stat_x.append(self.__dict_coord[int(s)-1][0])
            self.__coord_stat_y.append(self.__dict_coord[int(s)-1][1])

        f.close()

        ### Generating useful variables ###

        self.__C = self.__dict_param["Capacity"]
        self.__Q = self.__dict_param["Energy_cap"]
        self.__h = self.__dict_param["Energy_cons"]

        self.__dist_array = create_distance_matrix(self.__dict_coord)

    def get_coord_cust_x(self):
        return self.__coord_cust_x

    def get_coord_cust_y(self):
        return self.__coord_cust_y

    def get_coord_stat_x(self):
        return self.__coord_stat_x

    def get_coord_stat_y(self):
        return self.__coord_stat_y
```

```
    def get_dict_coord(self):
        return self.__dict_coord

    def get_dict_demand(self):
        return self.__dict_demand

    def get_dist_array(self):
        return self.__dist_array

    def get_instance_name(self):
        return self.__instance_name

    def get_list_cust(self):
        return self.__list_cust

    def get_list_stat(self):
        return self.__list_stat

    def get_C(self):
        return self.__C

    def get_Q(self):
        return self.__Q

    def get_h(self):
        return self.__h

    def get_plot_values(self):
        return self.__coord_cust_x, self.__coord_cust_y, self.__coord_stat_x,
self.__coord_stat_y, self.__dict_coord, self.__instance_name

    def get_instance_values(self):
        return self.__dict_coord, self.__dict_demand, self.__dist_array,
self.__list_cust, self.__list_stat, self.__Q, self.__C, self.__h

    def get_nbr_ev(self):
        return self.__dict_param["Nbr_EV"]

    def get_Dim(self):
        return self.__dict_param["Dim"]
```

```
class heuristic():

    def __init__(self, dict_coord, dict_demand, dist_array, list_cust ,
    list_stat, Q, C, h):

        self.__dict_coord = dict_coord
        self.__dict_demand = dict_demand
        self.__dist_array = dist_array
        self.__list_cust = list_cust
        self.__list_stat = list_stat
        self.__Q = Q
        self.__C = C
        self.__h = h

        self.dict_sol = {}

    def is_tour_capacity_ok(self, route):

        ui = self.__C

        for ind in route:
            #i = route[ind]
            #j = route[ind]
            list_cust = self.__list_cust[:]
            dict_demand = self.__dict_demand

            #print("ind: " + str(ind))
            if ind in list_cust and ind != 0:
                ui -= dict_demand[ind]
                #print("ui: " + str(ui))

            #print(str(i)+"-"+str(j)+ " ui: " + str(ui))

            if ui < 0:
                return False

        return True

    def capacity_violation(self, route):

        ui = self.__C
```

```
        for ind in route:
            list_cust = self.__list_cust[:]
            dict_demand = self.__dict_demand

            if ind in list_cust and ind != 0:
                ui -= dict_demand[ind]

        if ui < 0:
            return ui
        else:
            return 0


    def is_tour_battery_ok(self, route):

        yi = self.__Q

        for ind in range(1, len(route)):
            i = route[ind - 1]
            j = route[ind]
            Q = self.__Q
            h = self.__h
            dist_array = self.__dist_array
            list_stat = self.__list_stat[:]


            yi -= h*dist_array[i][j]
            #print(str(i)+"-"+str(j)+ " yi: " + str(yi))

            if yi < 0:
                return False

            if j in list_stat or j == 0:
                yi = Q

        return True

    def battery_violation(self, route):

        yi = self.__Q
```

```
        list_remaining_battery = []

        for ind in range(1, len(route)):
            i = route[ind - 1]
            j = route[ind]
            Q = self.__Q
            h = self.__h
            dist_array = self.__dist_array
            list_stat = self.__list_stat[:]
            yi -= h*dist_array[i][j]
            #print(str(i)+"-"+str(j)+ " yi: " + str(yi))

            """
            if yi < 0:
                return False
            """
            list_remaining_battery.append(yi)

            if j in list_stat:
                yi = Q

        min_battery = min(list_remaining_battery)

        if min_battery >= 0:
            return 0
        else:
            return min_battery


    def cost_station_instertion(self, vi, vj, station_ind):
            vf = station_ind
            dist_array = self.__dist_array
            return dist_array[vi][vf] + dist_array[vf][vj] - dist_array[vi][vj]

    def insert_best_stat(self, route, vi, vj, vj_last=False):

        list_cost = []
        list_stat_temp = self.__list_stat[:] + [0]
```

```
        for station_id in list_stat_temp:

            list_cost.append(self.cost_station_instertion(vi, vj, station_id))


        min_cost = min(list_cost)
        chosen_stat_ind = list_stat_temp[list_cost.index(min_cost)]

        if vj_last:
            vi_index, vj_index = route.index(vi), len(route) - 1
        else:
            vi_index, vj_index = route.index(vi), route.index(vj)

        new_route = route[:vi_index+1] + [chosen_stat_ind] + route[vj_index:]

        while not self.is_tour_battery_ok(new_route) and len(list_cost) >= 1:

            if len(list_cost) == 1:
                min_cost = list_cost[0]
                chosen_stat_ind = list_stat_temp[list_cost.index(min_cost)]
                new_route = route[:vi_index+1] + [chosen_stat_ind] +
    route[vj_index:]
                list_cost.remove(min_cost)
            else:
                del(list_cost[list_cost.index(min_cost)])
                min_cost = min(list_cost)
                chosen_stat_ind = list_stat_temp[list_cost.index(min_cost)]

                new_route = route[:vi_index+1] + [chosen_stat_ind] +
    route[vj_index:]


        if len(list_cost) > 0:
            return new_route

        else:
            return "Not feasible"


    def is_stat_reachable(self, last_node, yi):
```

```
        list_stat = self.__list_stat[:]
        dist_array = self.__dist_array
        h = self.__h


        for stat_node in list_stat:
            if yi - h * dist_array[last_node][stat_node] >= 0:
                return True

        return False

    def greedy_solve(self, is_stochastic=False):

        st = time.time()

        # Setting variables of the problem
        dict_coord = self.__dict_coord
        dist_array = self.__dist_array
        dict_demand = self.__dict_demand
        list_cust = self.__list_cust[:]
        list_stat = self.__list_stat[:]
        h = self.__h
        C = self.__C
        Q = self.__Q

        # Setting visited and non visited client list
        visited_clients = []
        non_visited_clients = list_cust[:]
        nbr_cust = len(list_cust)

        # Setting dictionnary containing solution
        dict_sol = {}

        # Setting temporary variables
        curr_route_ind = 0
        route = [0]

        # Initial plot
        instance_name = inst.get_instance_name()
        coord_cust_x = inst.get_coord_cust_x()
        coord_cust_y = inst.get_coord_cust_y()
```

```
        coord_stat_x = inst.get_coord_stat_x()
        coord_stat_y = inst.get_coord_stat_y()
        dict_coord = inst.get_dict_coord()

        available_move = True

        while len(visited_clients) < nbr_cust and available_move:

            # Creates a list of ordered feasible candidates
            list_candidates = self.create_list_feasible_candidates(route,
    non_visited_clients)
            ordered_cost_list, dict_candidates =
    self.cost_candidates(list_candidates)

            # If there's at least one remaining candidates
            if len(list_candidates) > 0:

                if is_stochastic:
                    rank_cost = rd.randint(0, min(len(list_candidates)-1, 2))
                else:
                    rank_cost = 0

                route = dict_candidates[ordered_cost_list[rank_cost]]
                #route = list_candidates[0]
                dict_sol[curr_route_ind] = route
                #dict_ax_route, fig, ax = self.update_plot(dict_ax_route,
    dict_sol, fig, ax, curr_route_ind)
                visited_clients, non_visited_clients =
    self.update_visited_clients(route, visited_clients, non_visited_clients)[:]

            # Else we terminate the route and create a new one
            else:
                if route == [0]:
                    available_move = False

                else:
                    route = self.ending_route(route)[:]
                    dict_sol[curr_route_ind] = route
                    curr_route_ind += 1
                    route = [0]
```

```
        print(non_visited_clients)

        if not available_move:
            for clients in non_visited_clients:
                dict_sol[curr_route_ind-1] = dict_sol[curr_route_ind-1][:-1] +
    [clients] + [0]
        else:
            dict_sol[curr_route_ind] =
    self.ending_route(dict_sol[curr_route_ind])[:]

        et = time.time()
        elapsed_time = et - st
        print('Execution time solve_naive:', elapsed_time, 'seconds')

        return dict_sol

    def cost_candidates(self, list_candidates):
        cost_list = [0]*len(list_candidates)
        dict_candidates = {}

        for ind, candidates in enumerate(list_candidates):
            cost = self.cost_route(candidates)
            cost_list[ind] = cost
            dict_candidates[cost] = candidates

        return sorted(cost_list), dict_candidates

    def ending_route(self, route):
        if self.is_tour_battery_ok(route + [0]):
            return route + [0]
        else:
            route = route + [0]
            while not self.is_tour_battery_ok(route):
                route = self.insert_stat_in_best_position(route)[:]
        return route

    def update_visited_clients(self, route, visited_clients,
    non_visited_clients):

        visited_clients = visited_clients[:]
        non_visited_clients = non_visited_clients[:]
```

```
        list_cust = self.__list_cust[:]

        for node in route:
            if node not in visited_clients and node in list_cust:
                visited_clients.append(node)
                non_visited_clients.remove(node)

        return visited_clients[:], non_visited_clients[:]

    def create_list_feasible_candidates(self, route, non_visited_clients):

        list_candidates = []

        for client in non_visited_clients:
            list_candidates.append(route + [client])

        list_candidates = self.remove_capacity_not_ok(list_candidates)[:]

        if len(list_candidates) <= 0:
            return []

        list_candidates = self.fix_battery_among_candidates(list_candidates)[:]
        if len(list_candidates) <= 0:
            return []

        list_candidates = self.remove_empty_routes(list_candidates)[:]

        if len(list_candidates) <= 0:
            return []

        list_candidates = self.remove_far_from_stat(list_candidates)[:]

        if len(list_candidates) <= 0:
            return []

        return list_candidates


    def create_list_candidates(self, route, non_visited_clients):

        list_candidates = []
```

```
        for ind in range(1, len(route)):
            for node_client in non_visited_clients:
                list_candidates.append(route[:ind] + [node_client] +
    route[ind:])

        for node_client in non_visited_clients:
            list_candidates.append(route + [node_client])

        return list_candidates


    def remove_capacity_not_ok(self, list_candidates):
        """
        This function takes the list of candidates and return a list of all the
        candidates that are note exceeding the capacity limit.

        Parameters
        ----------
        list_candidates : list
            List of list. Every sub list correspond to a candidate routes with
            an inserted nodes.

        Returns
        -------
        list
            The returned list only keeps candidates where the capacity is not
            exceeded.

        """
        return list(filter(self.is_tour_capacity_ok, list_candidates))

    def remove_battery_not_ok(self, list_candidates):
        """


        Parameters
        ----------
        list_candidates : TYPE
            DESCRIPTION.
```

```
        Returns
        -------
        TYPE
            DESCRIPTION.

        """
        return list(filter(self.is_tour_battery_ok, list_candidates))

    def is_close_from_stat(self, route):
        """
        The goal of this function is to test if the given route allow to reach
        a charging station or the depot next or not.

        Parameters
        ----------
        route : list
            List representing the order of nodes reached by this route.

        Returns
        -------
        bool
            True is returned if this route can possibly reach a charging
    station
            or the depot after its tail. Otherwise, False is returned.

        """
        list_stat = self.__list_stat[:]
        list_stat_dep = [0] + list_stat
        dist_array = self.__dist_array
        Q = self.__Q
        h = self.__h

        yi = Q
        last_node = route[-1]
        for ind in range(1,len(route)):
            i = route[ind-1]
            j = route[ind]
            yi -= h * dist_array[i][j]

        for stat_node in list_stat_dep:
            if yi - h * dist_array[last_node][stat_node] >= 0:
```

```
                return True

        return False

    def remove_far_from_stat(self, list_candidates):
        """


        Parameters
        ----------
        list_candidates : TYPE
            DESCRIPTION.

        Returns
        -------
        TYPE
            DESCRIPTION.

        """
        return list(filter(self.is_close_from_stat, list_candidates))

    def fix_route_battery(self, route):

        is_battery_ok = self.is_tour_battery_ok(route)

        if is_battery_ok:
            return route

        else:
            while not is_battery_ok:
                route = self.insert_stat_in_best_position(route)[:]
                is_battery_ok = self.is_tour_battery_ok(route)

            return route

    def fix_battery_among_candidates(self, list_candidates):
        return list(map(self.fix_route_battery, list_candidates))

    """
    def create_candidates_with_battery_ok(self, list_candidates):
```

```
        Parameters
            ----------
        list_candidates : TYPE
            DESCRIPTION.

        Returns
        -------
        TYPE
            DESCRIPTION.


        # Insert a station in any routes where it's needed
        for ind, route in enumerate(list_candidates):
            list_candidates[ind] = self.insert_stat_in_best_position(route)[:]

        #print("\n" + str(list_candidates) + "\n")

        # Remove impossible to insert routes
        list_candidates = self.remove_empty_routes(list_candidates)[:]

        #print(str(list_candidates) + "\n")

        # If battery is still not ok, remove route
        list_candidates = self.remove_battery_not_ok(list_candidates)[:]

        # Reorder candidates after insertion of the station
        tuple_candidates = self.order_candidates_by_cost(list_candidates)

        return list(tuple_candidates)
    """

    def remove_empty_routes(self, list_candidates):
        """


        Parameters
        ----------
        list_candidates : TYPE
            DESCRIPTION.
```

```
        Returns
        -------
        TYPE
            DESCRIPTION.

        """
        return list(filter(is_route_not_empty, list_candidates))

    def insert_stat_between(self, route, vi, vj, vj_last=False):

        list_charging_stat_dep = [0] + self.__list_stat[:]
        list_new_routes = []
        list_cost = []

        if vj_last:
            index_vi = route.index(vi)
            index_vj = len(route) - 1
        else:
            index_vi = route.index(vi)
            index_vj = route.index(vj)

        for stat in list_charging_stat_dep:
            temp_route = route[:index_vi+1] + [stat] + route[index_vj:]

            if self.is_tour_battery_ok(temp_route):
                list_new_routes.append(temp_route)
                list_cost.append(self.cost_route(temp_route))

        if list_new_routes == []:
            return "Not feasible"

        else:
            return list_new_routes[list_cost.index(min(list_cost))]

    def insert_stat_in_best_position(self, route):
        """


        Parameters
        ----------
```

```
        route : TYPE
            DESCRIPTION.

        Returns
        -------
        TYPE
            DESCRIPTION.

        """
        Q = self.__Q
        h = self.__h
        dist_array = self.__dist_array

        is_battery_ok = self.is_tour_battery_ok(route)

        if is_battery_ok:
            return route

        else:
            yi = Q

            for ind in range(1, len(route)):
                i = route[ind-1]
                j = route[ind]
                yi -= h * dist_array[i][j]

                if yi < 0:
                    break

            if j == 0:
                route = self.insert_stat_between(route, i, j, vj_last=True)[:]
            else:
                route = self.insert_stat_between(route, i, j)[:]

            if route == "Not feasible":
                return []
            else:
                return route

    def cost_route(self, route):
        """
```

```
        Parameters
        ----------
        route : TYPE
            DESCRIPTION.

        Returns
        -------
        cost : TYPE
            DESCRIPTION.

        """
        cost = 0
        dist_array = self.__dist_array
        for ind in range(1, len(route)):
            i = route[ind - 1]
            j = route[ind]
            cost += dist_array[i][j]
        return cost


    def all_clients_served(self, sol):

        list_cust = self.__list_cust[:]
        nbr_clients = len(list_cust)

        visited_clients = []

        for route in sol:
            for node in sol[route]:
                if node != 0 and node not in visited_clients and node in
    list_cust:
                    visited_clients.append(node)

        return len(visited_clients) == nbr_clients

    def number_non_served_clients(self, sol):

        list_cust = self.__list_cust[:]
        nbr_clients = len(list_cust)
```

```
        visited_clients = []

        for route in sol:
            for node in sol[route]:
                if node != 0 and node not in visited_clients and node in
    list_cust:
                    visited_clients.append(node)

        return len(visited_clients) - nbr_clients


    def spot_route_with_battery_prob(self, sol):

        list_prob = []

        for ind_route in sol:
            if not self.is_tour_battery_ok(sol[ind_route]):
                list_prob.append(ind_route)

        return list_prob

    def create_candidates_charging_station(self, route):
        list_candidates_bat = []

        for ind in range(1, len(route)):
            for stat in self.__list_stat:
                list_candidates_bat.append(route[:ind] + [stat] + route[ind:])

        return list_candidates_bat

    def adding_mirrored_candidates(self, list_candidates_bat):
        list_candidates_bat_full = list_candidates_bat[:]

        for candidates in list_candidates_bat:
            list_candidates_bat_full.append(list(reversed(candidates)))

        return list_candidates_bat_full

    def best_battery_candidate(self, list_candidates_bat):
```

```
        list_cost = []
        for candidate in list_candidates_bat:
            list_cost.append(self.cost_route(candidate))

        ind_min_cost = list_cost.index(min(list_cost))

        return list_candidates_bat[ind_min_cost]

    def fixing_cvrp_sol(self, sol):

        list_prob = self.spot_route_with_battery_prob(sol)

        for ind_route in list_prob:
            list_candidates_bat =
    self.create_candidates_charging_station(sol[ind_route])
            list_candidates_bat =
    self.adding_mirrored_candidates(list_candidates_bat)
            list_candidates_bat =
    self.remove_battery_not_ok(list_candidates_bat)

            if list_candidates_bat == []:
                return sol
            else:
                sol[ind_route] =
    self.best_battery_candidate(list_candidates_bat)

        return sol

    def cost(self, sol):

        S = 0
        dist_array = self.__dist_array

        for route_ind in sol:

            s = 0

            for ind in range (1, len(sol[route_ind])):
                i = sol[route_ind][ind - 1]
                j = sol[route_ind][ind]
```

```
                s += dist_array[i][j]

            S += s
            s = 0

        return S

    def feasibility(self, sol):

        battery_violation = 0
        capacity_violation = 0

        for route_ind in sol:
            battery_violation += self.battery_violation(sol[route_ind])
            capacity_violation += self.capacity_violation(sol[route_ind])

        unique_client_violation = self.unicity_of_client_violation(sol)

        coeff_battery = 1
        coeff_capacity = 1
        coeff_unique_client = 10

        return coeff_battery * abs(battery_violation) + coeff_capacity *
    abs(capacity_violation) + coeff_unique_client * unique_client_violation

    def unicity_of_client_violation(self, sol):

        list_sol = []

        for route_ind in sol:
            list_sol += sol[route_ind][1:-1]

        array_sol = np.array(list_sol)
        array_client = array_sol[array_sol <= max(self.__list_cust)]

        set_client = set(array_client.tolist())

        return len(array_client) - len(set_client)

### Defining functions ####
```

```
def euclidian_distance(t1, t2):
    """
    A function that takes two tuples representing the 2D coordinates of two
    nodes
    and returns the euclidian distance as float.

    Parameters
    ----------
    t1 : tuple
        Tuple of dimension two representing the coordinate of the point t1.
    t2 : tuple
        Tuple of dimension two representing the coordinate of the point t2.

    Returns
    -------
    float
        Euclidian distance between the two inputed points.

    """

    x1, y1 = t1[0], t1[1]
    x2, y2 = t2[0], t2[1]
    return m.sqrt((x1 - x2)**2 + (y1 - y2)**2)

def create_distance_matrix(dict_coord):
    """
    A function that takes the dictionnary of tuples representing 2D
    coordinates
    of the EVRP nodes. It returns an array of distances between each nodes.

    Parameters
    ----------
    dict_coord : dict
        Dictionnary with index of nodes as key and tuples of coordinates as
    value.

    Returns
    -------
    numpy.ndarray
        Multidimensional list of distance, calling [i][j] for the distance
    between and i and j.
```

```
    """

    max_ind_nodes = max(dict_coord)+1
    L, l = [], []

    for i in range(max_ind_nodes):
        for j in range(max_ind_nodes):
            l.append(euclidian_distance(dict_coord[i], dict_coord[j]))
        L.append(l)
        l = []

    return np.array(L)

def open_sol_pickle(path = "C://Users//Blaise//Business Analytics//Term
    3//Solving Electrical Vehicles Routing
    Problem//Solving-EVRP-MSCI-Business-Analytics-//Pickle solutions",
                    name = "E-n22-k4",
                    type_sol = "evrp_cvrp"):

    # Create and read instance
    inst = parse_instance(instance_name = name + ".evrp")

    # Set sol, set instance values

    coord_cust_x = inst.get_coord_cust_x()
    coord_cust_y = inst.get_coord_cust_y()
    coord_stat_x = inst.get_coord_stat_x()
    coord_stat_y = inst.get_coord_stat_y()
    dict_coord = inst.get_dict_coord()
    instance_name = inst.get_instance_name()

    dict_coord = inst.get_dict_coord()
    dict_demand = inst.get_dict_demand()
    dist_array = inst.get_dist_array()
    list_cust = inst.get_list_cust()
    list_stat = inst.get_list_stat()
    Q = inst.get_Q()
    C = inst.get_C()
    h = inst.get_h()
```

```
    heur = heuristic(dict_coord, dict_demand, dist_array, list_cust,
    list_stat, Q, C, h)

    complete_path = path + "//" + name + type_sol + ".pickle"
    pickle_in = open(complete_path, "rb")
    dict_pickle = pickle.load(pickle_in)

    sol = dict_pickle["sol"]
    cost = dict_pickle["cost"]

    return sol, cost

def plot_routes(sol,
                coord_cust_x,
                coord_cust_y,
                coord_stat_x,
                coord_stat_y,
                dict_coord,
                instance_name,
                show_labels=True,
                show_index=True):

    plt.plot(coord_cust_x,
             coord_cust_y,
             "^",
             label="Customers")

    plt.plot(coord_stat_x,
             coord_stat_y,
             "*",
             label="Charging Stations")

    plt.plot(dict_coord[0][0], dict_coord[0][1], "8", label="Depot")

    if show_index:
        for ind in dict_coord:
            plt.annotate(ind,
                         dict_coord[ind],
                         textcoords = "offset points",
                         xytext = (0,5),
                         ha = "center")
```

```
    for ind, route_ind in enumerate(sol):
        path_x, path_y = [], []
        for nodes in sol[route_ind]:
            path_x.append(dict_coord[nodes][0])
            path_y.append(dict_coord[nodes][1])

        plt.plot(path_x, path_y, label="Path " + str(ind))

    if show_labels:
        plt.legend(bbox_to_anchor=(1.05, 1),
                   loc="upper left",
                   ncol=1,
                   labelspacing=0.05,
                   borderaxespad=0.)

    plt.xlabel("x coordinate")
    plt.ylabel("y coordinate")
    plt.title(instance_name)

    plt.show()

def is_route_not_empty(route):
    if route == []:
        return False
    else:
        return True

def all_permutation_clients(List_client):

    list_permutation_clients = []

    permutations_clients = permutations(List_client)

    for el in permutations_clients:
        list_permutation_clients.append(list(el))

    return list_permutation_clients

def solve_cvrp(dist_array, list_cust, dict_demand, C):
```

```
dist_array_client = dist_array[:len(list_cust)+1 , :len(list_cust)+1]

last_col = []

for el in dist_array_client[:,0]:
    last_col.append([el])

last_col = np.array(last_col, dtype=np.float64)
last_col.T
DISTANCES = np.append(dist_array_client, last_col, axis = 1)

last_row = [[0]*(len(DISTANCES)+1)]
last_row = np.array(last_row)
DISTANCES = np.append(DISTANCES, last_row, axis = 0)
DISTANCES[:,0] = 0


# Demands (key: node, value: amount)
DEMAND = dict_demand
del DEMAND[0]

# The matrix is transformed into a DiGraph
A = np.array(DISTANCES, dtype=[("cost", float)])
G = from_numpy_matrix(A, create_using=DiGraph())

# The demands are stored as node attributes
set_node_attributes(G, values=DEMAND, name="demand")


# The depot is relabeled as Source and Sink
# The depot is relabeled as Source and Sink
G = relabel_nodes(G, {0: "Source", len(DISTANCES)-1: "Sink"})

prob = VehicleRoutingProblem(G, load_capacity=int(C))
temp_min = 1
prob.solve(time_limit = 60*temp_min)
#prob.solve(heuristic_only = True)
#prob.solve()

return prob.best_value, prob.best_routes, prob.best_routes_load
```

```
def get_number_heuristic():
    return 14

def apply_heuristic(h, sol):

    if h == 0:
        return swap_visits_inside_route(sol)
    elif h == 1:
        return swap_visits_between_routes(sol)
    elif h == 2:
        return moving_one_visit(sol)
    elif h == 3:
        return inserting_to_another_route(sol)
    elif h == 4:
        return inserting_to_a_new_route(sol)
    elif h == 5:
        return adding_station_clever(sol)
    elif h == 6:
        return adding_station_random(sol)
    elif h == 7:
        return removing_station(sol)
    elif h == 8:
        return reversing(sol)
    elif h == 9:
        return reverse_inserting(sol)
    elif h == 10:
        return swaping_block(sol)
    elif h == 11:
        return ruining_and_recreat(sol)
    elif h == 12:
        return inserting_block(sol)
    elif h == 13:
        return reverse_swaping(sol)


def swap_visits_inside_route(sol):
    """


    Parameters
    ----------
```

```
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    sol : TYPE
        DESCRIPTION.

    """

    rand_ind = rd.randint(0, len(sol)-1)
    rand_route = sol[rand_ind]

    if len(rand_route) > 3:
        x = rd.randint(1, len(rand_route) - 2)
        y = rd.randint(1, len(rand_route) - 3)

        if x == y:
            y = len(rand_route) - 2

        rand_route[x], rand_route[y] = rand_route[y], rand_route[x]

    return sol

def swap_visits_between_routes(sol):
    """
    Swap two visits in two different routes

    Parameters
    ----------
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    sol : TYPE
        DESCRIPTION.

    """

    if len(sol) < 2:
```

```
        return sol

    rand_ind_1 = rd.randint(0, len(sol)-1)
    rand_ind_2 = rd.randint(0, len(sol)-2)

    if rand_ind_1 == rand_ind_2:
        rand_ind_2 = len(sol) - 1

    rand_route_1 = sol[rand_ind_1]
    rand_route_2 = sol[rand_ind_2]

    if len(rand_route_1) >= 3 and len(rand_route_2) >= 3:
        x = rd.randint(1, len(rand_route_1) - 2)
        y = rd.randint(1, len(rand_route_2) - 2)

        rand_route_1[x], rand_route_2[y] = rand_route_2[y], rand_route_1[x]

    return sol

def moving_one_visit(sol):
    """
    Moves one value visit to the same route

    Parameters
    ----------
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    sol : TYPE
        DESCRIPTION.

    """

    rand_ind = rd.randint(0, len(sol)-1)
    rand_route = sol[rand_ind]

    if len(rand_route) > 3:

        loc = rd.randint(1, len(rand_route) - 2)
```

```
            value_loc = rand_route[loc]
            rand_route.__delitem__(loc)
            new_loc = rd.randint(1, len(rand_route) - 2)
            rand_route.insert(new_loc, value_loc)

    return sol

def inserting_to_another_route(sol):
    """
    Moves one visit from one route to another

    Parameters
    ----------
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    TYPE
        DESCRIPTION.

    """
    if len(sol) < 2:
        return sol

    rand_ind_1 = rd.randint(0, len(sol)-1)
    rand_ind_2 = rd.randint(0, len(sol)-2)

    if rand_ind_1 == rand_ind_2:
        rand_ind_2 = len(sol) - 1

    rand_route_1 = sol[rand_ind_1]
    rand_route_2 = sol[rand_ind_2]

    if len(rand_route_1) >= 3 and len(rand_route_2) >= 3:

        taking_loc = rd.randint(1, len(rand_route_1) - 2)
        taking_value = rand_route_1[taking_loc]
        inserting_loc = rd.randint(1, len(rand_route_2) - 2)

        if len(rand_route_1) == 3:
```

```
            rand_route_2.insert(inserting_loc, taking_value)
            del sol[rand_ind_1]
            updated_index_sol = {}
            for ind, route_ind in enumerate(sol):
                updated_index_sol[ind] = sol[route_ind]
            return updated_index_sol

        else:
            rand_route_1.remove(taking_value)
            rand_route_2.insert(inserting_loc, taking_value)

    return sol

def inserting_to_a_new_route(sol):
    """
    Move a visit from an existing route to a new one

    Parameters
    ----------
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    sol : TYPE
        DESCRIPTION.

    """

    rand_ind = rd.randint(0, len(sol)-1)
    nbr_ev = inst.get_nbr_ev()

    rand_route = sol[rand_ind]
    taking_loc = rd.randint(1, len(rand_route) - 2)
    taking_value = rand_route[taking_loc]

    for i in range(nbr_ev):
        if len(rand_route) > 3:
            sol[len(sol)] = [0] + [taking_value] + [0]
            return sol
```

```
    return sol

def adding_station_clever(sol):
    """
    Adding the best charging station in a random route at a random location

    Parameters
    ----------
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    sol : TYPE
        DESCRIPTION.

    """

    rand_ind = rd.randint(0, len(sol)-1)
    rand_route = sol[rand_ind]

    loc = rd.randint(1, len(rand_route) - 2)
    value_before = rand_route[loc-1]
    value_after = rand_route[loc]

    cost_insertion_list = []

    for stat in list_stat:

    cost_insertion_list.append(heur.cost_route([value_before]+[stat]+[value_after]))

    chosen_stat =
    list_stat[cost_insertion_list.index(min(cost_insertion_list))]

    rand_route.insert(loc, chosen_stat)

    return sol

def adding_station_random(sol):
    """
    Adding a random charging station in a random route at a random position
```

```
    Parameters
    ----------
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    sol : TYPE
        DESCRIPTION.

    """

    rand_ind = rd.randint(0, len(sol)-1)
    rand_route = sol[rand_ind]

    loc = rd.randint(1, len(rand_route) - 2)

    chosen_stat = rd.choice(list_stat)

    rand_route.insert(loc, chosen_stat)

    return sol

def removing_station(sol):
    """
    Removing a random charging station from a random route

    Parameters
    ----------
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    TYPE
        DESCRIPTION.

    """

    nbr_ev = inst.get_nbr_ev()
```

```
    rand_ind = rd.randint(0, len(sol)-1)
    rand_route = sol[rand_ind]

    for i in range(nbr_ev):
        stations = []

        for ind, el in enumerate(rand_route):
            if el in list_stat:
                stations.append(el)

        if len(stations) != 0:
            stat_to_remove = rd.choice(stations)

            rand_route.remove(stat_to_remove)

            if len(rand_route) <= 2:
                del sol[rand_ind]
                updated_index_sol = {}
                for ind, route_ind in enumerate(sol):
                    updated_index_sol[ind] = sol[route_ind]
                return updated_index_sol

            return sol

    return sol

def reversing(sol):
    """
    Reversing a random inside part of a route

    Parameters
    ----------
    sol : TYPE
        DESCRIPTION.

    Returns
    -------
    sol : TYPE
        DESCRIPTION.

    """
```

```
    rand_ind = rd.randint(0, len(sol)-1)
    rand_route = sol[rand_ind]

    if len(rand_route) > 3:
        x = rd.randint(1, len(rand_route) - 2)
        y = rd.randint(1, len(rand_route) - 3)

        if x == y:
            y = len(rand_route) - 2

        if x < y:
            sol[rand_ind] = rand_route[:x] + rand_route[y:x-1:-1] +
    rand_route[y+1:]
        else:
            x, y = y, x
            sol[rand_ind] = rand_route[:x] + rand_route[y:x-1:-1] +
    rand_route[y+1:]

    return sol

def reverse_inserting(sol):

    if len(sol) < 2:
        return sol

    rand_ind_1 = rd.randint(0, len(sol)-1)
    rand_ind_2 = rd.randint(0, len(sol)-2)

    if rand_ind_1 == rand_ind_2:
        rand_ind_2 = len(sol) - 1

    rand_route_1 = sol[rand_ind_1]
    rand_route_2 = sol[rand_ind_2]

    if len(rand_route_1) > 3:
        x = rd.randint(1, len(rand_route_1) - 2)
        y = rd.randint(1, len(rand_route_1) - 3)

        if x == y:
            y = len(rand_route_1) - 2
```

```
            if x < y:
                temp_block = rand_route_1[y:x-1:-1]
            else:
                x, y = y, x
                temp_block = rand_route_1[y:x-1:-1]
        else:
            return sol

        if len(rand_route_1) >= 3:
            loc = rd.randint(1, len(rand_route_2) - 2)

            sol[rand_ind_2] = rand_route_2[:loc] + temp_block + rand_route_2[loc:]
            sol[rand_ind_1] = rand_route_1[:x] + rand_route_1[y+1:]

            if len(sol[rand_ind_1]) < 3:
                del sol[rand_ind_1]
                updated_index_sol = {}
                for ind, route_ind in enumerate(sol):
                    updated_index_sol[ind] = sol[route_ind]
                return updated_index_sol

        return sol

    def reverse_swaping(sol):

        if len(sol) < 2:
            return sol

        rand_ind_1 = rd.randint(0, len(sol)-1)
        rand_ind_2 = rd.randint(0, len(sol)-2)

        if rand_ind_1 == rand_ind_2:
            rand_ind_2 = len(sol) - 1

        rand_route_1 = sol[rand_ind_1]
        rand_route_2 = sol[rand_ind_2]

        if len(rand_route_1) > 3 and len(rand_route_2) > 3:

            x1 = rd.randint(1, len(rand_route_1) - 2)
```

```
    y1 = rd.randint(1, len(rand_route_1) - 3)

    if x1 == y1:
        y1 = len(rand_route_1) - 2

    x2 = rd.randint(1, len(rand_route_2) - 2)
    y2 = rd.randint(1, len(rand_route_2) - 3)

    if x2 == y2:
        y2 = len(rand_route_2) - 2


    if x1 > y1 and x2 > y2:
        x1, y1 = y1, x1
        x2, y2 = y2, x2

        temp_block_1 = rand_route_1[y1:x1-1:-1]
        temp_block_2 = rand_route_2[y2:x2-1:-1]

        sol[rand_ind_2] = rand_route_2[:x2] + temp_block_1 +
rand_route_2[y2+1:]
        sol[rand_ind_1] = rand_route_1[:x1] + temp_block_2 +
rand_route_1[y1+1:]

    elif x1 < y1 and x2 < y2:
        temp_block_1 = rand_route_1[y1:x1-1:-1]
        temp_block_2 = rand_route_2[y2:x2-1:-1]

        sol[rand_ind_2] = rand_route_2[:x2] + temp_block_1 +
rand_route_2[y2+1:]
        sol[rand_ind_1] = rand_route_1[:x1] + temp_block_2 +
rand_route_1[y1+1:]

    elif x1 < y1 and x2 > y2:
        x2, y2 = y2, x2

        temp_block_1 = rand_route_1[y1:x1-1:-1]
        temp_block_2 = rand_route_2[y2:x2-1:-1]

        sol[rand_ind_2] = rand_route_2[:x2] + temp_block_1 +
rand_route_2[y2+1:]
```

```
            sol[rand_ind_1] = rand_route_1[:x1] + temp_block_2 +
    rand_route_1[y1+1:]

        elif x1 > y1 and x2 < y2:
            x1, y1 = y1, x1

            temp_block_1 = rand_route_1[y1:x1-1:-1]
            temp_block_2 = rand_route_2[y2:x2-1:-1]

            sol[rand_ind_2] = rand_route_2[:x2] + temp_block_1 +
    rand_route_2[y2+1:]
            sol[rand_ind_1] = rand_route_1[:x1] + temp_block_2 +
    rand_route_1[y1+1:]

    return sol

def swaping_block(sol):

    if len(sol) < 2:
        return sol

    rand_ind_1 = rd.randint(0, len(sol)-1)
    rand_ind_2 = rd.randint(0, len(sol)-2)

    if rand_ind_1 == rand_ind_2:
        rand_ind_2 = len(sol) - 1

    rand_route_1 = sol[rand_ind_1]
    rand_route_2 = sol[rand_ind_2]

    if len(rand_route_1) > 3 and len(rand_route_2) > 3:

        x1 = rd.randint(1, len(rand_route_1) - 2)
        y1 = rd.randint(1, len(rand_route_1) - 3)

        if x1 == y1:
            y1 = len(rand_route_1) - 2

        x2 = rd.randint(1, len(rand_route_2) - 2)
        y2 = rd.randint(1, len(rand_route_2) - 3)
```

```
        if x2 == y2:
            y2 = len(rand_route_2) - 2

        if x1 > y1 and x2 > y2:
            x1, y1 = y1, x1
            x2, y2 = y2, x2
            sol[rand_ind_1], sol[rand_ind_2] = rand_route_1[:x1] +
    rand_route_2[x2:y2] + rand_route_1[y1:], rand_route_2[:x2] +
    rand_route_1[x1:y1] + rand_route_2[y2:]

        elif x1 < y1 and x2 < y2:
            sol[rand_ind_1], sol[rand_ind_2] = rand_route_1[:x1] +
    rand_route_2[x2:y2] + rand_route_1[y1:], rand_route_2[:x2] +
    rand_route_1[x1:y1] + rand_route_2[y2:]

        elif x1 < y1 and x2 > y2:
            x2, y2 = y2, x2
            sol[rand_ind_1], sol[rand_ind_2] = rand_route_1[:x1] +
    rand_route_2[x2:y2] + rand_route_1[y1:], rand_route_2[:x2] +
    rand_route_1[x1:y1] + rand_route_2[y2:]

        elif x1 > y1 and x2 < y2:
            x1, y1 = y1, x1
            sol[rand_ind_1], sol[rand_ind_2] = rand_route_1[:x1] +
    rand_route_2[x2:y2] + rand_route_1[y1:], rand_route_2[:x2] +
    rand_route_1[x1:y1] + rand_route_2[y2:]

    return sol

def inserting_block(sol):

    if len(sol) < 2:
        return sol

    rand_ind_1 = rd.randint(0, len(sol)-1)
    rand_ind_2 = rd.randint(0, len(sol)-2)

    if rand_ind_1 == rand_ind_2:
        rand_ind_2 = len(sol) - 1

    rand_route_1 = sol[rand_ind_1]
```

```
    rand_route_2 = sol[rand_ind_2]

    if len(rand_route_1) > 3 and len(rand_route_2) > 3:

        x1 = rd.randint(1, len(rand_route_1) - 2)
        y1 = rd.randint(1, len(rand_route_1) - 3)

        if x1 == y1:
            y1 = len(rand_route_1) - 2

        if x1 < y1:
            temp_block_1 = rand_route_1[x1:y1]
            loc = rd.randint(1, len(rand_route_2) - 2)
            sol[rand_ind_2] = rand_route_2[:loc] + temp_block_1 +
    rand_route_2[loc:]
            sol[rand_ind_1] = rand_route_1[:x1] + rand_route_1[y1:]

            if len(sol[rand_ind_1]) < 3:

                del sol[rand_ind_1]
                updated_index_sol = {}
                for ind, route_ind in enumerate(sol):
                    updated_index_sol[ind] = sol[route_ind]
                return updated_index_sol
        else:
            x1, y1 = y1, x1

            temp_block_1 = rand_route_1[x1:y1]
            loc = rd.randint(1, len(rand_route_2) - 2)
            sol[rand_ind_2] = rand_route_2[:loc] + temp_block_1 +
    rand_route_2[loc:]
            sol[rand_ind_1] = rand_route_1[:x1] + rand_route_1[y1:]

            if len(sol[rand_ind_1]) < 3:

                del sol[rand_ind_1]
                updated_index_sol = {}
                for ind, route_ind in enumerate(sol):
                    updated_index_sol[ind] = sol[route_ind]
                return updated_index_sol
```

```
    return sol


def ruining_and_recreat(sol):

    rand_ind = rd.randint(0, len(sol)-1)
    rand_route = sol[rand_ind]

    if len(rand_route) <= 3:
        return sol
    else:
        arr = np.array(rand_route[1:-1])
        np.random.shuffle(arr)
        sol[rand_ind] = [0] + arr.tolist() + [0]

    return sol

### More operators only swapping clients ###

iterations_x_axis_1 = np.linspace(0, 10000, 11)
iterations_x_axis_2 = np.linspace(10000, 100000, 10)
iterations_x_axis = np.array(list(iterations_x_axis_1[:-1]) +
    list(iterations_x_axis_2))

def heuristic_method(sol, threshold_replay, percentage_best_cost):

    cost_over_iterations = np.array([])
    already_stored_index = []

    cost = heur.cost(sol) + 100000 * heur.feasibility(sol)
    new_sol = copy.deepcopy(sol)

    best_sol = copy.deepcopy(sol)
    best_cost = cost

    uti = [0]*get_number_heuristic()
    llh = list(range(0,get_number_heuristic()))

    i = 0
    ind_last_best = 0
```

```
max_iter = 100000

st = time.time()

while i <= 100000:

    if i - ind_last_best >= 0.05 * max_iter:
          llh = list(range(0,get_number_heuristic()))

    if i%1000 == 0:
        print(i,best_cost)

    if i in iterations_x_axis and i not in already_stored_index:
        already_stored_index.append(i)
        print("stock",i,cost)
        cost_over_iterations = np.append(cost_over_iterations, cost)

    h = llh[rd.randint(0, len(llh)-1)]
    sol = apply_heuristic(h, sol)

    if rd.randint(0, threshold_replay) == 0:
        continue

    cost_new = heur.cost(sol) + 100000 * heur.feasibility(sol)
    i += 1

    if cost_new < best_cost:
        ind_last_best = i
        llh.append(h)
        uti[h] += 1
        best_cost = cost_new
        best_sol = copy.deepcopy(sol)

    if cost_new <= cost or cost_new <= best_cost +
percentage_best_cost*best_cost:
        cost = cost_new
        new_sol = copy.deepcopy(sol)

    else:
        sol = copy.deepcopy(new_sol)
```

```
    et = time.time()
    elapsed_time = et - st

    list_operators = ["swap_visits_inside_route",
                      "swap_visits_between_routes",
                      "moving_one_visit",
                      "inserting_to_another_route",
                      "inserting_to_a_new_route",
                      "adding_station_clever",
                      "adding_station_random",
                      "removing_station",
                      "reversing",
                      "reverse_inserting",
                      "swaping_block",
                      "ruining_and_recreat",
                      "inserting_block",
                      "reverse_swaping"]

    df_uti = pd.DataFrame({"Name operator": list_operators, "Utility value":
    uti})

    print(df_uti)
    return best_sol, df_uti, cost_over_iterations, elapsed_time

### Main ###

name_file = "E-n22-k4"

inst = parse_instance(instance_name = name_file + ".evrp")

coord_cust_x = inst.get_coord_cust_x()
coord_cust_y = inst.get_coord_cust_y()
coord_stat_x = inst.get_coord_stat_x()
coord_stat_y = inst.get_coord_stat_y()
dict_coord = inst.get_dict_coord()
instance_name = inst.get_instance_name()

dict_coord = inst.get_dict_coord()
dict_demand = inst.get_dict_demand()
dist_array = inst.get_dist_array()
list_cust = inst.get_list_cust()
```

```
list_stat = inst.get_list_stat()
Q = inst.get_Q()
C = inst.get_C()
h = inst.get_h()
n = inst.get_Dim()


heur = heuristic(dict_coord, dict_demand, dist_array, list_cust, list_stat, Q,
    C, h)

sol_greedy = open_sol_pickle(name = name_file, type_sol = "_greedy")[0]

sol = copy.deepcopy(sol_greedy)

sol, df_uti, cost_over_iterations, elapsed_time = heuristic_method(sol, 10,
    0.01)

print("5", sol_greedy)
print(heur.cost(sol))
print(heur.feasibility(sol))
```

Bibliography

[1] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[2] Geoff Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

[3] Gilbert Laporte, Martin Desrochers, and Yves Nobert. Two exact algorithms for the distance-constrained vehicle routing problem. *Networks*, 14(1):161–172, 1984.

[4] Ricardo Fukasawa, Humberto Longo, Jens Lysgaard, Marcus Poggi de Aragão, Marcelo Reis, Eduardo Uchoa, and Renato F Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical programming*, 106(3):491–511, 2006.

[5] Roberto Baldacci, Nicos Christofides, and Aristide Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, 115(2):351–385, 2008.

[6] José Brandão. A tabu search algorithm for the open vehicle routing problem. *European Journal of Operational Research*, 157(3):552–564, 2004.

[7] Raúl Baños, Julio Ortega, Consolación Gil, Antonio Fernández, and Francisco De Toro. A simulated annealing-based parallel multi-objective approach to vehicle routing problems with time windows. *Expert Systems with Applications*, 40 (5):1696–1707, 2013.

[8] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, and Christian Prins. A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows. *Computers & operations research*, 40 (1):475–489, 2013.

[9] Ilker Kucukoglu, Reginald Dewil, and Dirk Cattrysse. The electric vehicle routing problem and its variations: A literature review. *Computers & Industrial Engineering*, 161:107650, 2021.

[10] Dominik Goeke and Michael Schneider. Routing a mixed fleet of electric and conventional vehicles. *European Journal of Operational Research*, 245(1):81–99, 2015.

[11] Alejandro Montoya, Christelle Guéret, Jorge E Mendoza, and Juan Villegas. *The electric vehicle routing problem with partial charging and nonlinear charging function.* PhD thesis, LARIS, 2015.

[12] Michalis Mavrovouniotis, Charalambos Menelaou, Stelios Timotheou, Christos Panayiotou, Georgios Ellinas, and Marios Polycarpou. Benchmark set for the ieee wcci-2020 competition on evolutionary computation for the electric vehicle routing problem. *KIOS COE*, 2020.

[13] Ahmed Alridha, Abbas Musleh Salman, and Ahmed Sabah Al-Jilawi. The applications of np-hardness optimizations problem. In *Journal of Physics: Conference Series*, volume 1818, page 012179. IOP Publishing, 2021.

[14] Ed Pike. Calculating electric drive vehicle greenhouse gas emissions. *International Council on Clean Transportaion: Washington, DC, USA*, 2012.

[15] Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017.

[16] Michalis Mavrovouniotis, Charalambos Menelaou, Stelios Timotheou, Georgios Ellinas, Christos Panayiotou, and Marios Polycarpou. A benchmark test suite for the electric capacitated vehicle routing problem. 07 2020. doi: 10.1109/CEC48606.2020.9185753.

[17] Mauro Dell'Amico, Giovanni Righini, and Matteo Salani. A branch-and-price approach to the vehicle routing problem with simultaneous distribution and collection. *Transportation science*, 40(2):235–247, 2006.

[18] Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck. A generic exact solver for vehicle routing and related problems. *Mathematical Programming*, 183(1):483–523, 2020.

[19] Elham Shadkam. Parameter setting of meta-heuristic algorithms: a new hybrid method based on dea and rsm. *Environmental Science and Pollution Research*, 29 (15):22404–22426, 2022.

[20] Shuai Zhang, Yuvraj Gajpal, SS Appadoo, and MMS Abdulkader. Electric vehicle routing problem with recharging stations for minimizing energy consumption. *International Journal of Production Economics*, 203:404–413, 2018.

[21] Yazid Sebsadji, Sébastien Glaser, Said Mammar, and Jamil Dakhlallah. Road slope and vehicle dynamics estimation. In *2008 American control conference*, pages 4603–4608. IEEE, 2008.