A vehicle scheduling tool for electric buses based on column generation

Master thesis

Master:	Manufacturing system engineering
Department:	Mechanical engineering
Research group:	Dynamics & Control
DC number:	DC 2021.022

Student:	X.H.Ouwerkerk
Identity number:	0912559
Thesis supervisors:	Dr.ir. A.A.J. Lefeber (TU/e) Prof.dr. H. Nijmeijer (TU/e) ir. S.J.A. Rutten (VDL ETS)
Date:	7th March 2021

Summary

The goal of this project is to create a tool which creates schedules for electric buses from timetables that can be used in practice. The objective is to minimize the number of buses and the number of fast chargers needed to create a schedule. The problem is a mixed-integer linear programming (MILP) problem and is solved with a column generation (CG) model. The columns in the CG model represent vehicle tasks, which describe a day route for a bus. The subproblem is a large MILP problem, which has to find new vehicle tasks with negative reduced costs. One of the main problems is the long computational time of solving the subproblem. In this thesis multiple methods are investigated to reduce the computational time of the subproblem. A greedy algorithm, a genetic algorithm and a diving heuristic are implemented. All three heuristics do not perform well enough. A different approach has also been investigated. The idea of this approach is to create a multiple step CG model, where the subproblem is decomposed into multiple subproblems. This attempt has been unsuccessful. A promising method is that of writing the subproblem as a shortest path problem (SPP) with the help of a graph. The graph consists of trip nodes and charger nodes. The arcs from and to trip nodes are deadhead trips, while the arcs between charger nodes are charging sessions. A shortest path is found with a label-correcting algorithm, which takes into account the reduced costs, the SoC, the minimum charge time and the minimum shift time. The label-correcting algorithm is based on the Bellman-Ford algorithm. This latter method is implemented in the CG model. The fractional solution of the optimal solution of the Restricted Master problem (RMP) in the CG model is rounded up with a diving heuristic, which has been chosen because it is in general relatively fast and gives relatively accurate solutions.

Timetables of different cities are simulated. An assumption is made that the energy consumption is 1.5 kWh/km. The results of the new model look promising and are compared to a lower bound for the number of buses needed. For the cities of Le Havre and Bordeaux the lower bound for the number of buses is reached. For Eindhoven the lower bound for the number of buses is not reached, but the schedule uses fewer buses than the current used schedule. The model can create schedules within reasonable time for timetables with the size of the timetable of Rotterdam. The CG model has three main disadvantages: The used charge curve is linear instead of non-linear, trips can be assigned in the schedule multiple times and it is not possible to restrict the number of chargers at a location.

Preface

The completion of my graduation project concludes the end of my master at the TU/e. This graduation project has given me an insight in a new field for me: The field of optimization. I have enjoyed it a lot to learn about optimization techniques. I would like to take this opportunity to show my gratitude to the people that helped me during my graduation project.

First and foremost, I want to thank Erjen Lefeber for his guidance, for answering all my questions during this project and for the weekly meetings which helped me a lot throughout this project. Furthermore, I would like to thank Henk Nijmeijer for his advice and guidance during our monthly meetings.

I would like to thank VDL and Stijn Rutten for giving me an opportunity to do my graduation project at VDL ETS. I want to thank Stijn Rutten for the many discussions during our weekly meetings that gave me new insights and for his guidance during this project.

Lastly, I want to thank my family for their support in these weird times.

Xander

Contents

Nomenclature

Abbreviations

CG	Column generation
CS	Concurrent scheduler
$ERC - SPP$	Elementary resource constrained shortest path problem
$eVSP$	Electric vehicle scheduling problem
ILP	Integer linear programming
LP	Linear programming
$MILP$	Mixed-integer linear programming
MP	Master problem
RMP	Restricted master problem
SP	Subproblem
SSP	Subsubproblem
$SSSP$	Subsubsubproblem

Sets

A_{list}	Set of all arcs in the graph ordered on earliest start time
C_l	Set of all charging locations
L	Set of all labels in a node
P_{C_l}	Set of all charging sessions at a charger location
T	Set of all trips
T_{final}	Set of all trips in the final solution
V	Set of vehicle tasks
V_{final}	Set of vehicle tasks in the final solution
Z	Set of timeblocks

Decision variables

δ_t	Determines if trip t is added to the solution in the subproblem
ϵ_z	Amount of energy charged during a timeblock [kWh]
σ_z	If a charger is assigned to a vehicle task during a timeblock
$n_c l$	Determines the number of chargers used at charger location cl
r_{pcl}	Determines the percentage used of a charging session in the subproblem
u_v	Determines if vehicle task v is chosen

Dual variables

π_t	Dual variable of the constraint for trip t
π_{pcl}	Dual variable of the constraint for number of chargers used for charging session p_{cl} at charging location cl
ρ	Dual variable for energy charged each timeblock
θ_ζ	Dual variable for the number of chargers used each timeblock

Variables

a_{tv}	Arcs in vehicle task v that go to trip t
b	Number of buses used
$e_{tot}(t)$	Total energy available in the schedule at each time point [kWh]

e_{zv}	Variable that states how much energy is charged in timeblock z in vehicle task v
$n_{p_{cl}final}$	Number of charger used in the final solution at charging session p_{cl}
$n_{p_{cl}}$	Number of chargers used during charging sessions p at charger location cl
s_{zv}	Variable that states if a charger is used in timeblock z in vehicle task v
v	A vehicle task
x_{tv}	Variable that states if trip t is assigned to vehicle task v

Parameters

ϵ_{max}	Maximum energy charged at a timeblock [kWh]
c_{cl}	Costs for a charger at charger location cl
c_v	Costs for vehicle task v
cl	Charger location cl
dt	Time interval between two charger nodes [min]
e_h	Energy costs of a deadhead trip
e_{charge}	Charge rate of a charger [% of the SoC per minute]
e_{max}	Maximum amount of energy in the battery [kWh]
e_{min}	Minimum amount of energy in the battery [kWh]
e_t	Energy costs of a trip
ht_{end}	End time of trip t [min]
ht_{start}	Start time of trip t [min]
l	A label at a node
n_s	Number of chargers allowed at a charger location
n_t	Number of trips in a schedule
p_{cl}	Charging session at charger location cl
SoC_{max}	Maximum State of Charge [% of total SoC]
SoC_{min}	Minimum State of Charge [% of total SoC]
t	A trip
z	A timeblock

Appendix: Sets

D	A set of shifts
L_σ	List of time block in which a charging is used
L_{ci}	A set of charging intervals
M_{pcl}	A set of trip nodes that have an outgoing arc to charging location pcl
N	A set of nodes
N_{pcl}	A set of charger nodes
Q_t	A set of nodes that have an outgoing arc towards trip t
S	A set of charger tasks
V_{new}	A set of newly created vehicle tasks

Appendix: Variables & Parameters

δ_c	Dual variable for each l_{ci}
c_d	Costs for a shift
c_s	Costs for a charger task
d	A shift
$e_{current}$	Amount of energy at current point in time [kWh]
g	Number of generations
l_{ci}	Charging interval
m_{pcl}	A trip node that has an outgoing arc to charging location pcl
ps	Number of participans
s	A charger task
SoC_N	State of charge at node N [%]
ts	Number of tournaments

u_d	Decision variable wheter a shift is used or not
u_s	Decision variable wheter a charging task is used or not
$u_{l_{ci}}$	Decision variable wheter a charging interval is used or not
v_{new}	A new vehicle task
z_t	A node that have an outgoing arc towards trip t

Chapter 1

Introduction

In recent years the transition from the use of fossil fuels to other alternative energy sources has been a global trend. The goal of the EU is to achieve that 27% of all energy sources comes from renewable energy sources by 2030 [1]. This can be partly achieved by making use of electric buses in the public transport sector. Use of electric buses lead to an increase in use of renewable energy resources and to a decrease in CO2 emission. Eight percent of the pollution in cities is from city buses [1]. Most buses in the EU still use fossil fuel, but a shifting trend to the use of electric buses is visible. In more and more cities buses equipped with drivetrains that contain combustion engines are replaced with buses that have alternative engines. In 2017 eighty percent of the buses used diesel engines, but it is expected that 52% of all buses use electric engines in 2030 [1].

The bus manufacturer VDL Bus & Coach is one of the front-runners in this new trend. In February 2020 670 electric buses of VDL Bus & Coach have been operational. However, there are still a lot of challenges left in the transition from diesel buses to electric buses. One of the major challenges is the short driving range of electric buses, which means that the buses need to be recharged during the day. This leads to several problems.

One of these problems is creating a schedule for electric vehicle buses. Currently, most schedules are made for diesel buses and have to be rewritten for electric buses. In the new schedule all the trips from a given timetable have to be assigned to electric buses. Each electric bus is assigned a set of trips, which the bus has to drive during the day. The schedule also has to take account for place and time for charging electric buses. A lot of elements have to be taken into account when creating a schedule for electric buses. The most important elements are the battery capacity of the bus, the energy consumption of the bus, the charger locations, the charger types and the number of chargers at a location.

VDL Bus & Coach often receives the question from the customer if they can provide a schedule for a given timetable of a city. The schedule has to use one of the electric city bus types from VDL Bus & Coach and needs to include how many buses and chargers are used and at which location. Currently, the employees working in the sales department create these schedules manually, which can take many hours. With an increasing size of the timetable, it becomes increasingly difficult to manually find a feasible and near optimal solution. To create a better and faster schedule VDL ETS has been working on creating a vehicle scheduling tool. The tool has to create a schedule for a given number of trips from a timetable which has time constraints. The problem that the tool has to solve is an optimization problem. This means that the tool has to find the best possible solution out of all feasible solutions. The problem can be placed in the category of an eVSP, which stands for electric vehicle scheduling problem.

VDL ETS and TU/e have started a cooperation to create this tool in order to solve the previous mentioned problem. TU/e student Monhemius has done an internship at VDL ETS. Monhemius has used a MILP (Mixed integer linear problem) solver to solve the problem [2]. The conclusion of the internship is that the use of a MILP solver leads to computational times that are too long. TU/e student Wijnheijmer has done a graduation project [3] with the goal of reducing the computational time. Wijnheijmer has made a concurrent scheduler based on the one of Adler [4], which creates a new schedule fast but often this schedule is suboptimal. Therefore, Wijnheijmer has made another model, which is a column generation (CG) model. The main problem of the column generation model of Wijnheijmer is the long

computational time. The model also misses some important features that make it possible to be used in practice. The most important ones are deadhead trips, multiple charger locations and non-linear charging. The goal of this project is to create a scheduling tool for VDL, where the CG model of Wijnheijmer is used as a starting point.

1.1 Timetable

The previous section has described that the customers give a timetable of the city as input. This section explains what a timetable looks like. The timetables contain the following information:

- A list with the start and end times of the trips
- The start and end locations of the trips
- The distance covered per trip
- The distance between locations (optional)
- The driving time from one location to another location (optional)
- The battery capacity of the bus
- The charge capacity of the charger per location
- The number of charge locations (and the number of chargers per location)

The table below shows the first part of a timetable:

From	Start Time	End Time	To	Distance [m]
Hôpital Estuaire	04:50:00	05:27:00	Graville	14295
Graville	05:00:00	05:40:00	Hôpital Estuaire	14487
Hôpital Estuaire	05:15:00	05:52:00	Graville	14295
Graville	05:33:00	06:13:00	Hôpital Estuaire	14487
Hôpital Estuaire	05:35:00	06:13:00	Graville	14295

Table 1.1: *The first part of the Timetable of Le Havre*

In general timetables contain between 200 until 4500 trips. Sometimes the distance of deadhead trips are also given in a timetable. Deadhead trips are trips from one location to another location without having passengers in a bus. An example is given below:

Distance [m]	Hôpital Estuaire	Graville	Depot_1	Fuel_1
Hôpital Estuaire	0	14295	400	14295
Graville	0	0	12900	0
Depot_1	400	12900	0	12900
Fuel_1	14295	0	12900	0

Table 1.2: *Distance of the deadhead trips of Le Havre*

Table 1.2 also gives the information on how many charging locations there are and where these are located. Depot_1 is the depot and also the location where slow chargers are available and Fuel_1 is a fast charger location. A similar table is created for the deadhead trips in which the drive time is stated. In Appendix A an overview is given on the timetables used in this project, where the most important information on the timetables is given.

1.2 Schedule

From the timetable a schedule has to be created, which has to provide insight regarding the amount of buses that are needed to drive the schedule. To each bus a part of the trips of the timetable is assigned.

Each trip in the timetable has to be assigned to a bus. The schedule also has to contain information on charging sessions meaning when, where and for how long a bus needs to charge. The cost for the bus company consists of the costs of the number of buses used and of the number of fast chargers used. These costs have to be minimized. The schedule has to be suited to be used in practice. Therefore, VDL ETS and the sales department of VDL Bus & Coach have several requirements which a tool has to take into account when creating a schedule. These requirements are listed below:

1. The schedule has to include deadhead trips, which are trips driven from one location to another location, while these trips are not listed in the timetable. For example a bus has driven a trip and needs to charge. The deadhead trip is the trip from the end location of the trip to the charger location.

2. There has to be a minimum shift time in the schedule. A shift is a series of sequential driven trips without the option of charging the bus in between driving the trips. A minimum shift time is needed, because the bus transport companies do not want the bus to leave the depot while driving only one short trip.

3. There has to be a minimum charge time in the schedule. The main reason is that in practice it is not possible to implement a bus charging for a period, which then stops charging, to make place for another bus and then starts charging again. Bus transport companies do not accept such solutions.

4. There is a one-minute charger connection time, which has to be implemented.

5. A bus cannot have a lower State of Charge (SoC) than the minimum allowable SoC for that bus type. The SoC is the percentage of charge left in the battery. A bus can be charged until a maximum SoC, which is a certain limit depending on which charger is used.

6. It must be possible to use multiple charging locations and different charger types, when these are available. There are multiple charging locations in most cities. At some locations only a limited number of fast chargers are allowed. In general there are also two different kinds of chargers used: A slow charger and a fast charger. Slow chargers are often placed at the depot, while fast chargers are often placed at charge locations closer to the city. For each bus one slow charger is available at the depot. Slow chargers do not have to be taken into account, when calculating the cost of the schedule.

7. There is a maximum idle time for the buses. In most locations it is not possible for a bus to stand still for a long period of time, since there is no parking space. The driver is also paid for waiting, if the bus is idle. It is preferable that the maximum idle time is as low as possible. The range for the maximum idle time is between a minute and ten minutes. A bus is allowed to be idle at the depot for a longer time than the maximum idle time.

8. The charge curve cannot be linear and has to follow the real slope, which is non-linear. The charge rate is based on the charge curve and the current SoC of the bus.

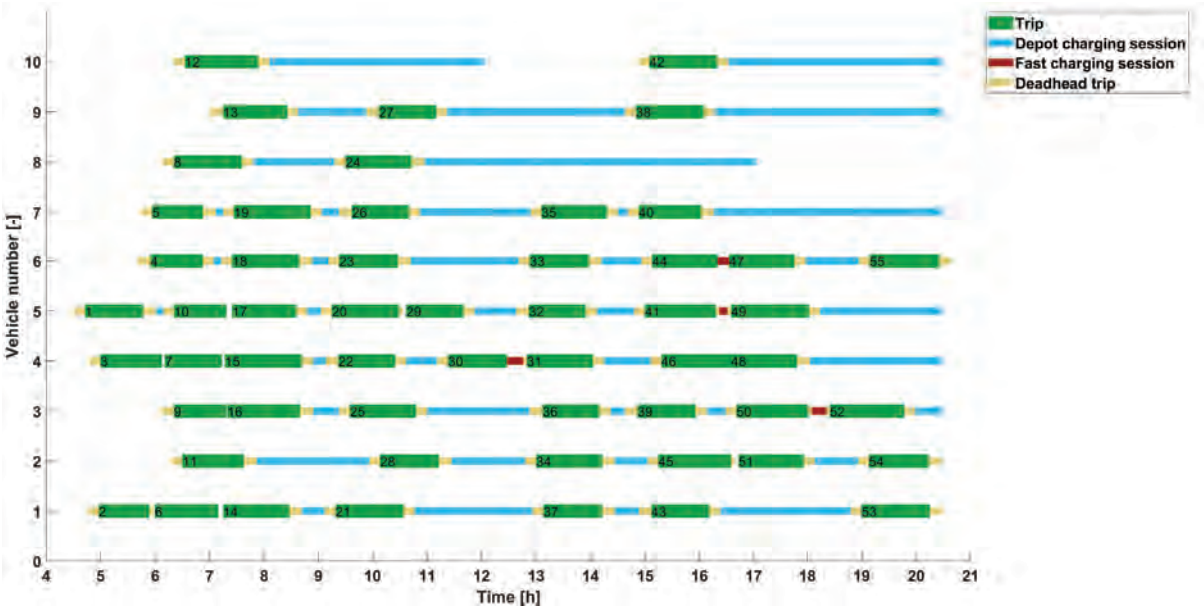


Figure 1.1: *Example of a schedule shown in a Gantt chart*

An example of a schedule can be seen in Figure 1.1. The green bars represent the trips, the yellow bars represent the deadhead trips, the blue bars represent the charging sessions at the depot with a slow charger and the red bars represent the charging sessions with a fast charger. The numbers in the green bars are corresponding with the numbers of the trips, which are implemented to increase the user-friendliness of the Gantt chart. The numbers are often left out in this thesis, since these are not relevant for explaining the model.

1.3 Problem definition

The main problem of this project is to create a scheduling tool for VDL ETS that has to make a schedule for electric buses for a given timetable. There are three problems:

1. The tool has to be able to create a schedule from a given timetable that can be implemented in practice. To be able to implement a schedule in practice the requirements in section 1.2 have to be met.

2. The tool has to minimize the cost of the schedule, which makes it an optimization problem. The cost consists of the number of buses and the number of fast chargers used in the schedule.

3. Schedules have to be created in a reasonable time, which is defined as that the schedule has to be able to be created within a day. The largest timetable contains around 4500 trips, but the goal of this project is to be able to simulate timetables with the size of the timetable for Rotterdam, which includes 1096 trips and 2 charge locations.

The starting point of this project is the column generation model of Wijnheijmer [3]. The computational time is too long for the current model of Wijnheijmer [3] and the created schedules cannot be used in practice. The schedules do not meet requirements 1,2,4,6,7 and 8 of section 1.2.

1.4 Outline

The thesis is structured as follows: In Chapter 2 a literature overview is given, then a short explanation about MILP is given and an explanation on the column generation method is given. The latter named method is the method which is used in this thesis. The results of the work of previous students Monhemius [2] and Wijnheijmer [3] are discussed in the last part of Chapter 2. In Chapter 3 first a lower bound of the problem is found in order to estimate the quality of solutions of the new models. The main part of Chapter 3 consists of explaining methods that are used to accelerate solving the subproblem.

These methods include heuristics for solving the subproblem, a multiple step column generation method and the method to rewrite the problem to a resource constrained shortest path problem. The latter named method is used in the tool. In Chapter 4 the new model is discussed, in which the implementation of the method is explained. In Chapter 5 the additional features, that are needed to be able to run the column generation model smoothly, are explained. Chapter 6 gives the results of the new model. The last chapter consists of the conclusion and of the recommendations to improve the model in the future.

Chapter 2

Background information and Literature review

In Chapter 1 the problem of this thesis has been introduced, which is the creation of a schedule for electric buses from a given timetable. In this chapter a literature review is given on creating schedules for electric vehicles. The second part of this chapter explains the basic theory on the optimization methods used in this thesis to solve large integer problems. These methods are also implemented in the work of Monhemius [2] and Wijnheijmer [3]. The results of these works are explained in the last part of this chapter.

2.1 Literature review

The problem that the tool has to solve is, according to literature, called a vehicle scheduling problem (VSP), which is a problem where a number of trips with fixed start and end times has to be assigned to (often a minimum number of) vehicles. There has been and there still is a lot of research done on VSP models. An overview of the VSP problem is given by [6]. The problem that has to be solved for VDL is an eVSP (electric vehicle scheduling problem) for electric city buses. The complexity of the eVSCP (electric vehicle scheduling and charging problem) is $\mathcal{NP}$-hard, which is proven by Sassi et al. [5]. A lot of different techniques can be used for solving the problem. Some have tried to use a MILP solver: Monhemius [2] and Pereira [7] which has tried to solve a multi-depot electric vehicle scheduling problem (MD-eVSP). The latter paper uses a MILP first and afterwards a global heuristic to solve the problem. The previous papers come to the conclusion that heuristics are needed to solve the problem. A variety of heuristics can be used. One of the options is using a Large Neighbourhood Search (LNS). Perumal et al. [8] uses this method for a single depot electric vehicle scheduling problem (SD-eVSP), in the heuristic a branch-and-price (B&P) algorithm is used to repair the solution. Chao & Xiaohong [9] use a Non-dominated Sorting Genetic Algorithm-II to solve a SD-eVSP problem. Teng et al. [10] solves a multi-objective SD-eVSP with a particle swarm optimization heuristic.

The majority of the papers use a column generation algorithm to solve an eVSP problem for electric buses. An in-depth explanation of using this method can be found in the book of Desaulniers et al. [11] and the paper of Lübbecke [12]. The papers that use column generation to solve the eVSP often differ in requirements. For example Adler [3] solves a VSP problem with buses that have alternative fuels, where the electric batteries are not charged at a station but are replaced with new batteries. The problem is solved with multiple algorithms including a concurrent scheduler and a column generation method. In the latter method the subproblem is formulated as a resource constrained shortest path problem (RC-SPP) and is solved with a labelling-correcting algorithm. Li [14] also considers using replaceable batteries. In the paper a column generation algorithm is used, where the subproblem is also structured as a RC-SPP and is solved with a labelling algorithm. Sundin [15] uses column generation for a SD-eVSP model, where it is assumed that multiple buses can charge at the same moment at the same charger. Van Kooten Niekerk [16] describes the subproblem as a RC-SPP and solves it with a labelling-correcting algorithm. Charging is modelled in discrete time and the costs of simultaneous charging of multiple buses is not taken into account. Posthoorn [17] uses the same approach as Van Kooten Niekerk, but the paper differs in finding an integer solution and it limits the number of chargers.

2.2 Introduction on solving large integer problems

The eVSP is a problem with many integer variables. The number of buses and the trips assigned to these buses are all integer variables. Integer problems can be formulated as an ILP (integer linear programming) problem. In the problem of this thesis continuous variables are also used, namely the duration of a charging session. The optimization problem which includes integer variables and continuous variables is called a MILP (mixed integer linear programming) problem. A MILP solver is often used to solve the MILP problem, for example the intlinprog solver of MATLAB can be used. The MILP problem and the use of a MILP solver is explained in the first subsection. In many cases a MILP solver is too slow to solve the problem. The MILP problem can be solved with a heuristic or it can be split into smaller problems. The latter method is used in the column generation method. This method is used in this thesis and the theory behind it is explained in section 2.2.2.

2.2.1 Mixed integer linear programming

For small integer problems with few variables often a MILP formulation is used and the problem is solved with a MILP solver. A standard notation for a MILP problem is as follows:

$$\min \quad \sum_{j=1}^{n} c_j^T x_j \tag{2.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j = b_i \quad i = 1, 2, ..., m \tag{2.2}$$

$$\sum_{j=1}^{n} a_{hj} x_j \leq b_h \quad h = m+1, m+2, ..., z \tag{2.3}$$

$$x_j \in \mathbb{Z}_0^+ \quad j = 1, 2, ..., r \tag{2.4a}$$

$$x_j \in \mathbb{R}_0^+ \quad j = r+1, r+2, ..., n. \tag{2.4b}$$

The objective function (2.1) states that the costs have to be minimized over all decision variables x_j. The equality constraints are given in (2.2), the inequality constraints are given in (2.3). For larger and more complex problems often the MILP formulation cannot be solved with a MILP solver due to long computational times. In most cases the problem becomes more complex and larger when there are more integer decision variables. The main reason for this is that the maximum steps a MILP solver needs to take to find the optimal solution is 2^j, when x_j is a binary variable.

2.2.2 Column generation

Different methods are used for solving complex MILP problems, which cannot be solved by a MILP solver in a reasonable time. One of these methods is called the column generation method. In large integer problems the number of decision variables is large, but not all of these are used in the final solution. Most of the decision variables are equal to zero and only some of them have a non-zero value. The column generation method tries to take advantage of this by only using a limited number of decision variables. The method starts with a feasible solution to the problem. The feasible solution consists of a subset of all decision variables. Each iteration a decision variable is added to the subset of decision variables that can improve the current solution, until the optimal solution is found. The column generation method is explained in-depth in the following paragraphs.

In order to create a column generation method, a master problem (MP) must first be formulated, which is an optimization problem with the following structure:

$$\min \quad \sum_{j \in J} c_j^T u_j \tag{2.5a}$$

$$\text{subject to:} \quad \sum_{j \in J} a_{ij} u_j \geq b_i \quad \forall i \in I \tag{2.5b}$$

$$u_j \in \{0,1\} \quad \forall j \in J. \tag{2.5c}$$

u_j are the decision variables, a_j are the associated columns, b_i are the resources available for each resource i. u_j describes if column a_j is present in the solution. Column a_j contains information on which resources of b_i are used. The master problem can be solved with a MILP solver. However, the problem size increases with 2^j. The problem becomes unsolvable in reasonable time in the case that there are many columns a_j and thus many decision variables u_j. For example: In a VSP each trip from a certain timetable has to be driven once during a day, while using as few buses as possible. The columns a_j are in this case defined as vehicle tasks. A vehicle task contains a set of trips driven by one bus during a day. In the master problem all possible vehicle tasks are enumerated. Thus, all possible combinations of trips that can be driven in one day by a bus are present in the MP. The minimum number of buses is obtained by choosing as few vehicle tasks as possible, while every trip is assigned once in the chosen vehicle tasks. The objective function determines which vehicle tasks a_j are chosen by choosing the linked decision variables u_j. The problem becomes unsolvable in reasonable time, when the number of vehicle tasks is huge.

The idea of column generation is to take advantage of that only a very small percentage of all vehicle tasks are present in the solution, while most vehicle tasks are not used at all. The column generation method uses a subset of J, namely $J' \in J$. The first step is to split up the MP into two parts. The first part is called the restricted master problem (RMP) and the second part is called the subproblem (SP). The RMP is derived from the master problem (MP). The RMP is as follows:

$$\min \quad \sum_{j \in J'} c_j^T u_j \tag{2.6a}$$

$$\text{subject to:} \quad \sum_{j \in J} a_{ij} u_j \geq b_i \quad \forall i \in I \tag{2.6b}$$

$$0 \leq u_j \quad \forall j \in J'. \tag{2.6c}$$

where J' is a small subset of J. The RMP is very similar to the MP. There are two differences the subset J' is used and the integer decision variables (2.6c) are relaxed. The idea is that it is also possible to find the optimal solution with a small subset of J, if the subset is correctly selected. Initially, J' consists of a set of columns that are a feasible solution of the RMP. Each iteration a new column is added to the RMP which can improve the current solution of the RMP. The selection of the new columns is done in the subproblem. This means for the previous example of a VSP, that not all vehicle tasks are enumerated in the formulation, but that only a small set of these vehicle tasks are present in the RMP and that the subproblem creates each iteration a new vehicle task that is added to the RMP.

To find new columns that improve the solution of the RMP in the subproblem, sensitivity analysis is needed. This analysis has to be performed on the RMP. In general, it is difficult to perform sensitivity analysis on a MILP problem, therefore the lower bounds and upper bounds of the integer decision variables (2.6c) are relaxed. This means that the integer decision variables have become continuous decision variables. The relaxed RMP can be solved with a LP solver. Sensitivity analysis can be performed on a LP problem with the use of the (dual) simplex algorithm, which is often used by LP solvers to solve a LP problem. The LP solver performs sensitivity analysis with the help of the dual problem of the RMP. To understand this the primal problem of the RMP is rewritten to the dual problem:

$$\min \quad \sum_{i \in I} y_i b_i \tag{2.7a}$$

$$\text{subject to:} \quad \sum_{i \in I} y_i a_{ij} \leq c_j \quad \forall j \in J' \tag{2.7b}$$

$$y_i \geq 0 \quad \forall i \in I. \tag{2.7c}$$

The dual problem computes for each constraint in the primal problem a dual variable. This dual variable is sometimes also called shadow price or dual price. The dual variable describes the change in value in the objective for each constraint, if the value of the right-hand side of the constraint in the primal problem (or in this case the RMP) is increased by one unit, while the rest of the RMP stays unchanged. Thus, it

describes the change in the objective function (2.6a), if one constraint out of the set of constraints (2.6b) is changed as follows:

$$\sum_{j \in J} a_{ij} u_j \geq b_i + 1, \tag{2.8}$$

while the other constraints remain the same. The dual variables can be combined with the costs of the decision variable in an equation that gives information on how to choose a new column to improve the objective value of the RMP. This equation is called the reduced cost and stems from the simplex algorithm, where it is used to determine which non-basic variable is placed in the basis. The equation is derived from the set of constraints (2.7b) and is as follows:

$$c_j - \sum_{i \in I} y_i a_i, \tag{2.9}$$

where c_j is the cost in the objective function for using the new decision variable, a_i describes the number of resources used for each constraint by the new decision variable and y_i are the dual variables per constraint of the primal problem. The reduced costs give the change in objective value when adding a small part of the new variable to the basis of the solution of the RMP. The column generation method uses the reduced cost equation (2.9) to find a column that improves the objective function (2.6a). The idea is to find a column with negative reduced costs. This is done with the help of the subproblem.

The equation (2.9) is the objective function of the subproblem. The values of a_i are the decision variables and are often integer variables. For a minimization problem the goal of the subproblem is to maximize $\sum_{i \in I} y_i a_i$. The subproblem of the RMP is as follows:

$$\min \quad c_j - \sum_{i \in I} y_i a_i \tag{2.10a}$$

$$\text{subject to:} \quad a_i \in \{0, 1\} \quad i = 1, 2, ..., I. \tag{2.10b}$$

In most models constraints are added to the subproblem to limit the choice of a_i. The solution of the subproblem, which is a column consisting of the variables a_i, is added to the RMP. This column is coupled to a new decision variable u_j. Then the RMP is solved again and the dual variables are updated. This cycle continues until the subproblem cannot find a column with negative reduced costs. Then the RMP is solved to optimality. This means for the VSP example, that the RMP is solved first. Then the subproblem is solved, which creates a new vehicle task with a new combination of trips. The vehicle task is added to the RMP. The RMP is solved with the new vehicle task. Then the subproblem is solved again. This continues until the subproblem cannot find set of vehicle task with negative reduced cots.

To explain column generation model in geometrical terms: The RMP can be seen as a polyhedron, which is created by the current constraints in the RMP. Solving the RMP with a simplex algorithm gives the vertex with the lowest objective costs for the current polyhedron. Then the subproblem has to be solved. The subproblem considers every non-basic variable. The objective function of the subproblem is the reduced cost, which is the slope of the arc from the current vertex to a new vertex. The new vertex is a solution with a new column in the basis compared to the current vertex. The subproblem adds a column with negative reduced costs to the RMP. Adding a variable to the basis with a negative slope means that the objective value has to decrease, except for the case that the new variable cannot be placed in the basis due to e.g. it can lead to an infeasible solution. It is not known by how much the objective value decreases, when a new variable is added. The reduced cost only tells something about the slope to the new vertex. The column with the lowest reduced costs is not in all cases the best option. The polyhedron that represents the RMP is changed when a new column is added. Therefore, the RMP is solved again and it finds the new vertex as solution. The optimal minimum solution is reached in the case that for all non-basic decision variables the reduced costs are non-negative. This is because in that case, it can be concluded that adding any possible new decision variable cannot lead to a vertex with lower costs.

An overview of column generation is given in Figure 2.1. The decision variables in the optimal solution are often not integer for large problems, but are fractional, since the lower bounds and upper bounds of the decision variables are relaxed in the RMP. The fractional result has to be converted to an integer solution. This is the last step of the column generation method. Converting the fractional optimal solution of the RMP can be done in multiple ways. In the paper of Joncour et al. [18] and the paper

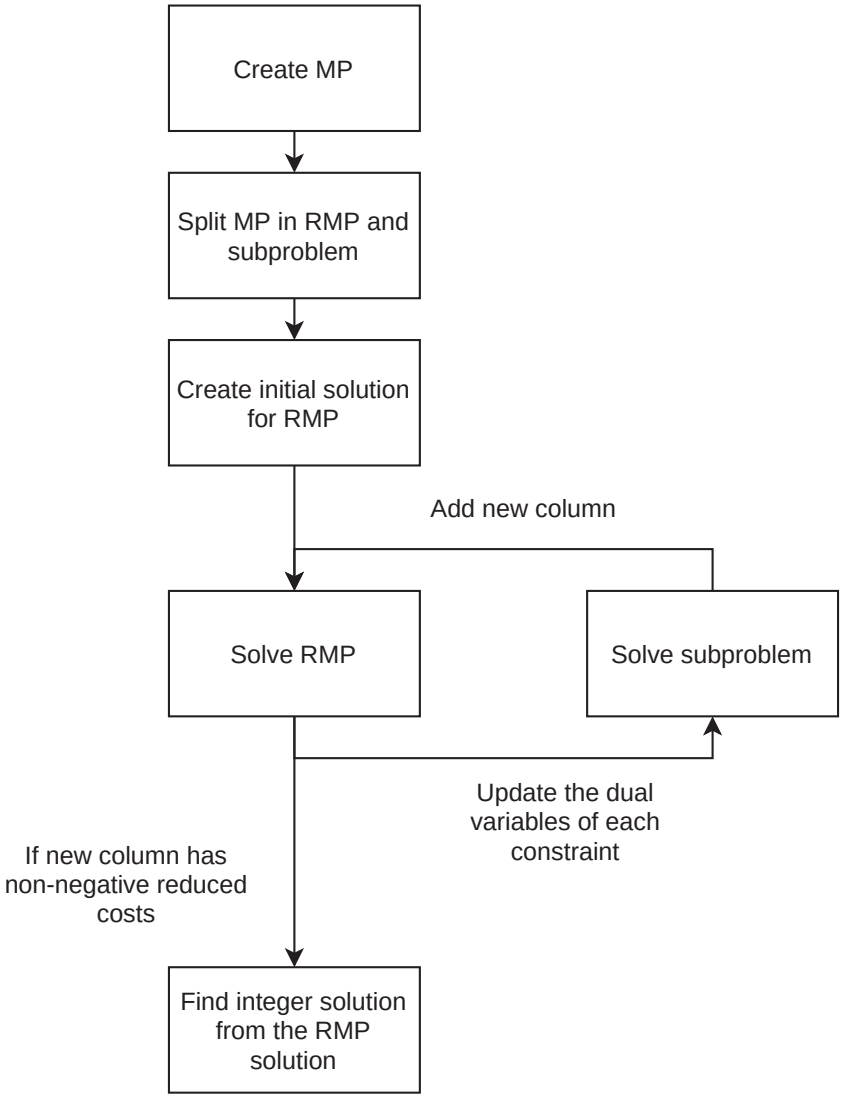


Figure 2.1: *Overview of the column generation method*

of Barnhardt et al. [19] some of the more common methods are shortly described. There are two main approaches to solve this problem.

The first one is called branch-and-price. Branch-and-price is based on the branch-and-bound technique, where the bounds are replaced with the objective costs of the RMP. The branch-and-price algorithms start by solving the RMP until optimality. Then one of the nodes, which are often decision variables, is branched on. For example one decision variable is set to one and another decision variable is set to zero. This leads to two new problems and thus two different RMP's. The two problems are often smaller than the original problem. These are solved again with column generation until optimality is reached. Then again a node is branched. This continues until the optimal integer solution is found. In a VSP or a VRP (vehicle routing problem) often the arcs in a graph are branched on instead of the decision variables, where the arcs in the graph are the deadhead trips and the nodes the trips. The reason for this is that it makes the problem faster smaller, which leads to a reduction in computational time [20]. The nodes are pruned in a minimization problem when the objective value is higher than the lower bound. A lower bound can be found by obtaining a feasible integer solution. These are often found with the help of primal heuristics, before applying the branch-and-price algorithm. Primal heuristics try to find a fast and good integer solution from a fractional solution.

The use of primal heuristics is the second main approach to find an integer solution from the fractional RMP solution. Primal heuristics are also used just for the purpose of obtaining a solution to the

CG model. The advantage of using a primal heuristic is that it is relatively fast. The disadvantage is that the solution can be suboptimal, whereas the branch-and-price algorithm leads to an optimal solution. There are multiple primal heuristics used in literature. In general two heuristics are used the most which are the restricted master heuristic and the diving heuristic. The first one solves the RMP with the current decision variables in the RMP, where the upper and lower bound of the decision variables are not relaxed. This means that the problem is a MILP problem and can be solved with a MILP solver. This has two disadvantages. The first one is that the RMP can contain a lot of columns and thus a lot of decision variables. The problem might become too large to solve with a MILP in the case that all these decision variables become integer variables. The second disadvantage is that this method often does not give the optimal solution, since the columns needed for finding an optimal fractional solution are often different from the columns needed for finding an optimal solution to the MILP problem. The second primal heuristic is the diving heuristic of which the idea is based on rounding the fractional decision variables in steps. One decision variable is rounded up, when the RMP is solved to optimality. The decision variable with the largest fractional value is rounded up and is fixed to one in the RMP. This leads to a new RMP and thus the RMP has to be solved again until optimality with the help of column generation. Then a new decision variable is rounded up. This cycle continues until there are no more fractional decision variables and all decision variables are integer. In this thesis it is chosen to use a diving heuristic. The reason for this is that it is faster than the restricted master heuristic and it also often creates a better solution. The branch-and-price method is not chosen due to the possible long computational time.

2.3 Previous work

As mentioned earlier, this thesis is made with as starting point the work of two previous students. The first work is from Monhemius [3]. Monhemius had formulated a MILP problem to solve the eVSP. The model includes charging, which is modelled in continuous time, a choice of different vehicle or charger types, a minimum charge time, minimum and maximum break time and multiple charger locations. The main problem of this approach is the long computational time of solving a MILP problem with a MILP solver. The problem with the model is, according to Monhemius, that there are too many variables to solve the problem with a MILP, since solving a problem with a MILP solver is exhaustive. Monhemius shows that the number of variables have an exponential relation to the computational time and concludes that using a MILP is not a viable option to solve an eVSP. Monhemius recommends using different techniques to solve the eVSP, one of them is the column generation method.

This method is used in the thesis of Wijnheijmer [3]. First, Wijnheijmer has created a concurrent scheduler to solve the problem of VDL. It finds the optimal solution in 17 of the 18 test cases. These cases are however very small problems and most methods or algorithms for solving an eVSP find an optimal solution for very small eVSP's. Wijnheijmer concludes that the results of the concurrent scheduler are not always optimal and can be improved in some cases. The column generation method improves the result between 8.1% and 18.7% in the work of Adler [4], compared to the concurrent scheduler of Adler. Note that the problem is different and in this example the CG method is solved until optimality and the results are only for relatively small examples. Wijnheijmer has created a column generation model to improve the results. The objective function of the RMP (2.11a) is to minimize the number of vehicle tasks and the amount of energy that is used. The constraints are that each trip has to be driven at least once by a bus (2.11b), the number of chargers have to stay below a certain number during each time block (2.11c) and the amount of energy charged cannot exceed the maximum allowable amount of energy during each time block (2.11d). The charging is thus modelled in discrete time. Wijnheijmer divides the schedule in a hundred time blocks to take into account charging. The RMP is shown below, because it is often used in Chapter 3:

$$\text{obj min} \quad \sum_{v \in V'} c_v u_v \tag{2.11a}$$

$$\text{subject to} \quad \sum_{v \in V'} x_{tv} u_v \geq 1 \quad \forall t \in T \tag{2.11b}$$

$$\sum_{v \in V'} s_{zv} u_v \geq n_s \quad \forall z \in Z \tag{2.11c}$$

$$\sum_{v \in V'} e_{zv} u_v \geq \epsilon_{max} \quad \forall z \in Z \tag{2.11d}$$

$$u_v \geq 0 \quad \forall v \in V', \tag{2.11e}$$

where V' are the vehicle tasks, u_v is a decision variable that is one if vehicle task v is used in the solution, x_{tv} is one if trip t is assigned to vehicle task v, s_{zv} is one if a charger is used in vehicle task v during time block z, n_s is the maximum number of chargers allowed, e_{zv} is the amount of energy charger during time block z by vehicle task v and ϵ_{max} is the maximum allowed amount of energy charged during a time block. There are three different dual variables for this RMP. The first one is π_t and corresponds with constraint (2.11b) and it gives a shadow price for each trip. The second one is θ_ζ, which corresponds to the constraint (2.11c) and gives a shadow price for the number of chargers used in each time block. The last dual variable is ρ_ζ, which corresponds to constraint (2.11d) and gives a shadow price for the total energy chargers per time block. The equation for the reduced cost [3] is as follows:

$$c_v - \sum_{t \in T} \pi_t \delta_t + \sum_{z \in Z} (\theta_z \sigma_z + \rho_z \epsilon_z), \tag{2.12}$$

where δ_t is a decision variable and determines if trip t is used, σ_z determines if a charging session is used at timeblock z and ϵ_z determines the amount of energy charged in timeblock z.

The subproblem has the objective to obtain a new vehicle task with the lowest reduced cost. The vehicle task describes the day task for a bus. It includes the information on which trips are driven, in which time blocks a charger is used and how much energy there is charged during each time block. There are multiple constraints for the new vehicle task: The first one is that trips can not be driven simultaneously, charging cannot happen while a trip is driven, the SoC always has to be higher than a certain minimum SoC and lower than a certain maximum SoC and the last constraint is that there is a minimum charge time.

This model is simulated and tested by Wijnheijmer with 4 timetables. Only the small timetables with 14 and 15 trips are fully simulated for a limited number of chargers. The tests give promising results, namely the schedule created by the CG model is cheaper than the schedule of the concurrent scheduler. A decent solution is not found for the larger problems. For Timetable 4, which consists of 203 trips and 1 charger, a solution is found, but the costs are high. For Timetable 7, which consists of 1096 trips and 4 chargers, not one solution is found due to a long computational time. The details of Timetable 4 and Timetable 7 can be found in Appendix A.

The starting point of this thesis is the CG model of Wijnheijmer. Therefore, it is interesting to go deeper into the problems of the model. There are two main problems in the model of Wijnheijmer. The first one is the long computational time using a CG method. The column generation model of Wijnheijmer solves an eVSP. The computational time of the model is tested for schedules with a different number of trips and 4 chargers. The results are shown below:

Number of trips	Computational time [sec]	Bottleneck
13	42	print function
99	394	intlinprog function of subproblem
203	980	intlinprog function of subproblem
267	2119	intlinprog function of subproblem
332	4343	intlinprog function of subproblem
477	53565 (stopped after 3 iterations)	intlinprog function of subproblem

Table 2.1: *Computational time for a varying number of trips of the CG model of Wijnheijmer (Using a zbook 15 with an Intel core i7-4700 MQ with 8 GB RAM)*

The model cannot solve the timetable of 477 trips within a reasonable time. Note that there is a stop criterion in the model, which makes the CG model stop after 200 iterations, which means that not all timetables are solved until optimality. The computational time increases in proportion by more than the number of trip increases. For larger schedules it can be concluded that the computational time of the model of Wijnheijmer is too long. The bottleneck in the CG model of Wijnheijmer is the subproblem. The largest schedule that VDL needs to simulate contains 4500 trips, but the target goal is to simulate

the timetable of Rotterdam, which includes 1096 trips. One of the main problems is to accelerate solving the subproblem. The combination of charging and assigning trips makes the problem complex and difficult to solve. Chapter 3 is dedicated to find a method to accelerate solving the subproblem.

The second problem is that parts of the requirements for using the schedule in practice are missing. The first requirement that is missing is the introduction of deadhead trips in the model. A deadhead trip is the trip a bus needs to drive between two trips or a charger and a trip in a model. The second requirement that is missing is the option of having multiple charger locations. The current model only allows charging at the depot. The third part that still needs to be implemented is the use of non-linear charging instead of linear charging. In the current model of Wijnheijmer the buses are charged according to a linear charge curve, while in practice the charge curves are non-linear. The fourth extension that needs to be implemented is that a minimum shift time has to be introduced. In the current model it can happen that one bus has to drive one single trip, which in theory can be part of an optimal solution, but in practice the customer does not accept this. The last requirement that is not implemented is that there is a limit on how long a bus can stand still at a location. Buses are only allowed to stand still for a long time in the depot. In the new CG model these requirements have to be implemented.

The model of Wijnheijmer seems to have potential to work well, if the two problems are solved. First the problem of a long computational time of the subproblem has to be fixed, before making the problem more complex with the extra constraints. This only leads to longer computational times to solve the model. In Chapter 3 heuristics are implemented to try to speed up the solving of the subproblem.

2.4 Summary

The problem the tool has to solve is known as an electric vehicle scheduling problem (eVSP). In this chapter previous work on solving the eVSP has been given. Then an introduction has been given on solving large mixed linear integer problems, where the theory behind the column generation method has been explained. At the end of the chapter the work of two previous students who have tried to solve the same problem has been given. The problems of both their works have been discussed. The work of Wijnheijmer [3] can have potential, if the subproblem is solved faster and if the model can be extended to include multiple constraints. The most difficult problem to solve is the long computational time of the subproblem. Therefore, the next step is to find a method that accelerates solving the subproblem of Wijnheijmer [3].

Chapter 3

Methods for solving the subproblem

The starting point of this thesis is the column generation model of Wijnheijmer[3], which has been discussed in Chapter 2. The main problem of the model of Wijnheijmer is the long computational time of the subproblem. The objective of the subproblem of Wijnheijmer is to find a vehicle task with negative reduced costs. The main goal of this chapter is to find a fast and accurate method to solve the subproblem that creates a vehicle task with negative reduced costs.

The most often used solution is solving the subproblem with a heuristic, which is described in [12] and [21]. The subproblem does not have to be solved until optimality. Any created vehicle task with negative reduced costs can improve the objective value of the RMP as is explained in Chapter 2. Three heuristics for solving the subproblem of Wijnheijmer [3] are introduced in this chapter. These are a greedy algorithm, a genetic algorithm and a diving heuristic. The subproblem can also be solved faster by reducing the size of the subproblem of Wijnheijmer [3], which can only be made smaller when both the RMP and the subproblem of Wijnheijmer [3] are reformulated. A method is introduced in section 3.3 which formulates a model such that the subproblem of creating a vehicle task is split in multiple subproblems that are solved in series. The method is called the multiple step column generation method. In section 3.4 a simplified model of an eVSP is used to show the potential of rewriting the subproblem to a shortest path problem (SPP). This has the benefit that algorithms are already available that can be used to solve the subproblem efficiently. This method has been implemented in the final model. The other methods have not been implemented in the final model for various reasons, which are explained in this chapter.

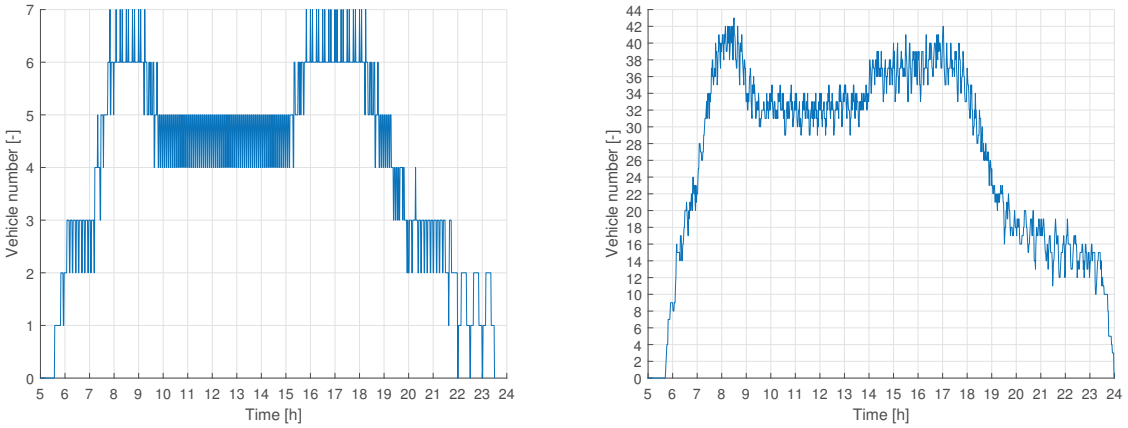
The performances of the different methods have to be measured in order to decide if a method is sufficient to use in the tool. The main performance parameter in all models is the number of buses used in a schedule. A lower bound for the number of buses needed for a schedule is created to get an estimation on how far a solution of one of the methods is off the optimal solution. The lower bound is introduced in section 3.1. The Timetable 4 and Timetable 7 are often used in this chapter. Timetable 4 is restricted to the use of one charger and Timetable 7 is restricted to the use of 4 chargers. The details of these timetables can be found in Appendix A. These timetables have also been used by Wijnheijmer [3]. This makes a comparison possible of the performance of the models of Wijnheijmer and the new models. In this chapter only large models are used, since the timetables in practice are large and most problems occur when creating schedules for large timetables.

3.1 Lower bounds

In this section lower bounds are created to be able to estimate the quality of the solutions of the new methods. The quality of the solution is mainly based on the number of buses that are used in a schedule for a given timetable. The number of fast chargers used in a solution is also a factor. In this section three lower bounds are introduced. The first two lower bounds determine a lower bound for the number of buses used. The first lower bound is based on the number of trips driven simultaneously and the second one is based on the total energy needed to drive all trips in a timetable as a function of time. These lower bounds are discussed first. The third lower bound is used to find a minimum value for the number of fast chargers needed to create a schedule given a certain number of buses. This lower bound

is also based on the total energy needed to drive a timetable as a function of time and is last discussed in this section.

The first lower bound is based on the trips that need to be driven simultaneously and is similar to the lower bound used by Wijnheijmer [3]. The lower bound is determined by counting the amount of trips driven simultaneously each minute. The lower bounds for Timetable 4 and Timetable 7 are shown in Figure 3.1. The lower bound for the number of buses is 7 for Timetable 4 and 43 for Timetable 7. In



(a) *Timetable 4* **(b)** *Timetable 7*

Figure 3.1: *A lower bound of the number of buses as function of time based on simultaneously driven trips*

each figure two peaks, which correlate with the rush hour times, can be noticed. These two rush hour peaks are typically seen in other timetables for cities.

The second lower bound is based on the progression of the total amount of energy in the model during a day. The model starts with one bus. At the first minute of the day the total energy available in the system is equal to the number of buses times the maximum allowable energy per bus. This can be formulated as

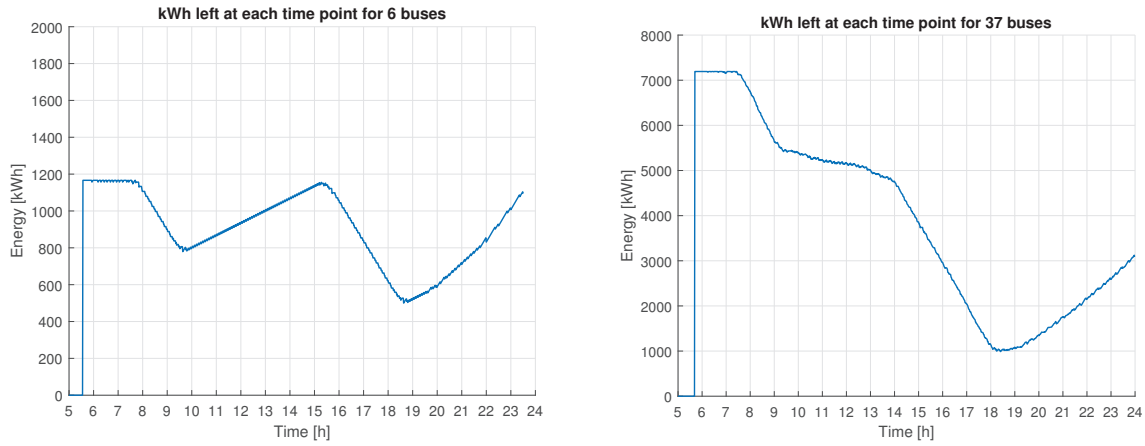
$$e_{tot}(1) = be_{max}, \tag{3.1}$$

where $e_{tot}(1)$ is the total amount of energy available at the first minute of the schedule, e_{max} is the maximum allowable amount of energy in a bus and b is the number of buses used. Each minute $e_{tot}(t)$ is updated and is reduced with the energy costs of a trip e_t, if trip t is finished. Each minute it is tracked if the total amount of energy available in the model is larger than the current number of buses times the minimum allowable amount of energy per bus, when this is not the case a bus is added to the model. Thus:

$$e_{tot}(t) \geq be_{min}, \tag{3.2}$$

where $e_{tot}(t)$ is the total amount of energy in the system and e_{min} is the minimum allowable amount of energy per bus. The model is reset, when equation (3.2) does not hold. An extra bus is added to the system and the model is run again. This continues until equation (3.2) holds for every minute of the day.

The variable $e_{tot}(t)$ can also increase, when there are chargers available. Buses also have to be available to use the charger. The number of buses available can be calculated by subtracting the number of currently driven trips from the number of total buses. The total amount of energy is then improved by the charging rate times the number of available chargers (or the number of available buses) and cannot exceed the number of buses times the maximum allowable amount of energy in a bus. The results for Timetable 4 and Timetable 7 are shown in Figure 3.2, which shows the progression of the total energy in the system for the minimum amount of buses that can be used. The total energy becomes lower than the sum of the minimum SoC needed of all buses, when one bus less is used. In both cases the lower bound is lower than the lower bound based on simultaneously driven trips. The lower bound based on total energy in the model can lead to a tighter lower bound in some cases than the lower bound based on simultaneously driven trips, but in most practical timetables it is a less tight lower bound. A combination of these lower bounds is used in attempt to solve the eVSP problem with a heuristic instead of taking

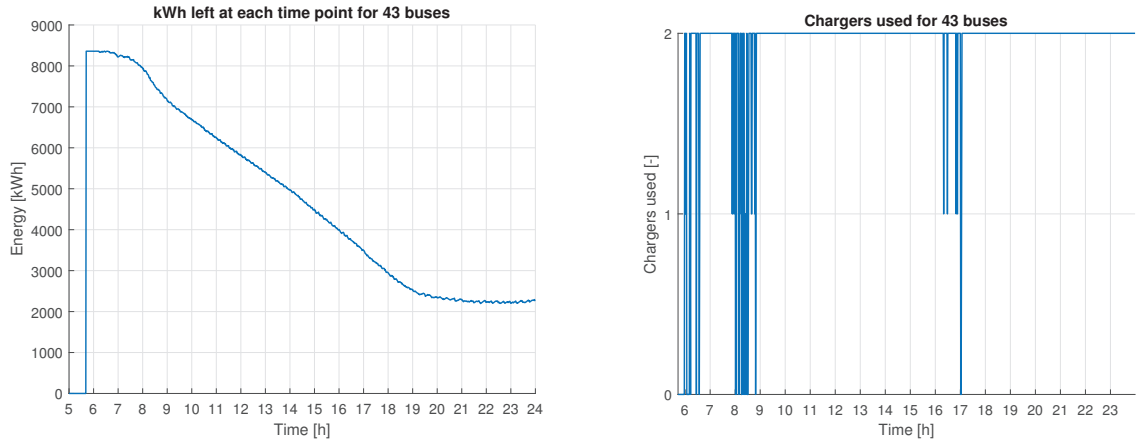


(a) *Timetable 4: Total energy in the system for 6 buses* **(b)** *Timetable 7: Total energy in the system for 37 buses*

Figure 3.2: *Lower bound based on available energy in system for Timetable 4 and Timetable 7*

the column generation model approach to solve the eVSP. This heuristic is described in Appendix B.1, but it does not perform well enough.

The third bound is a lower bound to determine the number of fast chargers that have to be used, when the number of buses used is known. The lower bound works on the same principle as the previous described lower bound and is almost similar to the lower bound used by Monhemius [2]. In this lower bound the number of buses is fixed and the number of chargers are changed. An extra charger is added to the system, if equation (3.2) does not hold. The number of used chargers in the model is the lower bound for the minimum needed amount of chargers. The lower bound for the number of chargers for Timetable 7 with 43 buses is shown in Figure 3.3. Figure 3.3b shows that the lower bound for chargers for



(a) *Timetable 7: Total energy in the system for 43 buses and two chargers* **(b)** *Timetable 7: Number of used chargers per minute*

Figure 3.3: *Lower bound for chargers based on available energy in system for Timetable 7 for the use of 43 buses*

Timetable 7 is 2. The lower bound is calculated differently, when there are slow chargers available. Slow chargers are chargers that are available at the depot. Slow chargers are mainly used to charge the bus at night, but can also be used to charge the bus during the day. There are as many slow chargers available as there are buses. Taking these into account the lower bound has to be found in a slightly different manner for the number of fast chargers. The model that creates the lower bound starts with as many slow charger as there are buses. A slow charger is exchanged for a fast charger, when equation (3.2) does not hold. This leads to a lower bound for the number of fast chargers. This lower bound is conservative, which means that probably the lower bound often cannot be reached with the optimal solution.

3.2 Heuristics

Heuristics are methods to obtain results in a faster manner than the exact solution methods but are not guaranteed to be optimal. Heuristics are often used to solve the subproblem in the CG model. The advantage of using a heuristic is that it can solve the subproblem of Wijnheijmer [3] much faster than a MILP solver. The advantage of solving the subproblem with a MILP solver is that it finds the column with the lowest reduced costs in case of a minimization problem. Finding the column with the lowest reduced costs is not necessary, since any column with negative reduced costs can decrease the objective value. Another disadvantage of using a MILP solver is that most freely available MILP solvers only have one solution as output instead of multiple solutions. There is a possibility that the model structure of Wijnheijmer [3] is sufficient for use, if a heuristic is used instead of a MILP solver. Multiple heuristics can also be used. For example one heuristic can be used in the first few iterations to quickly improve the RMP (restricted master problem) solution and dual variables, while another more accurate heuristic that is slower can be used for the last iterations. Using multiple heuristics in one iteration is also possible. In this section multiple algorithms are discussed that can replace the current MILP solver that is used in the subproblem of Wijnheijmer [3].

3.2.1 Placement of restrictions on the MILP solver

The main problem of the model of Wijnheijmer is that it takes too much time for the MILP solver to find an optimal solution. Using a different and faster MILP solver can reduce the computational time. The current solver implemented is the intlinprog solver of MATLAB, but this is not the fastest solver. The website of prof. Mittelman [22] compares the computational time of multiple MILP solvers for multiple problems. The commercial Gurobi solver is on average more than 10 times as fast as the current MILP solver. Not all MILP solvers can be used, because a requirement of VDL is that the solver has to be freely available (with exception to the solvers of MATLAB). The CBC MILP solver is almost twice as fast on average as the intlinprog solver and is free for use. The solver is implemented in the model of Wijnheijmer. This leads to half the computational time:

Number of trips	Computational time(sec)
13	34
99	176
203	982
267	1694

Table 3.1: *Computational time for a varying number of trips of the CG model of Wijnheijmer with the CBC MILP solver (Using a zbook 15 with an Intel core i7-4700 MQ with 8 GB RAM)*

It can be concluded that the computational time is reduced, when comparing Table 3.1 with Table 2.1. The computational time is not reduced enough, because for timetables with a thousand trips the computational time is still too long. An option to reduce the computational time even more is to solve the MILP suboptimally with the MILP solver. The idea is that there is a possibility that the MILP solver can find a suboptimal solution within a reasonable time. MILP solvers are often already equipped with algorithms to find a fast solution, because a decent lower bound for the problem can be found. This is useful for the branch & bound method. The MILP solver can be forced to solve a problem suboptimal by setting the maximum iterations to a lower number and by setting the maximum solve time to a low time. The maximum number of iterations and the maximum solve time has been set to gradually increase over the iterations, where in the last few iterations the MILP solver is solved till optimality. The timetable with 99 trips has been solved in 141.4 seconds. It reduces the computational time compared to the result in Table 3.1. The simulation for Timetable 7 has been stopped manually after 50 iterations and 13 hours simulation time. It can be concluded that using a MILP solver for solving the subproblem of the model of Wijnheijmer is not an option, instead a heuristic has to be made for this problem.

3.2.2 Greedy algorithm

A relative simple and fast heuristic that can be implemented is the greedy algorithm, which is a heuristic where locally at each step the best choice is made. It is often used to find a fast and good approximation

of the solution and it is implemented to investigate the performance of such an algorithm in solving the subproblem of Wijnheijmer. The algorithm tries to create a vehicle task (i.e. a day task for one bus) based on the dual variables for trips π_t and the dual variables for charging sessions θ_ζ. These dual variables are based on the dual of the RMP (2.11a).

The algorithm determines which trips are chosen based on the dual variables π_t of each trip. The trips are prioritized over the chargers, since it is hard to determine how much charging needs to be done before the trips are planned. The algorithm is shown in the Appendix B.2 and it consists of three parts. The first part assigns a trip with the highest dual variable π_t to a bus. Consecutive driven trips are planned with positive dual variables π_t until the minimum SoC is reached, then the battery is fully charged. The second part is used to remove the surplus energy, which is left in the bus at the end of the day. The latest charging sessions are deleted until there is no surplus energy any more at the end of the vehicle task. The third part is implemented to take into account the dual variables for the charging sessions. The first step of the third part is to replace charging sessions with high positive dual variables to empty earlier time blocks, where the dual variables for the charging sessions are low. The second step of the third part is to replace earlier assigned trips with charging sessions, when this is profitable.

The greedy algorithm has been implemented in the model of Wijnheijmer and the model has been simulated. The maximum amount of iterations has been set to 800. The fractional results of the relaxed RMP have been rounded with the rounding algorithm of Wijnheijmer [3]. The RMP has been re-optimized by creating 50 new columns, after a variable has been rounded up. The costs for a bus have been set to 111.11 euro per day and the costs for the energy have been set to 0.2 euro per kWh [3]. The results of the CG model with the greedy algorithm can be found in Table 3.2 and Figure 3.1.

	Costs Timetable 4	Simulation time Timetable 4	Costs Timetable 7	Simulation time Timetable 7
Concurrent scheduler	1294 euro	4.38 sec	7902 euro	81.92 sec
CG with MILP	9465 euro	1712 sec	-	-
CG with Greedy algorithm	1550 euro	34.04 sec	7404 euro	1421 sec

Table 3.2: *Results of the greedy algorithm compared to the results of Wijnheijmer [3]*

There are mixed results in Table 3.2. For Timetable 7 the CG model plus the greedy algorithm works better than the other options. For Timetable 4 the concurrent scheduler is the best method to use. The conclusion that the results are poor can be drawn from Figure 3.1. The number of buses used is well above the lower bound for the number of buses, which is 43 buses. There are also too many empty gaps in the schedule.

The algorithm can be run longer with more columns, but this is not going to lead to much better results, because the improvement in the objective value of the RMP per iteration has been too low. The implemented greedy algorithm can still be improved in a few ways. The main improvement is to fill the gaps with trips of the relocated charging sessions. The greedy algorithm has not been improved anymore, since it is unlikely that improving the CG model with extra rules is going to give accurate enough results. It is too difficult to create the perfect set of rules for choosing the best choice at each time step. The main reason is that it is beforehand unknown what the most beneficial place and duration is to charge. The greedy algorithm can still serve as a heuristic to reduce the objective value quickly of the RMP. A more accurate algorithm is needed for the later phase of the simulation of the column generation method.

3.2.3 Genetic algorithm

A genetic algorithm has been implemented to investigate if this algorithm is able to solve the subproblem accurately. The genetic algorithm is based on the evolution theory of natural selection, which makes it an evolutionary algorithm. It consists of 3 main steps. The first step is to create an initial population, which is in this case the solution of the greedy algorithm. Thus, the population is a set of columns. The second step is to select a part of the population, which is used to create the new generation. There are multiple different methods used for this step. These methods are called selection methods. The third

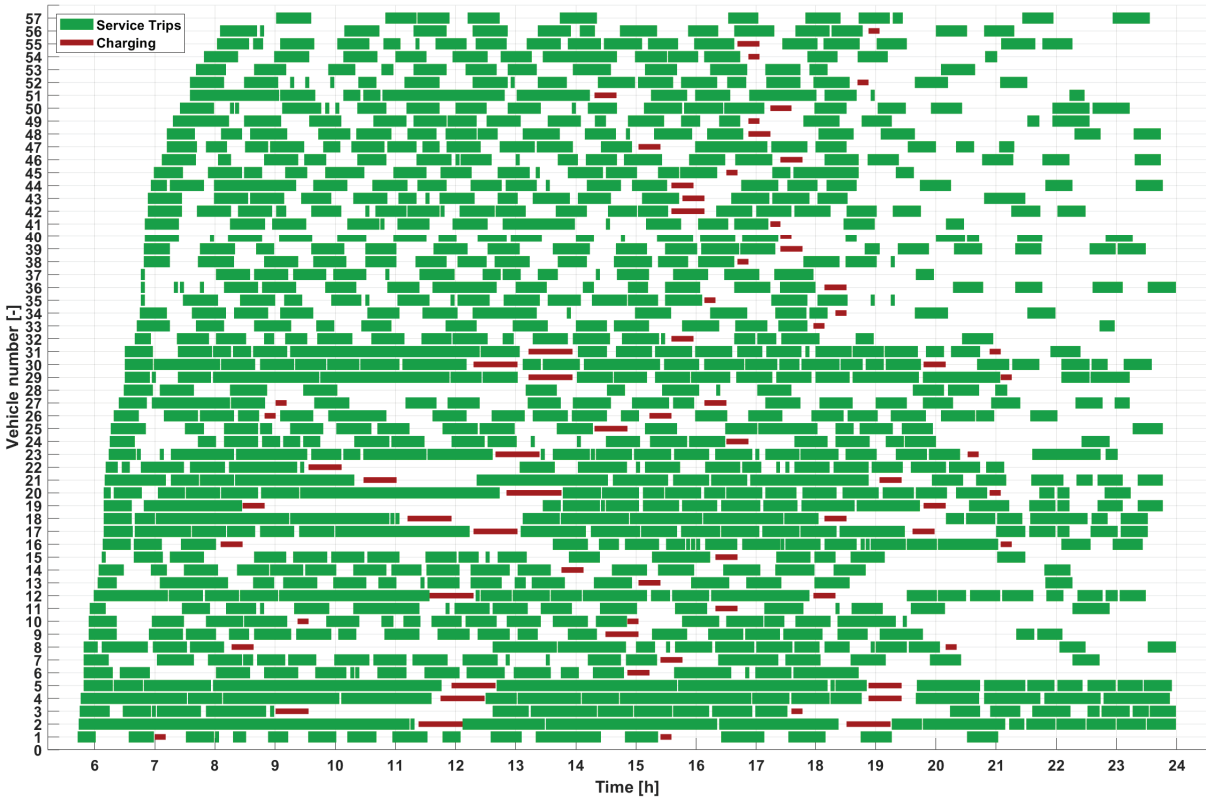


Figure 3.1: *A schedule for Timetable 7 simulated with a greedy algorithm*

step is to create the new generation, which is often done based on crossover or mutation. Crossover is performed by making a new variable based on assigning chromosomes of different variables to the new variable and mutation is performed by changing one of the chromosomes of the variable. A chromosome is a part of the column and the variables are the columns. Crossover mixes multiple parts of columns and mutation changes a part of the column.

First, a standard genetic algorithm that is available in the global optimization toolbox of MATLAB has been implemented. The standard genetic algorithm often gives an infeasible solution for small timetables such as Timetable 1. Feasible solutions can be found for Timetable 1, after tuning the parameters of the algorithm. The computational time is long and the results are often not feasible. It can be concluded that the standard genetic algorithm of MATLAB is not a viable option.

Heuristics that are created to be able to solve a large set of problems often obtain a worse solution for a problem than heuristics that are created for that problem specifically. Therefore, a genetic algorithm has been made for this subproblem specifically. The main idea of creating a genetic algorithm has been to check, if such an algorithm has the potential to solve the subproblem accurately. The algorithm can be found in the Appendix B.3. The initial population of the genetic algorithm is the solution of the greedy algorithm. Four chromosomes are used. Thus, each vehicle task is split in four parts. The number of chromosomes per vehicle task has to be limited due to the chance of obtaining an infeasible vehicle task. These can occur, because of violation of the maximum or minimum SoC allowed in the bus.

As selection method the tournament selection has been chosen, which works as follows: In a tournament a participant for the next generation is chosen based on choosing a random number between one and zero. Each participant has an interval, which represents the percentage of negative reduced costs of one participant compared to the sum of all the negative reduced costs of all the participants. The participants are the vehicle tasks and the chromosomes. For the four different groups of chromosomes and for the group of complete vehicle tasks a certain number of tournaments are held. In the last step of the genetic algorithm only crossover actions are performed. Crossovers are done between the tournament winners of the chromosomes parts and the tournament winners of the vehicle tasks. One of the chromo-

somes of a vehicle task is deleted and replaced with a chromosome that won the tournament. Mutations are not performed due to the complexity of implementing this. It is difficult to respect all constraints, when applying a mutation to a chromosome or a vehicle task. The best columns of each generation are stored.

The greedy algorithm has been used to create a decent initial solution. The genetic algorithm has been simulated with a number of different values for the number of generations, the number of tournaments and the number of participants. The genetic algorithm creates a lot of new columns each iteration. More columns are created when the number of generations has been set to a higher number. The computational time increases with the number of generations. The number of tournaments and participants do not have much influence on the quality of columns found. In general, the genetic algorithm is slow and is not able to find much better columns than the greedy algorithm. The final costs barely reduce when using the genetic algorithm. From the results it can be concluded that the current genetic algorithm does not have the potential to solve the subproblem in CG model. A genetic algorithm can maybe still be used to solve this subproblem, but it is a complex task to create an efficient genetic algorithm for this subproblem, therefore it has been decided to look for a different method.

3.2.4 Diving heuristic

A diving heuristic is a heuristic that dives into a branch-and-bound tree. Instead of calculating the objective value of every branch it follows at each point in the tree one branch of all the possible branches. The algorithm can be found in Appendix B.4, The first step is to relax the subproblem of Wijnheijmer [3]. This means that all integer variables are converted to continuous variables by relaxing their upper and lower bounds. The subproblem can then be solved with the help of an LP solver. A fractional solution is the result of solving the relaxed subproblem with an LP solver and has to be converted to an integer solution. All variables that are equal to one in the fractional solution are fixed to one with help of equality constraints. The variable with the largest fractional value is fixed to one, if there is no non-fixed integer variables. The subproblem is then solved again. This cycle continues till all non-zero variables are fixated to one.

The algorithm has been implemented and simulated. The problem is that the diving heuristic is very slow for large problems. The simulation time is a few minutes per iteration for Timetable 7. The diving heuristic often finds a good result, but it also often finds a poor result. Sometimes results become infeasible, when a variable up is rounded up. This variable then has to be set equal to zero. One of the many problems is to determine if the trips or the charging sessions have to be fixated first. The best result after a lot of tuning for Timetable 4 using all three the heuristics is a cost of 1480 euro and the use of 10 buses, which is too high. It can be concluded that making heuristics manually is not the best option to solve this problem.

3.3 Multiple step column generation

Finding the best type of heuristic to solve the subproblem accurately has been difficult, especially writing an accurate heuristic for the subproblem has been a complex task. Therefore, another approach is taken, where the idea is to make the subproblem smaller. This also leads to a lower computational time. The subproblem can be made smaller by splitting the subproblem in multiple problems. It is one of the requirements of VDL to have a minimum shift time, since a customer does not want to send a bus from the depot to drive only one short trip. The subproblem of creating a vehicle task can be split into two parts, because of this requirement. One part creates the shifts of trips and the other part creates the vehicle tasks of shifts.

The size of the subproblem of creating vehicle tasks out of shifts is further reduced by taking the charging constraints out of the subproblem. The subproblem of creating vehicle tasks only has to guarantee that it is still possible to manage to respect the SoC constraints, when charging sessions are assigned to the vehicle task. The objective of the new charging part is to create charger tasks for a charger. Each vehicle task has charging intervals, which are the time intervals in-between two shifts. These charging intervals can be assigned to a charger similar to the assignment of shifts to buses. There are however a few differences compared to assigning shifts to vehicle tasks. The main difference is that charging does

not happen during the complete charging interval, but the charging session can also only use a part of the charging interval. This leads to that charging intervals may overlap in some cases in the same charger task. Splitting the subproblem of creating a vehicle task in multiple problems, also means that the RMP has to be changed. The model of Wijnheijmer [3] is not used anymore and therefore a new model is formulated in this section.

The problem is now split in 4 parts. These problems are the assignment of trips to shifts, the assignment of shifts to vehicle tasks, the assignment of charging sessions to charger tasks and the RMP that has to choose the vehicle task. The main idea is to create a column generation model with multiple steps, where each step solves one of three steps described above. An overview of the model is given in Figure 3.1 to give a more clear view on the model. The three subproblems are placed in series, because

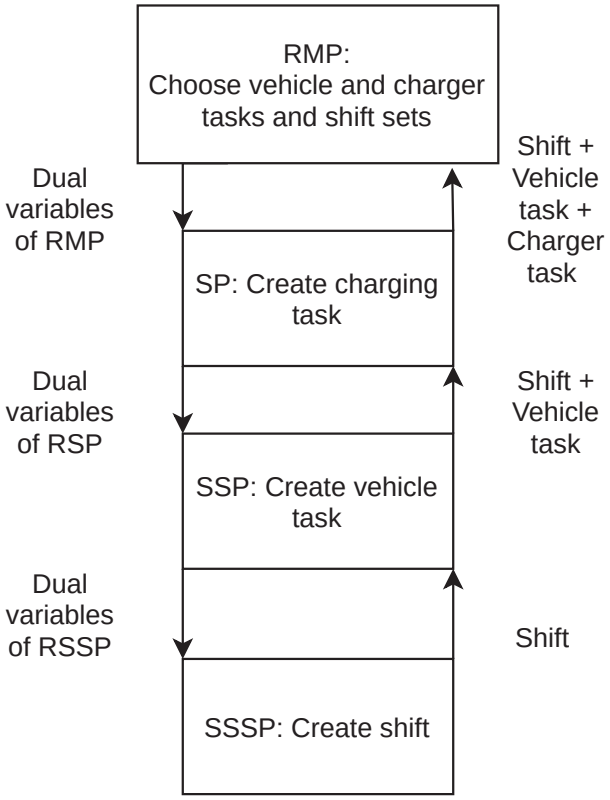


Figure 3.1: *Overview of the multiple step column generation model*

each subproblem depends on another subproblem. The new vehicle task depends on the creation of shifts and the new charger task depends on the new vehicle task, since the charging intervals are obtained from the interval between two shifts in the vehicle tasks. The main problem is connecting the three subproblems. Each subproblem sends a part of the new column to the step above. The column consists of a new shift, a new vehicle task and a new charger task. Each step has to give information of the relaxed problem of that step to a step below in order to find a solution that can improve the relaxed problem. A relaxed problem means that a MILP problem is converted to an LP problem by relaxing the integer constraints. Information is transferred with the help of the dual variables. The main problem is creating a correct formulation that can do this. Such a formulation has not been found. Another problem is that at each step the relaxed problem is solved. This means that this is often a non-integer solution, while an integer solution is needed. The formulation of the new RMP of this model is given in the Appendix B.5.

The main problem of this method is that new charging intervals can exist, when the SSP (subsubproblem) creates a new vehicle task. These charging intervals are not yet taken into account in the RMP and therefore no dual variables for these charging intervals are available in the SP. The new charging interval cannot be used in the SP. It creates a new row in the RMP and in order to obtain the dual variables of this new charging interval first the RMP has to be solved again. The same holds for when the SSSP (subsubsubproblem) creates a new shift, then the SSP cannot use this shift since the dual variable of this shift is now known. An in detail explanation is given in Appendix B.5. There are two other

possible problems with using a multiple step CG model. The first problem is the unknown influence on the optimality gap the rounding of the fractional relaxed subproblem and relaxed SSP solution to an integer solution in each step has. The second problem is that it can be difficult to implement deadhead trips without increasing complexity of the problem significantly. It can be concluded that a different method is needed.

3.4 Create a graph to obtain a shortest path problem

The PhD theses of Adler [4] and van Kooten Niekerk [16] propose a method that includes a combination of reducing the complexity of the subproblem and using a heuristic to solve the subproblem of creating a vehicle task in a VSP. The method is to create a graph based on the timetable. The nodes in the graph represent the trips and the arcs represent deadhead trips, since the arcs are the connection between two nodes, i.e., between two trips. This makes the subproblem a shortest path problem (SPP). The idea of rewriting a mixed integer subproblem to a well-known problem is advised in most literature on the CG method. This makes it easier to find an algorithm to solve the problem properly, as there are many alternatives. The shortest path can be found with the help of a labelling method, which makes use of dynamic programming. A labelling method gives to every node labels, which contain the information of paths from the start node to the labeled node. This information often includes the distance of the path and the nodes or arcs that are visited in-between. The non-dominated labels are often stored, while dominated paths, if any, are removed. A dominated path is a path that can never obtain a better result than one of the non-dominated paths. These paths are not stored in order to reduce the running time of an algorithm. Dominated paths are often determined on the basis of a dominance rule, which varies per model. There are two labelling methods, namely label-setting algorithms and label-correcting algorithms. The label-setting algorithms are based on that each label is only set once. The most well known label-setting algorithm is the Dijkstra algorithm. In the label-correcting algorithms the labels can be visited and updated multiple times. An example of a such an algorithm is the Bellman-Ford algorithm.

The problem with electric buses is that the buses are depended on their SoC. A standard SPP algorithm does not work, since at some point the battery is empty and the bus needs to charge. Thus, the energy constrains have to be taken into account. The problem with charging makes it a resource constrained shortest path problem (RC-SPP). An overview and solutions of the RC-SPP's are given in [23].

The VSP problem does not revisit any nodes or edges in a solution due to that the graph is directed, therefore the problem is elementary. In other words one bus cannot drive the same trip twice. The problem is rewritten to an elementary resource constrained shortest path algorithm (ERC-SSP). This problem is solved in the paper [24] with a label-correcting algorithm, which is based on the Bellman-Ford algorithm.

A simplified problem of VDL is introduced to show how an adjusted label-correcting algorithm based on the one used in [16] can be used for the problem in this thesis. The example only includes scheduling trips, where the SoC of the bus cannot fall below a certain minimum SoC. The RMP is as follows:

$$\text{obj min} \quad \sum_{v \in V'} c_v u_v \tag{3.3}$$

$$\text{subject to} \quad \sum_{v \in V'} a_{tv} u_v = 1 \quad \forall t \in T \tag{3.4a}$$

$$0 \leq u_v \leq 1 \quad \forall v \in V', \tag{3.4b}$$

where c_v are the costs per vehicle task, a_{tv} are all ingoing arcs to t in vehicle task v, u_v is a decision variable and is equal to one, if the corresponding vehicle task v is present in the optimal solution. The matrix V' contains vehicle tasks and T are all the trips in the timetable. The objective function (3.3) minimizes the number of vehicle tasks, the constraint (3.4a) constraints that every trip has to be present in the final solution once and the lower and upper bound (3.4b) are respectively zero and one. The RMP can be written to the following dual problem:

$$\text{obj max} \quad \sum_{t \in T} \pi_t \tag{3.5}$$

$$\text{subject to} \quad \sum_{t \in T} A_{tv}^{T} \pi_t \leq c_v \quad \forall v \in V', \tag{3.6}$$

where π_t is the dual variable for each trip. The reduced cost is defined as:

$$c_v - \sum_{t \in T} \pi_t \delta_t, \tag{3.7}$$

where δ_t is a decision variable and is one if a trip is included in the new column. The subproblem has to find at least one column that makes the reduced cost (3.7) negative. The new column represents a new vehicle task. There is one constraint in this example which states that at every moment in time the SoC of the bus cannot fall below the minimum SoC.

The start node and end node of the label-correcting algorithm are the depot, since every bus has to start the day and end the day at the depot. The nodes in the graph consist of all the trips in the timetable plus the previously mentioned two depot nodes. Each node has a label stored that refers to paths starting from the start node to that specific node. These labels contain information on the reduced costs of the non-dominated paths to the node, the current SoC of the paths and the paths itself. The reduced costs replace the distance in a normal SPP algorithm. This means that the label-correcting algorithm is searching for a path with the lowest reduced costs. Taking an arc to trip t increases the reduced costs of the path by $-\delta_t \pi_t$ and reduces the SoC by e_t, which are the energy costs for driving trip t. The structure of the labels are is as follows:

$$\text{label} = \begin{bmatrix} \text{Reduced costs} \\ \text{SoC value} \\ \text{No. start node} \\ \vdots \\ \text{No. current node} \end{bmatrix}, \tag{3.8}$$

At the start node the SoC is 100% and the reduced costs are zero. At the start of the algorithm the other nodes have labels with a high positive reduced costs and a very low SoC. These labels are dominated by every path created from the start node. New labels are added to a node, when there is an ingoing arc from a different node. For example node j receives an ingoing arc from node i. Then all labels from node i are copied to node j and updated. Each label of i is updated by adding the dual variable from node j to the reduced costs, by reducing the SoC by the SoC costs of driving trip j and by adding the number of j to the path. For large problems the number of labels that need to be stored is huge, therefore often a dominance rule is used which reduces the number of labels. The problem is an ERC-SPP, where the recourse variable is in this case the SoC, which means that the dominance rule has to take into account two variables. The dominance rule is as follows: a path in node i is dominated by another path in node i, if the SoC is equal or lower and if the reduced costs is higher or equal. A path that is dominated can never obtain a better result, since the graph is directed.

The arcs in the graph represent the deadhead trips, which are neglected in this example. All arcs are stored in a list which is ordered on the start time of the arcs. So the arc with the lowest start time is first in the list. The arcs represent all possible choices that can be made to create the vehicle task. The list of arcs is to run through in the label-correcting algorithm in order to create the paths. Example 1 is given to explain how it works. Example 1 has a timetable with two trips:

Trip	Start time	End time	Energy consumption (SoC)
Trip 1	9:00	10:00	40
Trip 2	12:00	14:00	80

Table 3.1: *Timetable of Example 1*

The graph of Example 1 is shown in Figure 3.1, where the arcs are numbered according to the order in the list of arcs. The first arc is taken from the list. This is arc 1 and it connects the depot node 1 (dp1) to the trip node 1 (t1). A label is created in t1 and is as follows:

$$\text{label node t1} = \begin{bmatrix} c_v - \pi_1 \\ 60 \\ \text{dp1} \\ \text{t1} \end{bmatrix}. \tag{3.9}$$

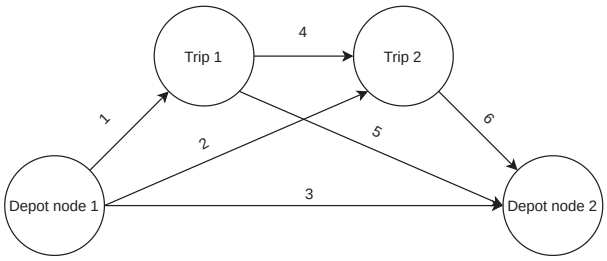


Figure 3.1: *The graph of Example 1*

This label dominates the starting label in node t1 and the inferior label is removed. Then the next arc is taken from the list, this is arc 2 and it leads to the following label:

$$\text{label node t1} = \begin{bmatrix} c_v - \pi_2 \\ 20 \\ \text{dp1} \\ \text{t2} \end{bmatrix}. \tag{3.10}$$

The new label dominates again the label that is already present in t2. The next arc is arc 3. This leads to the following label:

$$\text{label node dp2} = \begin{bmatrix} c_v \\ 100 \\ \text{dp1} \\ \text{dp2} \end{bmatrix}. \tag{3.11}$$

Then arc 4 is chosen. This leads to the following label:

$$\text{new label node t2} = \begin{bmatrix} c_v - \pi_2 - \pi_1 \\ -20 \\ \text{dp1} \\ \text{t1} \\ \text{t2} \end{bmatrix}. \tag{3.12}$$

The current SoC of the path is below the minimum SoC of 0. This means that this label is not feasible and is deleted. Arc 5 is taken next, this leads to:

$$\text{label 2 node dp2} = \begin{bmatrix} c_v - \pi_1 \\ 60 \\ \text{dp1} \\ \text{t1} \\ \text{dp2} \end{bmatrix}. \tag{3.13}$$

The label is not dominated in the case that $\pi_t > 0$ and does not dominate the previously mentioned label (3.11), since in one label the SoC is higher and in the other label the reduced costs are lower. This means that node dp2 has two labels. The last arc is arc 6. This gives the new label:

$$\text{label node dn2} = \begin{bmatrix} c_v - \pi_2 \\ 20 \\ \text{dp1} \\ \text{t2} \\ \text{dp2} \end{bmatrix}. \tag{3.14}$$

This label is dominated by the label in (3.13) in the case that $\pi_2 \leq \pi_1$, otherwise it is added to the two existing labels in node dp2. The new vehicle tasks can be chosen out of the labels in dp2. Every label with negative reduced costs can be transmitted to the RMP and added to matrix of vehicle tasks V. This example is only used to show how it works, but it is not used to show the efficiency of the algorithm. This can mainly be shown in larger models, where an early domination of a label can lead to a huge decrease in labels overall in the model and thus a huge reduction of the number of paths to the end node. The number of steps in this algorithm is equal to the number of arcs in the graph, which depends on the number of trips. The maximum number of steps is $T^2 - T$, while the maximum number of

steps in a MILP solver is 2^{T^2-T}. Note that this label-correcting algorithm gives an exact solution to the subproblem. A formulation for a scheduling subproblem for the label-correcting algorithm is as follows:

Algorithm 1: A label-correcting algorithm

$l_{dp1}(1) = 0$
$l_{dp1}(2) = SoC_{max}$
$l_{dp2}(1) = 10000$
$l_{dp2}(2) = 0$
for *all* t **do**
  $l_t(1) = 10000$
  $l_t(2) = 0$
end
for *all* $a_{ij} \in A_{list}$ **do**
  for *all labels in* L_i **do**
    for *all labels in* L_j **do**
      if $l_i(2) - e_j \geq min\_SoC$ **then**
        if $l_j(1) > l_i(1) - \pi_j$ *OR* $l_j(2) < l_i(2) - e_j$*;* **then**
          Add the newly updated label to L_j ;
        end
        if $l_j(1) \geq l_i(1) - \pi_j$ *AND* $l_j(2) \leq l_i(2) - e_j$*;* **then**
          Delete l_j;
        end
      end
    end
  end
end

L are all the labels stored at a node, e_t are the energy costs for every trip expressed in SoC, π_t are the dual variables per trip, A_{list} is the list of arcs in the graph.

Rewriting the graph and solving the ERC-SPP on the graph with a label-correcting algorithm seems to have potential. The problem can be reduced by a large amount by reducing the graph. For example: An arc can be removed from the graph, if the interval between two trips is large, since a bus is not allowed to be idle for a long period of time between trips. The problem is also reduced by the use of the dominance rule. That is why this method has been applied in the VDL tool. In the next chapter the implementation of the complete model is discussed.

3.5 Summary

In the first part of this chapter two lower bounds are created for determining the minimum number of buses required to make a schedule. A lower bound for fast chargers has also been created. This lower bound is conservative. Different algorithms have been written to solve the subproblem of Wijnheijmer [3] or the complete problem faster. None of the algorithms implemented has the potential to solve the electric vehicle scheduling problem (eVSP) of VDL. The greedy algorithm is not accurate enough, the genetic algorithm is too slow and barely improved the solution and the diving heuristic is too slow and infeasible solutions occur. A multiple step column generation method has been created, where multiple subproblems are placed in series. The method fails due to that correct transmission of dual variables between the subproblems is not possible. The last part of the chapter describes a method that uses a graph and rewrites the subproblem of an eVSP to a shortest path problem (SPP). This method looks promising, since there are algorithms available to solve the SPP and is implemented in the next chapter.

Chapter 4

Implementation of the model

In this chapter the implementation of the model for creating a schedule for a given timetable is discussed. The model is based on the column generation method, where the subproblem is rewritten to a shortest path problem (SPP). The column generation model has been discussed in Chapter 2 and the method for the subproblem has been discussed in Chapter 3. In this chapter the model is formulated. The master problem and restricted master problem (RMP) are given. The dual problem is given of the RMP to show which dual variables are used and to define a function to determine the reduced cost. The subproblem is rewritten to a SPP with the help of a graph. The creation of the graph is explained, as well as an explanation for the reduction of the number of arcs in the graph is given. The chapter concludes with the implementation of a label-correcting algorithm, which is an extension of the algorithm discussed in section 3.4. This label-correcting algorithm is used to solve the subproblem and thus to find the shortest path.

4.1 The Master problem

The Master problem has two different variables that are included in the objective function. These are the number of vehicle tasks and the number of chargers per location or per charger type. The use of one vehicle task is equal to the use of one bus. The costs for each vehicle task is the same and the costs for each charger location or charge type can change. For example the costs for slow chargers are zero, since there are always as many slow chargers as buses. The Master problem has two sets of constraints, where the first one assures that each trip has to be included at least once in the final solution and the second one is used to determine the number of chargers in the system. The number of chargers has to be at least as high as the chargers used in all chosen vehicle tasks per charging location. The lower bound for the decision variables of the vehicle tasks is zero and the upper bound is one. A vehicle task cannot be performed twice, since this is not useful. The lower bound for the number of chargers at a location is zero and the upper bound is equal to the total number of trips, since this is the upper bound on the total number of buses needed. The Master problem is formulated as follows:

$$\text{obj min} \quad (\sum_{v \in V} c_v u_v + \sum_{c_l \in C_l} c_{c_l} n_{c_l}) \tag{4.1}$$

$$\text{subject to} \quad \sum_{v \in V} a_{tv} u_v = 1 \quad \forall t \in T \tag{4.2}$$

$$-\sum_{v \in V} r_{p_{c_l} v} u_v + n_{c_l} \geq 0 \quad \forall p_{c_l} \in P_{c_l}, c_l \in C_l \tag{4.3}$$

$$\begin{aligned} u_v &\in \{0,1\} \quad \forall v \in V \\ n_{c_l} &\in \{0...n_t\} \quad \forall c_l \in C_l, \end{aligned} \tag{4.4}$$

where c_{cl} are the costs per charger for a location, n_{cl} are the number of chargers per location, n_t is the total number of trips in the schedule, a_{tv} are all the ingoing arcs to trip node t per vehicle, $r_{p_{cl}}$ is percentage of a charging session used, P_{cl} are all charging sessions at charger location c_l and C_l are all the charger locations.

The columns representing the vehicles tasks have the following structure:

$$v = \begin{bmatrix} a_{(cn_1,t_1)} \\ \vdots \\ a_{(cn_{end-1},t_{end})} \\ a_{(t_1,cn_2)} \\ \vdots \\ a_{(t_{end},cn)} \\ r_{pcl_1} \\ \vdots \\ r_{pcl_{end}} \end{bmatrix}, \tag{4.5}$$

where cn are the charger nodes, a are the arcs in the graph and r is the percentage used of a charging session. The vehicle task includes all the arcs present in the graph and it includes for every charging session a variable r, which can only be zero or one in the MP, because n_{c_l} is an integer variable. To solve the MP all possible vehicle tasks v are enumerated. The number of vehicle tasks to be enumerated is huge. The idea behind CG is that not every vehicle task is used, but only a small subset of all vehicle tasks is used to find the optimal solution. To accomplish this the RMP is introduced in the next section.

4.2 The Restricted Master problem

The RMP is derived from the MP. An inequality constraint is used for constraint (4.7), because it is easier to find an integer solution from the fractional RMP solution. An in-depth explanation is given in subsection 5.1.2. The upper and lower bound of variables n_{c_l} and u_v are relaxed. The variables need to be relaxed in order to perform a sensitivity analysis with help of the dual variables. The RMP is as follows:

$$\text{obj min} \quad (\sum_{v \in V'} c_v u_v + \sum_{c_l \in C_l} c_{c_l} n_{c_l}) \tag{4.6}$$

$$\text{subject to} \quad \sum_{v \in V'} a_{tv} u_v \geq 1 \quad \forall t \in T \tag{4.7}$$

$$-\sum_{v \in V'} r_{p_{c_l} v} u_v + n_{c_l} \geq 0 \quad \forall p_{c_l} \in P_{c_l}, c_l \in C_l \tag{4.8}$$

$$\begin{aligned} 0 \leq u_v \leq 1 \quad & \forall v \in V' \\ 0 \leq n_{c_l} \leq n_t \quad & \forall c_l \in C_l, \end{aligned} \tag{4.9}$$

where V' is a subset of V and is a matrix with all vehicle task currently in the RMP.

4.3 The dual of the Restricted Master problem

The dual of the RMP is needed to determine the dual variables which can be used to determine the reduced costs of each column. The dual of the RMP is as follows:

$$\text{obj max} \quad \sum_{t \in T} \pi_t \tag{4.10}$$

$$\text{subject to} \quad \sum_{t \in T} A^T_{tv} \pi_t - \sum_{p_{c_l} \in P_{c_l}, c_l \in C_l} R^T_{p_{c_l} v} \pi_{p_{c_l}} \leq c_v \quad \forall v \in V' \tag{4.11}$$

$$\sum_{p_{c_l} \in P_{c_l}, c_l \in C_l} \pi_{p_{c_l}} \leq c_{c_l} \quad c_l \in C_l \tag{4.12}$$

$$\pi_t \geq 0 \quad \forall t \in T \tag{4.13}$$

$$\pi_{p_{c_l}} \geq 0 \quad \forall p_{c_l} \in P_{c_l}, c_l \in C_l. \tag{4.14}$$

There are two sorts of dual variables. The first one is π_t and is associated with constraint (4.7). The dual variable π_t gives the value of the increase in objective function, if the constraint for trip t in the set

of constraints (4.7) is changed such that trip t has to be driven at least twice. The second dual variable $\pi_{p_{c_l}}$ can be coupled to the variable r_{pcl}, which determines whether a trip is included in the new column or not. The dual variable is coupled to constraint (4.8) and gives the increase in objective function, if one extra charger at charger session p at charger location c_l is used. The equation to calculate the reduced costs of this problem is as follows:

$$c_v - \sum_{t \in T} \pi_t \delta_t + \sum_{p_{c_l} \in P_{c_l}, c_l \in C_l} \pi_{p_{c_l}} r_{pcl}, \tag{4.15}$$

where δ_t is one, if trip t is included in a column. The RMP is a minimization problem, therefore the reduced cost has to be negative. The reduced cost (4.15) are used as an objective function in the subproblem. In the case that the subproblem finds a column, which has negative reduced costs, the column is added to V'. The RMP is solved until optimality, in the case that only positive columns can be found.

4.4 The subproblem

The RMP has been determined in the previous section and the equation for the reduced cost (4.15) has been given. The objective of the subproblem is to find a column (i.e., vehicle tasks) that has negative reduced costs. The subproblem consists of two parts: The graph and the label-correcting algorithm to solve the graph. In general, the graph describes the subproblem, whereas the label-correcting algorithm solves the subproblem. However, the label-correcting algorithm also enforces some constraints. In this section first the creation of the graph is discussed and then the label-correcting algorithm is discussed.

4.4.1 Creation and reduction of the graph

This section explains how a subproblem can be created in such a way that it becomes a shortest path problem. A graph is created that takes the dual variables into account.

In the previous chapter a simple example is given on creating a graph for a timetable with trips. There are many more requirements for this subproblem model, which have to be taken into account when making the graph. The requirements relevant for creating a graph are the possibility of charging, the implementation of multiple charge locations/types, the inclusion of deadhead trips in the model and a maximum idle time at every location, except for the depot (see section 1.2). In this section the creation of the graph is explained by an example which is called Example 2. Two tables are given, the first one states the information of the trips that need to be driven and the second one states the duration of the deadhead trips.

Trip	Start location	End location	Start time	End time	Energy consumption (SoC)
Trip 1	A	A	9:00	10:00	50
Trip 2	A	A	10:00	11:00	20
Trip 3	B	A	10:30	11:00	20
Trip 4	A	A	11:00	12:00	30
Trip 5	A	A	12:00	13:00	30
Trip 6	A	A	14:00	15:00	20

Table 4.1: *Timetable Example 2*

Time (min)	Location A	Location B	Location Depot	Location Charger 1
Location A	0	30	0	30
Location B	30	0	30	0
Location Depot	0	30	0	30
Location Charger 1	30	0	30	0

Table 4.2: *Duration of deadhead trips Example 2*

First, the implementation of charging at the depot is discussed. Charger location 1 is ignored. Charging on the depot is implemented by creating for each trip two depot nodes. The first one is equal to the start

of the trip minus the time of the deadhead trip and the second one is equal to the end time of the trip plus the time of the deadhead trip. This means that a bus is not idle at any moment between going from a charger node to a trip node. The graph is shown for the timetable in Table 4.1 with only the depot nodes and the trip nodes in Figure 4.1. In Figure 4.1 the trip nodes represent every trip in Table 4.1.

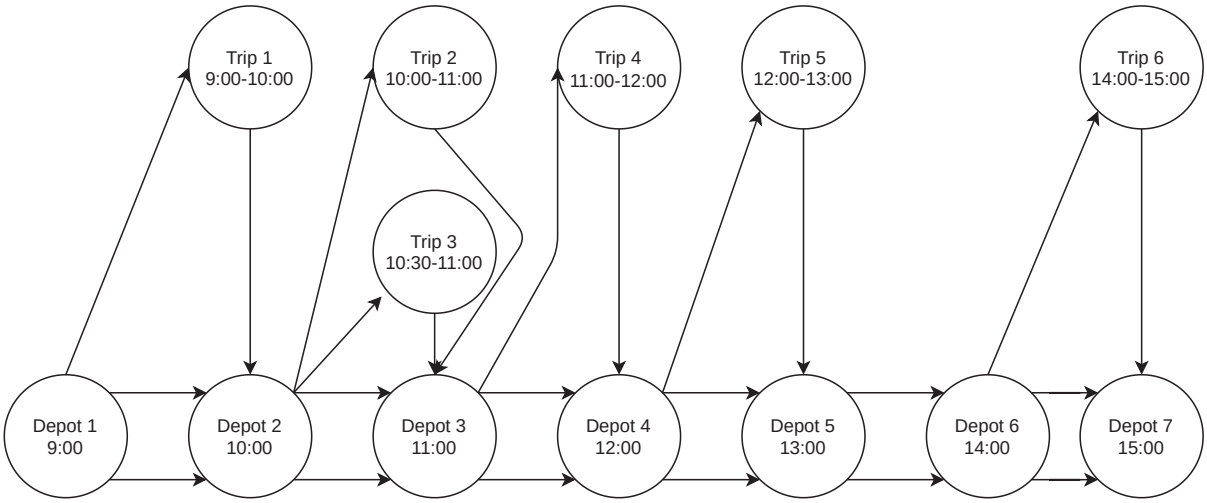


Figure 4.1: *Graph with depot nodes and trip nodes of Example 2*

For each trip node two depot nodes are created, where one of the depot nodes is deleted, if there are two depot nodes with the same time point. The arcs between the trip nodes and the depot nodes represent the deadhead trips. For example the arc from depot node 2 to trip 3 takes 30 minutes to drive. The pair of arcs between the depot nodes do not represent deadhead trips. On the depot a bus can be idle or use a slow charger. These two actions are represented in the graph by two having arcs between the depot nodes. One of the arcs between the depot nodes represents a charging session and the other arcs represent an option for a bus to be idle. A charging session takes as long as the time interval between the pair of depot nodes. For example the charging session between depot node 1 and depot node 2 takes one hour. The first depot node is called the start node of the model, since all buses start in the beginning of the day at the depot. The last depot node is called the end node, since all buses have to end at the depot.

Charger location 1 also has to be implemented in the graph. Fast chargers are used at charger location 1. Charger nodes are placed for the start time and end time for each trip node, just as has been done for the depot nodes. The time of the deadhead trips to and from charger location 1 are included. There are two differences between depot nodes and charger nodes. The first difference is that it is not allowed for a bus to be idle on a charger location. This means that there is only one arc between two charger nodes, which represents a charging session. The second difference is that all charging nodes that are placed before the start node of the first depot node and all charging nodes that are placed after the end node are not implemented in the graph. These nodes are deleted, because they can never be reached by a bus that starts and ends the day at the depot. Figure 4.2 shows the implementation of the charger nodes of Example 2.

In some cases there are two types of chargers at a charger location. Each type is seen as a different charger location (with the same deadhead trips to other nodes), where for each location a new line of charger nodes is created. An important thing to note is that there are no arcs between charger nodes from different locations or charger types. This can lead to a suboptimal solution of the subproblem. Fast chargers on the depot are treated as a charger location, which means that a bus cannot be idle at the depot after using a fast charger on the depot. To solve this problem an option is implemented to choose between the use of one or two arcs in-between two nodes in the case that there are two different chargers at the depot. This option is by default off in the model, since it increases the computational time.

The second part of this section discusses the creation of arcs between trip nodes. One of the most important parts of this method is to keep the graph as small as possible, while keeping the optimality gap between the solution and the optimal solution of the subproblem as small as possible. The maximum number of arcs between all trips is $T^2 - T$. The graph is too large for Timetable 7 (1096 trips) in the

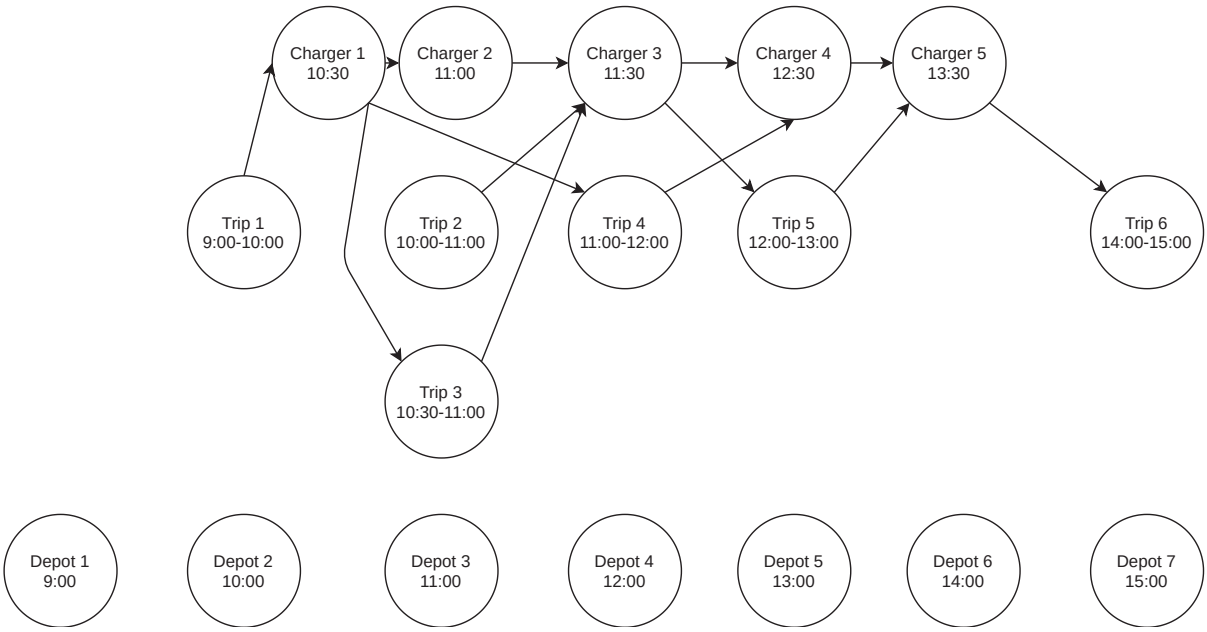


Figure 4.2: *Graph with depot nodes, trip nodes and charger nodes of Example 2*

case that all arcs are created between every combination of two trips. Therefore, not all trips are directly connected with each other by arcs. The graph of Example 2, where all trips are connected with each other is shown in Figure 4.3. All arcs that lead to time incompatibility (i.e., simultaneously driven trips)

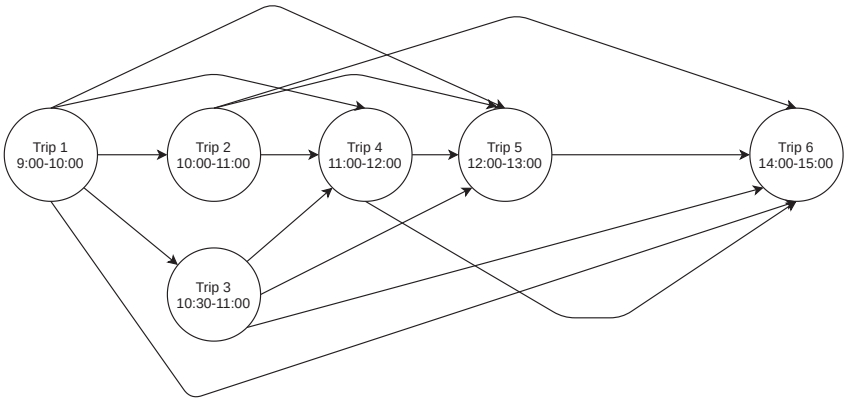


Figure 4.3: *Graph with arcs between all trips nodes of Example 2*

between two trips are already deleted in the graph. There are two rules applied to delete additional trip to trip arcs from Figure 4.3. The first rule states that the idle time between two trips cannot exceed a certain threshold, which is expressed in minutes, i.e., the bus is not allowed to stand still for longer than a certain period of time. This is also a requirement of the sales department of VDL Bus & Coach (see section 1.2).

The second rule states that a deadhead trip to a trip can only take a certain amount of minutes. The reason why this rule is used is because in general it is inefficient if a bus ends a trip in the north of the city and then starts a next trip in the south of the city, because this costs time and energy. It is unlikely that a vehicle task with long deadhead trips is used in the optimal solution for the subproblem. In some cases this rule can lead to a suboptimal solution, but it is expected that this rarely happens. In this example it is not allowed to be idle for longer than 10 minutes and deadhead trips cannot take longer than 20 minutes. This leads to the graph in Figure 4.4. The number of arcs are reduced from 14 to 4. The final graph is shown in Figure 4.5.

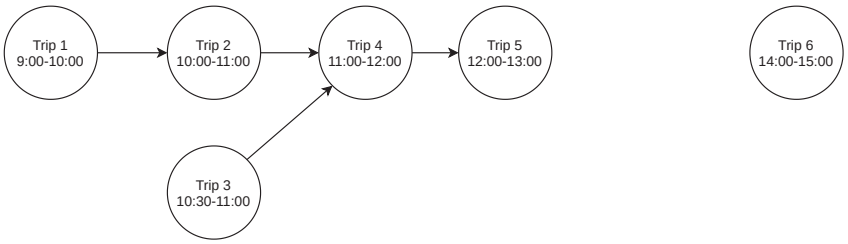


Figure 4.4: *The connection between the trip nodes of the reduced graph of Example 2*

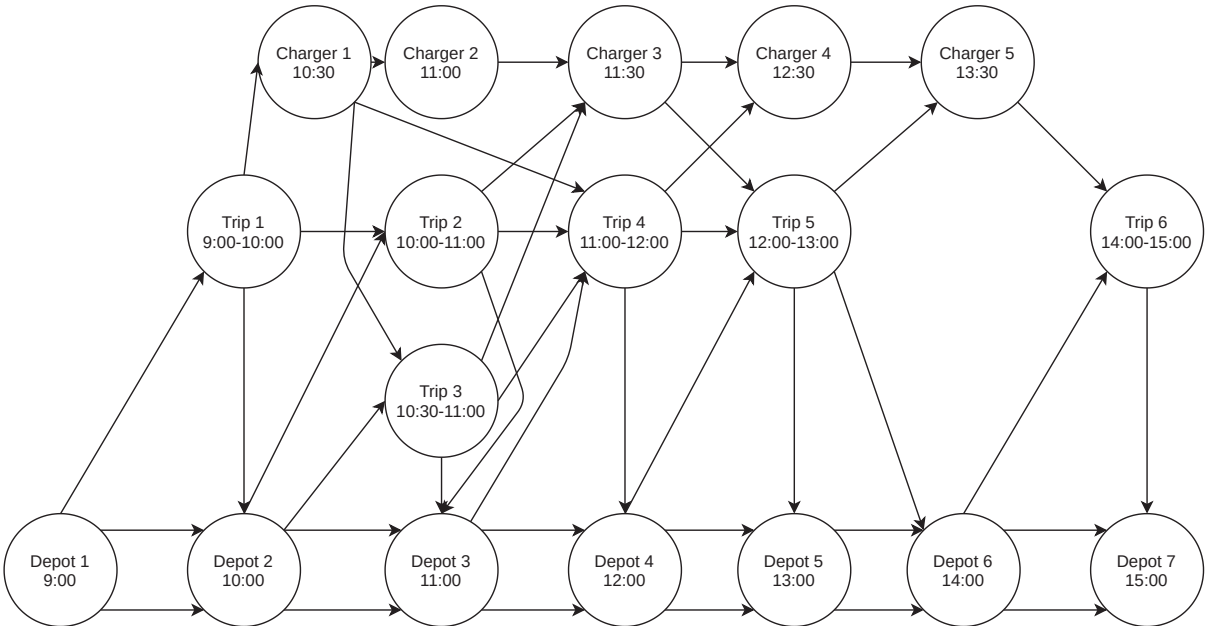


Figure 4.5: *The final graph of Example 2*

In this example no limit is set on the drive time of deadhead trips to charger location 1. In the model it is possible to also set a limit on these deadhead trips. This also reduce the graph of large models a lot. The motivation for implementing this option is the same as for the maximum drive time of deadhead trips between two trips. It is advised that for each schedule every trip is directly connected by an arc to at least one fast charger location. Slow chargers (i.e., the depot nodes) are always reachable from trip nodes. There are no limits on the time of the deadhead trips to the depot, since the depot always has to be reachable, because a bus has to be able to stand still at the depot. In one case it is still possible to drive long deadhead trips, namely by going from a trip node to a depot node and then immediately going to a new trip node.

4.4.2 Implementation of a label-correcting algorithm

On the graph a shortest path has to be found each time the RMP is solved. The subproblem is an ERC-SPP which has been discussed in section 3.4. The section explains that such a problem can be solved with a label-correcting algorithm which is beneficial as it can exclude a lot of paths by deleting dominated paths early on. In this section the algorithm is extended with some requirements of VDL (1.2) and is adjusted to the changes in the graph which are not yet taken into account in section 3.4. These are the inclusion of deadhead trips, double arcs between depot nodes, multiple charger locations, minimum shift time and minimum charge time. In this section first the influence of the changes in the graph on the algorithm is explained and afterwards the implementation of minimum charge time and minimum shift time is discussed. The implementation of the charger connection time and a non-linear charge rate curve are discussed in Appendix C.

4.4.2.1 Inclusion of the charging nodes

There are five different types of arcs in a graph. The differences between these arcs have to be taken into account in the label-correcting algorithm and each one has to be treated differently. These five types of arcs are the arc from trip to trip, the arc from trip to charger, the arc from charger to trip, the arc from charger to charger and the arc from depot to depot, where in the latter one the bus is idle. Each type of arc has a different set of constraints that have to be met and updates the label of a path differently. The inclusion of the different arcs is explained on the basis of the following overview:

Figure 4.6: *Overview of the label-correcting algorithm with multiple arc types*

Step 1,2,3,7,8 and 9 are exactly the same as in section 3.4. In step 1 all arcs from the graph are ordered based on the starting time of each arc and are placed in a list. The starting node label is created which has a maximum SoC and the other labels have labels that can be dominated by any path. Step 2 picks the first arc from the list. The main difference is in step 3.

In step 3 the paths of the starting node of the arc are updated depending on the arc type. Each arc has a different influence on the labels of the paths. The charger to charger arc updates the reduced costs with the dual variable $\pi_{p_{cl}}$, since the arc represents a charging session. The duration of the charging arc $dt_{p_{cl}-p-1_{cl}}$ times the charging rate of the charger e_{charge} times the percentage of the charging arc r_{pcl} used determines how much SoC is added to the current SoC of the path. The variable r_{pcl} is always one, since the algorithm can only choose to use a charging session or not. Each $\pi_{p_{cl}}$ which is equal to zero and linked to a fast charger is given a very small positive value to prevent unnecessary charging. The reduced costs and the SoC of the paths are not updated, when taking a depot to depot arc, which is the arc where the bus is idle for the duration of the arc. The reduced costs stay the same for a path,

when the paths follow the trip to charger arc. The SoC decreases with the SoC costs for taking the deadhead trip from the end of trip to charger location $e_{h_{tcl}}$. The reduced costs is reduced with π_t, when a path takes the charger to trip arc. The SoC decreases with the SoC costs of trip e_t and the SoC costs of taking the deadhead trip from the charger location to the start point of the trip $e_{h_{clt}}$. Taking the trip to trip arc reduces the reduced costs of a path with π_t and decreases the SoC of a path with e_t. An overview of the influence of taking an arc type on the variables in the labels of a path is shown in Table 4.3.

The updated paths are checked for meeting the SoC requirements after step 3. The second difference to the model in section 3.4 is that the paths are checked on different SoC constraints depending on which arc is taken. For the charger to charger arcs, the new updated paths are checked for whether the maximum SoC is exceeded. The percentage of the charging time r_{pcl} used is decreased to meet the constraint, when the maximum SoC is exceeded. The new updated paths are checked for whether the SoC drops below minimum SoC for when the arc is a charger to trip arc, a trip to charger or a trip to trip arc.

It cannot be guaranteed that the label-correcting algorithm always obtains an optimal solution, because of the maximum SoC constraint of a charger to charger arc. It can happen that a path is dominated at a charger node by a path with a maximum SoC value, which cannot take an arc to the next charger node, since the battery is already full and thus the bus cannot be charged. The dominated path with a lower SoC can take the charger arc which leads to a new charger node. It is possible that from this charger node a trip can be reached, which the path with the maximum SoC cannot reach. This situation is however not likely to often occur, because the battery is not often full. A second reason that the solution is not always optimal is that $r_{p_{cl}}$ is always one. A large time interval between two charger nodes leads to a long charging session. Sometimes a vehicle task does not need to charge the entire charging session. It might be better to split this charging session between two buses, which is not possible at the moment. This can be implemented by altering the vehicle task, after the label-correcting algorithm has created it.

Step 5 is again the same as in section 3.4. In this step the updated paths and the old paths in the end node of the arc are compared with each other with the help of the dominance rule. All dominated paths are deleted and all non-dominated paths are added in the form of labels to the end node of the arc. Then the arc is deleted in step 8. If the list is not empty, a new arc is picked. When the list is empty, the simulation ends and the shortest paths can be found in the last depot node.

4.4.2.2 Extension with minimum charge time and minimum shift time

The minimum charge time needs to be implemented to prevent short charging sessions of one minute, which are not realistic to be implemented. The minimum shift time is needed to prevent that a bus has to drive only one trip when leaving the depot, because sometimes customers do not accept such a solution. The minimum charge time and minimum shift time can be enforced by changing the constraints after step 3 in the overview in Figure 4.6. Two new variables are implemented in the labels of the paths which are the current charge time and the current shift time. On a side note, on the contrary to section 3.4 the arcs are now stored instead of the nodes, since these are used to describe a vehicle task. The new structure of a label of each path looks as follows:

$$label = \begin{bmatrix} \text{Reduced costs} \\ \text{SoC value} \\ \text{Current shift time} \\ \text{Current charge time} \\ \text{No. starting arc} \\ \vdots \\ \text{No. current arc.} \end{bmatrix} \tag{4.16}$$

Introducing these new variables means that these variables have to be updated in step 3 of Figure 4.6. The current charge time and the current shift time are influenced differently per arc type. The current charge time is increased with the time between two charge nodes $dt_{pcl-p-1cl}$, when the charger to charger arc is taken. It is reset to zero, when a charger to trip arc is taken or a depot to depot arc is taken. The current shift time is increased by the duration of a trip, when the charger to trip arc is taken or the trip to trip arc. It is reset, when the trip to charger arc is taken. An overview is given in Table 4.3.

Type of arc	Event	Change in reduced costs	Change in SoC [%]	Change in charge time [min]	Change in shift time [min]
Charger to charger arc	Charging session	$\pi_{p_{cl}}$	$r_{pcl}e_{charge}dt_{pcl-p-1cl}$	$dt_{pcl-p-1cl}$	0
Depot to Depot arc	Stand still at the depot	0	0	0	0
Trip to charger arc	Deadhead trip	0	$-e_{h_{tcl}}$	0	0
Charger to trip arc	Deadhead trip + trip	$-\pi_t$	$-e_t - e_{h_{clt}}$	0	$h_{t_{end}} - h_{t_{start}}$
Trip to trip arc	Deadhead trip + trip	$-\pi_t$	$-e_t - e_{h_{t-1t}}$	0	$h_{t_{end}} - h_{t_{start}}$

Table 4.3: *Overview of the influence of taking arc types on the labels*

The parameter $h_{t_{end}}$ is the end time of a trip, $h_{t_{start}}$ is the start time of a trip, e_h are the energy costs for each deadhead trip, e_t are the energy costs per trip and e_{charge} is the charge rate.

As previously mentioned, the minimum shift time and the minimum charge time can be enforced with constraints. There are extra constraints for the trip to charger arc, the charger to trip arc and the depot to depot arc. A trip to charger arc can only be taken, if the updated path has a current shift time which is larger than the minimum shift time. A charger to trip arc can only be taken, if the updated path has a current charge time which is larger than the minimum charge time. The depot to depot arc can only be taken, when the current charge time of the updated path is zero or the current charge time is larger than the minimum current charge time. These constraints enforce that a path always has long enough charging sessions and long enough shifts.

The dominance rule also needs to be changed, when enforcing the minimum charge and shift time. For example: Currently, there are two paths at the same charger node. One path meets the minimum charge time requirement and one path does not meet the minimum charge time requirement. The path that does meet the requirement can take charger to trip arcs, while the path that does not meet the requirement can only take the charger to charger arc. This means that it is not known if the latter path is for certain always a better path than the path that does meet the time requirements, even if it has lower reduced costs and a higher SoC. The path that meets the requirement has the option to maybe drive a trip with a very positive dual variable π_t, which leads in the end to the path with the lowest reduced costs. It does however not mean that the path with a longer current shift time or charge time cannot be dominated. The current charge or shift time is irrelevant, if both paths exceed the minimum charge or shift time. The addition of the two new variables leads to four different situations:

1. The current charge time of one of the paths is shorter than the minimum charge time. Then the current charge time is taken into account in the dominance rule.
2. The current charge time of both paths is longer than the minimum charge time. Then the current charge time is not taken into account in the dominance rule.
3. The current shift time of one of the paths is shorter than the minimum shift time. Then the current shift time is taken into account in the dominance rule.
4. The current shift time of both paths is longer than the minimum shift time. Then the shift time is not taken into account in the dominance rule.

Taking these situations into account the dominance rule is updated as follows: In the same node a path i is dominated by another path j if the following conditions are true:

1. The reduced costs of path i are higher or equal to the reduced costs of path j.
2. The SoC of path i is lower or equal to the SoC of path j.

3. The current charge time of path i is non-zero and shorter than the minimum charge time and the current charge time of path j.
4. The current shift time of path i is shorter than the minimum shift time and the current shift time of path j.

The minimum charge time and the minimum shift time are now fully implemented in the model.

4.5 Summary

In this chapter the master problem (MP) is formulated, which minimizes the number of vehicle tasks and the number of chargers. The columns are the vehicle tasks, which describe the route a bus drives during the day and describes when and where the bus has to charge. The restricted master problem (RMP) and the subproblem are derived from the MP. The subproblem is written as a shortest path problem (SPP). This is done by creating a graph, which takes into account the trips, the deadhead trips, the different charger locations, charging sessions and the place where buses can have a break. The shortest path is found with the help of a label-correcting algorithm, which takes the reduced costs and the SoC of a bus into account. The algorithm also enforces a minimum shift time and a minimum charge time for each path.

Chapter 5

Obtaining an integer solution and accelerating the model

In Chapter 4 a column generation (CG) model has been described to solve the electric vehicle scheduling problem of VDL. The model only gives the optimal fractional result, which is because the lower bound and upper bound of the variables have been relaxed in the RMP (see section 4.2). An integer solution has to be found for the RMP without relaxed constraints. This has been done with a diving heuristic, which has been described in section 2.2.2. The implementation and the problems of implementing this diving heuristic are described in this chapter.

It is not trivial to implement the model of Chapter 4 in MATLAB. The main problem for implementing this model is the computational time. For example: The implementation of constraints in the model can have a huge influence on the computational time. In the second part of this chapter the essentials to create a well working column generation method are described. These are: The choice of the LP solver, the implementation of the constraints, the warm starts, column management and a method to cut-off the column generation method.

5.1 Finding an integer solution

The column generation method described in Chapter 4 is only suited to find an optimal fractional solution of the RMP. The number of vehicle tasks and the number of chargers used have to be expressed as an integer value. There are multiple methods to accomplish this, which has been discussed in subsection 2.2.2. The branch-and-price algorithm has not been implemented, since it is expected that the computational time is too high for practical use. A primal heuristic is chosen and specifically a diving heuristic. The reason for this choice is that it is expected that a diving heuristic gives a relatively good solution in a short computational time. A diving heuristic is a heuristic that chooses only one option out of all options at each node it passes in a branch-and-bound tree in order to obtain one solution in the branch-and-bound tree in a fast way. Thus, it creates one path from the top of the branch-and-bound tree to the bottom of the tree. The implemented diving heuristic is explained in-depth in the first subsection. In the second subsection it is discussed why trips are allowed to be driven twice in a schedule instead of only once and why it is difficult to limit the number of chargers at a location.

5.1.1 The rounding algorithm based on a diving heuristic

The diving heuristic consists of two parts. The first part consists of the standard diving heuristic and in the second part this heuristic is implemented into the CG model. An overview is given in Figure 5.1 to explain the standard diving heuristic used, after the CG model has found an optimal fractional solution for the RMP. The decision variables u_v from the fractional solution that are linked to the vehicle tasks are rounded up one at a time and fixed to one by the diving heuristic. The corresponding vehicle tasks are used in the final solution. The variables of the number of chargers per charger location n_{c_l} are not rounded with the diving heuristic, but are rounded when the final solution of vehicle tasks is completed.

The diving heuristic starts with checking if there is already a (non-fixed and non-zero) integer vari-

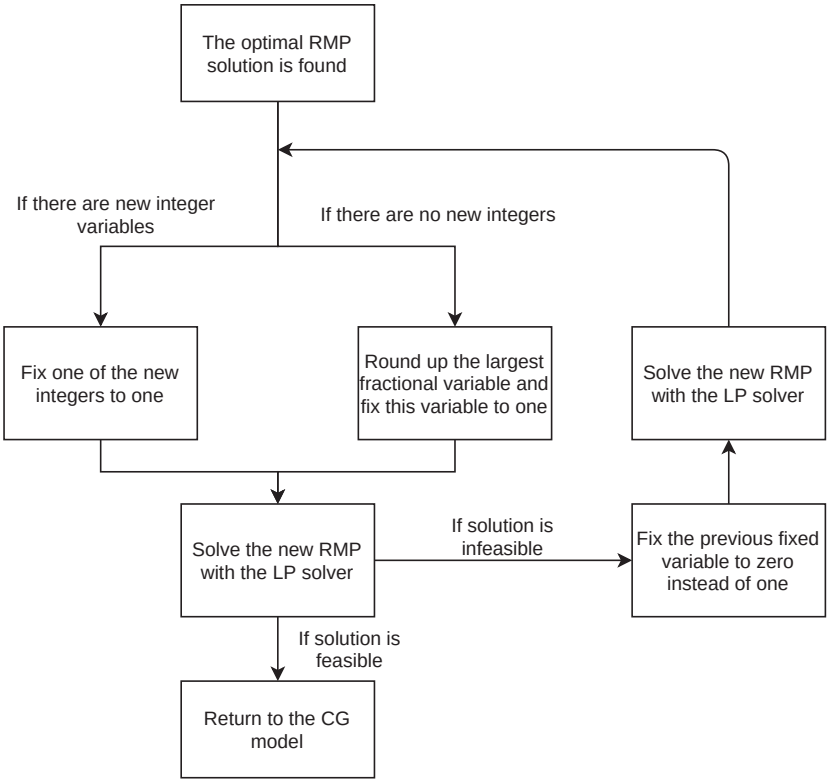


Figure 5.1: *Overview of the diving heuristic*

able present in the fractional RMP solution, when this is the case the variable is fixed to one. In case that there are multiple integer variables only one of them is fixed. Simulations have shown that fixing variables one by one can lead to a better result. The variable with the largest fractional result is fixed to one, when there are no (non-fixed) integer variables available. The variables are temporary fixed by adding equality constraints to the RMP in the diving heuristic. The new RMP is tested on feasibility in the diving heuristic. The RMP is solved by an LP solver, which shows if the current RMP is solvable. In the case that the RMP is not solvable, the previously rounded variable is set to zero. This leads again to a change in the RMP and the RMP has to be solved again to identify the next variable that has to be rounded up. Once the RMP is feasible, the first part of the diving heuristic is complete. Then the second part of the diving heuristic needs to be used to implement the rounded variables in the CG model.

This test for feasibility is implemented, because early on in the creation of the diving heuristic the constraint (4.7) has been an equality constraint, which means that trips can only be assigned once. A higher bound has also been set on the number of chargers at each charger location by changing the set of constraints (4.8). This is not the case anymore in the current RMP. The reason for the change in constraints is discussed in the second section 5.1.2. In the current implemented RMP the rounded up decision variables u_v are always feasible.

At this point it is known how to determine which variable has to be rounded up, but it is not yet discussed how the new value of the rounded up variable can be fixed in the RMP of the column generation model. The second part of the rounding model is used to implement the result of the diving heuristic in the CG model. Variables cannot be constrained by adding equality constraints to the RMP that constrain the rounded variables to one. The reason for this is that it influences the dual variables and with this the objective function of the subproblem incorrectly. The following is an example of a problem that can occur: At some point the fixed integers provide a worse result than the non-integer solution in most models. It can happen that the columns with the lowest reduced costs are columns with trips that are already fixed, because the combination of fixed variables are not present in the best non-integer solution of the RMP. The rounded variable has to be fixed in another way to prevent this. In [18] it is suggested to change the right-hand side of the constraints in the RMP, which are in other words the b values in the $Ax \leq b$ structure for constraints. This can also be used in this CG model. The

vehicle task associated with the new fixed variable is stored in the final solution matrix V_{final}, when the associated variable is rounded up. The trips that are assigned to this vehicle task are subtracted from the right-hand side of the associated constraints (4.7). This leads to the following constraint:

$$\sum_{v \in V'} a_{tv} u_v \geq 0 \quad \forall t \in T_{final}, \tag{5.1}$$

where T_{final} is a set of trips that are currently present in the final solution. In the new RMP these trips do not have to be driven. The previously created columns that have these trips can however be present in the new solution of the RMP. The trip variables present in the final solution can also completely be deleted from the problem. The constraints (5.1) have to be deleted from the problem. This means that also all vehicle tasks that contain such a trip have to be deleted. The advantage of this is that there are no trips driven twice in the schedule. The disadvantage is that after every rounding iteration many columns have to be recreated. Deleting columns can also lead to infeasibility due to a still to be assigned trip not being present in any of the remaining columns. Therefore, an initial solution that contains every left over trip has to be added each rounding iteration to prevent an infeasible RMP. The option is implemented in the MATLAB script, but is disabled by default. The reason for this is that the computational time is long for large timetable, because a lot more columns have to be created. Another option is implemented, which is that the trips that are already present in the final solution are forbidden to be in the solution of the subproblem. Thus, the arcs that lead from and to these trips are deleted from the list of the label-correcting algorithm. This leads to a decrease in computational time and a decrease of trips driven twice in a schedule.

The constraint for the chargers also needs to be updated, after a variable u_v has been rounded up. The chargers used per charging session per charging location are counted in the newly fixed vehicle task and are added to the right-hand side of the constraints in (4.8). The updated constraints look like this:

$$-\sum_{v \in V'} r_{p_{c_l} v} u_v + n_{c_l} \geq n_{p_{cl\,final}} \quad \forall p_{c_l} \in P_{c_l}, c_l \in C_l. \tag{5.2}$$

An overview of the second part of the rounding algorithm can be found in Figure 5.2. The second part of the diving heuristic also determines when the simulation is complete, which is when all decision variables u_v are integer. This is the case when all constraints of (4.7) are transformed to the constraints (5.1), i.e., all trips are assigned to the buses. The number of chargers used are still fractional. These are rounded upwards at the end of the complete simulation.

5.1.2 Explanation of the disadvantages of the rounding algorithm

The current diving heuristic gives an integer solution in a relatively short time. There are however a few disadvantages of using the diving heuristic of Figure 5.1 as a rounding algorithm. These disadvantages are discussed in this subsection.

The first disadvantage is that an inequality constraint has to be used for constraints (4.7) instead of an equality constraint. The latter one is preferred, since driving trips twice is then impossible. Having an equality constraint has led too often to an infeasible RMP, when rounding variables up in the diving heuristic. An infeasible RMP can occur, because a trip can only be driven in a combination of different vehicle tasks. For example, there are three trips and two vehicle tasks. The first vehicle task drives the first two trips and the second vehicle task drives the second and third trip. An infeasible RMP occurs, if the first vehicle task is rounded up, because the third trip cannot be driven. As discussed in the previous subsection, a method is built in to spot and fix variables that are rounded up and make the RMP infeasible. The variable that leads to infeasibility is set to zero and the next largest fractional variable gets chosen, until a feasible solution is obtained. Fixing a variable to zero creates a new RMP, which has to be solved to optimality again. It is however difficult to fix a variable to zero in the RMP due to the way the label-correcting algorithm works, because it cannot forbid to create one specific vehicle task. It has been decided to not solve the new RMP to optimality, but only check the new RMP on feasibility with an LP solver. Then the new largest fractional variable is chosen. This can lead to poor results, especially in the case that a couple of infeasible rounded up variables occur in a row. To overcome this problem it is chosen to use an inequality sign as in equation (4.7). The advantage of this is that no infeasible RMP can occur. The disadvantage is that trips can be driven twice, which means that a trip

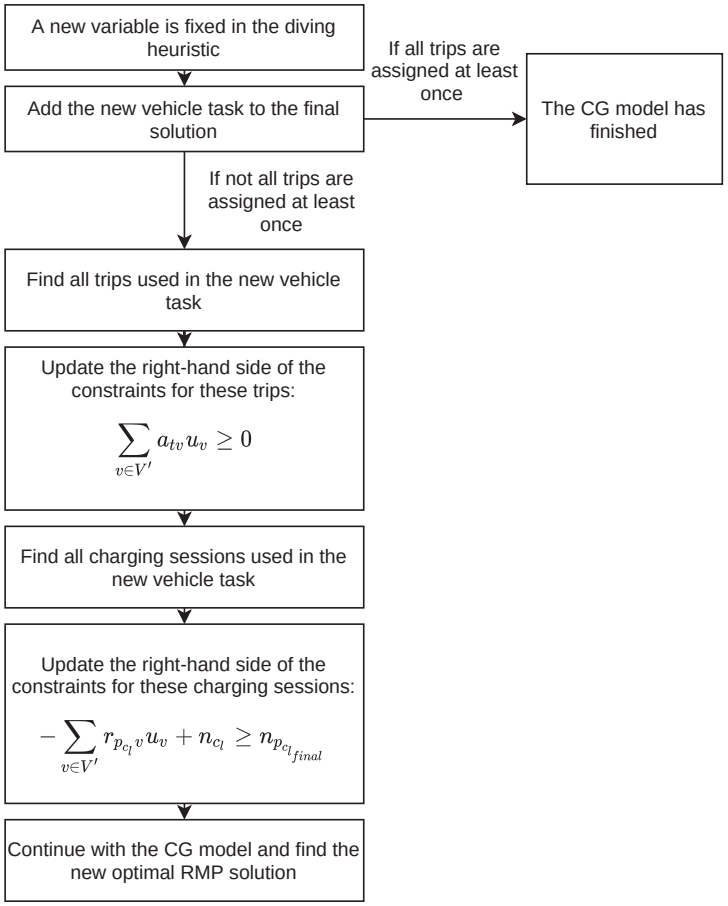


Figure 5.2: *Overview of implementing the results of the diving heuristic in the CG model*

can be assigned in the final solution to two vehicle tasks. The double trips can be seen as a deadhead trip in the case that a trip is assigned twice. Driving a trip double in a schedule costs extra energy and time, therefore it has been expected that few trips are driven double.

The same problem as for the equality constraint for the trips occur, when a low upper bound is placed on the charger per location. Sometimes it can be of use for VDL to limit the number of chargers used at a location. This can be implemented by changing the constraint (4.8) to the following constraint:

$$-\sum_{v \in V'} r_{p_{c_l} v} u_v \leq n_{p_{c_l}} \quad \forall p_{c_l} \in P_{c_l}, c_l \in C_l, \tag{5.3}$$

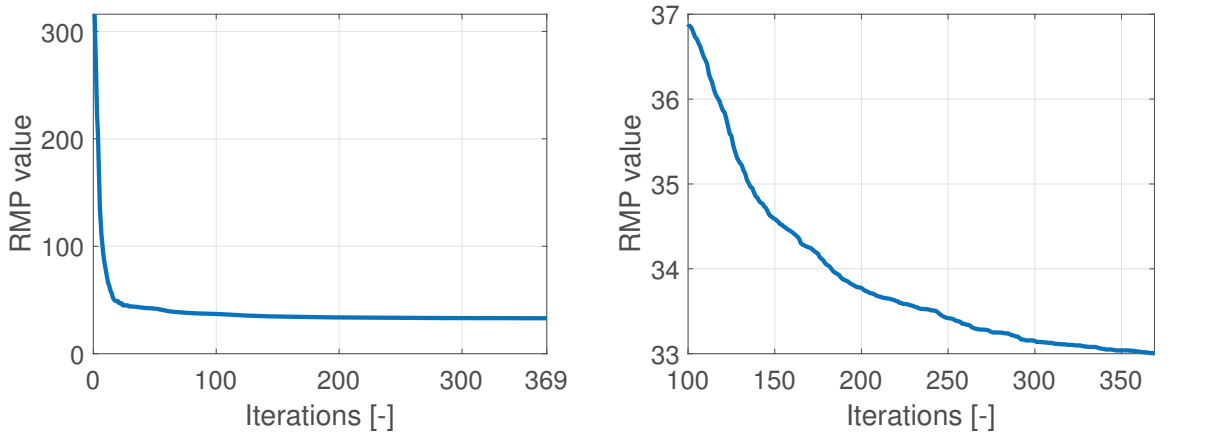
where $n_{p_{cl}}$ is the number of chargers per charging session per charger location. It has an upper bound n_s, which is the maximum number of chargers available for charger location c_l. The constraint in (5.3) is deleted, when n_s is reached in the interim final solution, which consists of the vehicle tasks which decision variables have been set to one. The variable $n_{p_{c_l}}$ that reaches the limit of n_s is removed from the RMP. The problem is again that it can lead to infeasible solutions during the process of the diving heuristic shown in Figure 5.1. This happens, when there are relatively few chargers allowed compared to the trips that have to be driven, since then the upper bound of the chargers per charger location can be met. Constraint (5.3) is not used in the model due to the previous mentioned problems.

The second disadvantage of using a diving heuristic is that it does not guarantee the optimal solution. In general, it has to give a good solution, but the solution does not have to be optimal. The variable with the largest fractional value is rounded up, but it is not sure that this is the best variable to round up. The larger the fractional value is the higher the chance is that it is the best variable to round up and that it has to be placed in the final solution. How large the fractional values of the optimal solution of the RMP are depends on two factors. The first one is the most important one and is the integrality gap. This is the gap between the optimal final solution of the RMP and the optimal integer solution. A larger

integrality gap means generally that the largest fractional value of the variables is probably low. The size of the integrality gap depends among other things on the formulation of the MP/RMP. The second factor is in the case that there are multiple optimal solutions. It is better to have columns that differ a lot from each other in V' than columns that are very similar. In the label-correcting algorithm only one of very similar vehicle tasks is added to the RMP. How much the optimality gap increases between the found solution and the optimal solution by using this diving heuristic is not known and has to be estimated with help of the lower bounds, which are mentioned in section 3.1.

5.2 The tailing-off effect in the column generation model

A well-known problem in column generation is the so called tailing-off effect: The RMP solution converges slower to the optimal solution each time the optimality gap between the current solution and the optimal solution decreases. This problem is shown in Figure 5.1. The main reason for this effect is



(a) *Change in objective value of the RMP* **(b)** *Change in objective value of the RMP after 100 iterations*

Figure 5.1: *Timetable Bordeaux: The tailing-off effect*

that the dual variables are unstable over the iterations, this is called the bang-bang effect. The primal solution (i.e., The RMP solution) always improves, but the dual solution of the column generation model does not always improve each iteration. This effect occurs due to that the simplex algorithm jumps from one extreme vertex to another extreme vertex. This leads to oscillation in the dual variables of the column generation model over the iterations, which leads to a slow convergence of the RMP. One of the reasons for these oscillations in the values of the dual variables is that there are degenerate solutions in the dual problem, i.e., there are multiple optimal dual solutions to choose from. One of these multiple optimal dual solutions is randomly chosen. To see the bang-bang effect, a dual bound has to be created for the RMP solution. The RMP solution is optimal, when the dual bound and the primal bound (the RMP solution) are equal. Since no dual bound has been found and therefore no stabilization method has been implemented, a reference is made to the papers of Desrosiers et al. [13] and Lübbecke [12] for an in-depth explanation on the reasons for the tailing-off effect, stabilization of the dual variables and finding a dual bound. Investigating the effects of stabilization on the dual variables in the current model is a recommendation for further research. This can reduce the number of iterations needed to solve the RMP optimally significantly.

To reduce the computational time the column generation method is often cut-off, before the optimal solution is found. A trade-off has to be made between the computational time and the optimality gap, which is the gap between the solution and the optimal solution. A common way to determine, when to cut-off the column generation method, is to use a dual bound. Such a bound has not been found for this problem as has been mentioned in previous paragraph. A different method is to track the improvement of the objective values in the RMP in percentage over multiple iterations. The model is cut-off, when the RMP does not improve 0.5% over 50 iterations. The number of iterations decreases to 20 iterations, after the first variable is rounded up. The RMP is thus not solved till optimality every time, but this most likely does not increase the optimality gap. This method has been implemented and a simulation

has been done with the same conditions as in Figure 5.1. Timetable Bordeaux has been used. The costs for a bus is 1 and the costs for the fast chargers is set to 0. The lower bound for the number of buses is 33. The simulation without the cut-off method has taken 369 iterations to find the optimal RMP solution, which has been 33. The simulation with the cut-off method has taken 323 iterations and the RMP objective value has been 33.098.

5.3 Choice of LP solver

The choice of LP solver can be important when many columns are created. The RMP is namely difficult to solve, when it contains many columns. There are multiple types of LP solvers that can be chosen to solve the RMP. The most used solver is the solver that uses the (dual) simplex method. The interior-point solver is one of the other options. (This solver gives benefits in the stabilization of the dual variables.) A choice has been made to solve the RMP with a simplex solver, because this is the most common used solver and thus most information can be found on using this solver. The standard LP solver of MATLAB is the linprog solver. This is according to the website of Mittelman [22] not the fastest available solver. The MDOPT and COPT solver are on average 20 times as fast as the linprog solver according to the benchmark of Mittelman. These solvers are however not commercially free. The best commercial-free solver is the CLP solver [25], which is almost up to five times as fast according to the benchmark of Mittelman. Therefore, it has been decided to implement this LP solver. The CLP solver can be used in MATLAB with the help of the OPTI toolbox [26]. The toolbox does not get updated anymore. The current version of the CLP solver is v1.16.11. In the future it might be possible that linprog becomes a faster solver. The model in MATLAB can be easily adjusted in order to implement a different solver.

5.4 Column management

At some point the number of columns in the RMP becomes too large to handle for the CLP solver. Column management is introduced to limit the number of columns and to keep the RMP solvable within a reasonable time. There are often a lot of columns that are not necessary to keep in the RMP and these can be deleted. Most of the used methods are based on deleting columns with the largest reduced costs. A limit is set on the maximum allowable number of columns in the RMP. Once this limit is exceeded, the reduced costs for each column is calculated. The columns with the highest reduced costs are deleted. The column management model implemented is based on this method.

Two column pools are used, where one pool is active and one pool is inactive. Columns from the active pool are transferred to the inactive pool, when the limit of maximum columns in V' is reached. The limit of number of columns allowed is determined by tuning, i.e., by running models multiple times with a different number of columns. This limit is 1500 columns in this model for all timetables in Appendix A. Using more than 1500 columns has led to an increasingly larger computational time. A check is done if the limit is exceeded once in so many iterations. This number also has to be tuned and is 5 in this model and depends on the time it takes to calculate the reduced costs of all columns versus the reduction in time of having a few columns less in the active pool. Columns are deleted from the active column pool in the case that the number of columns exceeds the limit of the maximum allowed number of columns in the active column pool. A list is created of vehicle tasks ordered on the current reduced costs. The columns that are deleted are the columns with the highest positive reduced costs. The columns that are present in the last solution of the RMP are excluded from this list, which is done to assure a feasible solution of the RMP in the next iteration. The columns that are created in the last few iterations are also excluded from the list. The columns that are deleted are transferred to the inactive column pool. At the same time columns from the inactive pool are checked whether there are any columns with negative reduced costs. The columns with negative reduced costs in the inactive pool are transferred to the active pool. Columns are deleted from the inactive pool, when there are so many columns in the inactive pool that storage of the columns becomes a problem or the calculation time for the reduced costs for each column becomes a problem.

5.5 Implementation of constraints

The implementation of constraints is at least as important as the choice of the LP solver and the implementation of column management, especially for large models. Efficient implementation of constraints is crucial to the computational time. It is not uncommon that the CG model needs a thousand iterations. The computational time is increased by a lot, if the creation of the RMP takes only a second per iteration. The solver in the OPTI Toolbox [26] requires the constraints in the following structure:

$$Ax \leq b \tag{5.4a}$$

$$Ax = b, \tag{5.4b}$$

where x are the decision variables in the objective function. The constraints in the script of Monhemius [2] and Wijnheijemer [3] are created with the solver-based method in MATLAB. The fields of the A matrix in $Ax \leq b$ is filled one by one, which makes it time-consuming to create the constraints for the RMP iteratively for a large model. Therefore, a different method has to be found.

The standard form of writing constraints in most solvers is in the MPS format. It is difficult to write constraints in MPS format and therefore it is not often used. A popular option is to write the constraints in an algebraic constraint programming language, which writes the constraints in an algebraic form. This makes it possible to create a set of constraints at once, which makes it a fast method to implement constraints. Common algebraic constraint programming languages are LP, GAMS and AMPL, which is the most popular one. The problem with using this language is that the program AMPL is not commercially free, which makes it difficult to use this language in the OPTI toolbox for MATLAB. This also holds for using GAMS. The commercially-free program language ZIMPL can create code in LP and MPS. There is one main problem. The main problem is that it is time-consuming to write constraints in ZIMPL and then convert them to MPS (,because the CLP solver with the use of the OPTI toolbox is not compatible with ZIMPL) and then send the constraints to the solver each iteration. It has been decided that an external language outside MATLAB is not convenient to use. MATLAB also has another manner of implementing constraint besides the solver-based method, which is called the model-based method. It has been introduced in MATLAB 2017b.

The model-based method works the same as an algebraic constraint programming language. The method looks promising, when it is implemented in the model of Wijnheijmer to test the speed. The constraints in the subproblem for Timetable 7 are created in a few seconds, when previously it has taken at least 800 seconds. The RMP constraints of the model in Chapter 4 have been modelled with the model-based method. The model-based method has at some point not been fast enough in implementing constraints for the model, because it uses as data type optimization variables, which are slowly progressed in MATLAB. The second problem has been that the LP solver only accepts the solver-based method as input. Thus, the model-based constraints have to be converted to the solver-based method, which takes too much time. Therefore, a switch has been made back to the solver-based method, after realizing that using array indexing with the solver-based method is actually the same as using algebraic constraints in the model-based method. The A matrix is now not filled field by field, but array by array with the help of array indexing. This reduces the computational time significantly. The solver-based method is implemented to create the constraints for the RMP each iteration. The advantage of the solver-based method is that it is faster. The advantage of the model-based method is that it is more clear for usage, which is useful for difficult constraints to implement. An improvement is maybe to use the sub2ind function in MATLAB. This function makes it possible to fill in an entire matrix at once. This is not necessary, since the time to create constraints is not the bottleneck.

5.6 Warm start

The last part that is implemented to accelerate the column generation method is the warm start. In the first iteration an initial solution is needed to create a feasible RMP, which can be solved. The model is initialized by creating columns for the RMP by planning one trip into one vehicle task, until each different trip is assigned to one vehicle task. The initial solution is poor and also the dual variables obtained from it are poor. The model takes a lot of time to be solved, when using only this initial solution, since the solution is so far off the optimum solution. Therefore, a warm start can be added

to the initial columns of the initialization. A warm start is a term for adding a decent solution at the initialization of the CG model and it can be used to reduce the computational time. A good warm start to implement is the concurrent scheduler of Wijnheijmer [4] or the greedy algorithm in section 3.2.2. This has not been done, because the models and constraints changed a lot during this research project and due to time constraints on the project the algorithms have not been adjusted to the current model. The warm starts, which are easy to implement, have been created at the end of the project. These are described in the following paragraph.

An easy warm start to implement is to create vehicle tasks which each consists of one shift. First, the trip with the latest start time is assigned to a new shift. Then the shift is expanded with trips that have the least time between the start time of the previous assigned trip to the shift and the end of the new trip. Trips are assigned to the shifts until the SoC costs of the trips reaches the maximum allowable value. Then the trip with the latest start time that has not been assigned to a shift is picked and assigned to a new shift. This repeats itself until all trips are assigned to a shift. The vehicle tasks have to be feasible columns, which means that a path has to be made from the start node of the depot to the end node of the depot at the end of the day. This means that first depot to depot arcs have to be taken until the arc is reached that goes to the first trip of the shift. Then trip to trip arcs are taken to all trips in the shift. From the last trip node in the shift an arc is taken back to the depot. Then depot to depot arcs are taken until the last depot node. For example trip 4 and trip 5 is a shift in Example 2 (see Figure 4.5). Then the path would be as follows: Depot node1, Depot node 2, Depot node 3, Trip node 4, Trip node 5, Depot node 5, Depot node 6 and as last Depot node 7.

This warm start still leads to a relative poor schedule and can only be used once. It cannot be used every time a variable is rounded up and a new RMP has to be solved. Another warm start can be easily implemented by making use of the label-correcting algorithm in Chapter 4. A relatively good solution can be found by changing the dominance rule of the label-correcting algorithm. The dominance rule includes four variables, but the number of variables that are taken into account can also be reduced. The idea is to only take the reduced costs into account. Thus, a path is dominated by another path in the same node, if the reduced costs are lower or equal in value. The column generation method is run with this adjusted dominance rule until the RMP does not improve with 1% in 10 iterations or no column can be found that has negative reduced costs. These latter two values can be tuned. Then the dominance rule is changed to that only the reduced costs and SoC are taken into account. This ends again, when the RMP does not improve 1% in 10 iterations or no column can be found with negative reduced costs. Then the label-correcting algorithm with the complete dominance rule is run. One of the main advantages of using these two algorithms to warm start the full label-correcting algorithm is that the algorithms can also be used when a new RMP is created after a variable is rounded up and fixed.

Using only the reduced costs in the dominance rule leads to a large increase in paths that are dominated early on in the label-correcting algorithms, which leads to a large reduction in computational time. The path with the most negative reduced costs is probably not found, but often a path is found that has negative reduced costs. A decent solution can be found, before the complete dominance rule in the label-correcting algorithm is used. Taking only reduced costs into account leads to that fast chargers are not used, since these increase the reduced costs of the path. Slow chargers can be used, since the use of the slow chargers has no influence on the reduced costs. The second dominance rule takes the reduced costs and the SoC into account, which means that the paths created can contain charging sessions from fast chargers. The second algorithm creates a good solution and is much faster than using the complete dominance rule, if long charge times and shift times are required.

5.7 Summary

In this chapter the implementation of the diving heuristic for rounding the fractional solution of the RMP to obtain an integer solution has been given. The problems of this rounding method have been discussed. Then the tailing-off effect on the improvement of the RMP solution has been discussed. A short explanation has been given with references to an in-depth explanation of this effect. A method has been given to prevent long computational times, which are caused by this effect. Then the choice of the LP solver has been substantiated. The LP solver that has been chosen is the CLP solver, which uses the (dual) simplex method. The goal of column management has been explained. This method has been used

to limit the number of columns in the column generation method to prevent the RMP from becoming too difficult to solve. Next, an explanation has been given on how constraints can be created fast. This has been done in MATLAB with the solver-based method and with the help of array indexing. Lastly, an explanation on the warm starts has been given. The warm starts create good schedules which can be used for the RMP to obtain good dual variables before the complete column generation algorithm is run. The first warm start consists of shifts that set-partition the trips. Each shift is assigned to one bus. The second warm start is created by running the column generation model with different dominance rules of the label-correcting algorithm. All elements have been implemented to simulate the model successfully.

Chapter 6

Results

In Chapter 4 the model to solve the eVSP of VDL has been explained. In Chapter 5 additional parts of the model have been described that make the implementation of the model possible. In this chapter the results of the implemented model are discussed. First, the results for scheduling the timetables of different cities are discussed. Secondly, the performance of methods to reduce the double driven trips are shown. Then a comparison is made between the results of Wijnheijmer [3] and the new results as far as the different constraints of both models allow. In the next section the diving heuristic (see section 5.1) is compared to the restricted master heuristic (see section 2.2.2). In section 6.5 the computational time of the label-correcting algorithm is compared to the computational time of using a MILP in the subproblem. The following section describes the influence of the number of deleted arcs from the graph. Lastly, the performance of the warm starts are shown. The model is implemented in MATLAB. The MATLAB scripts can be found in Appendix F. All simulations are done with a zbook 15 with an Intel core i7-4700 MQ with 8 GB RAM. Information on the timetables used in this chapter can be found in Appendix A.

6.1 Results for four cities

The model has been simulated for four timetables which can be found in Appendix A. The minimum charge time is 5 minutes and the minimum shift time is 0 minutes. The reason for this is that a minimum shift time increases the computational time a lot and is in most cases not necessary to use. The costs for a bus is 1 and the costs for a fast charger is 0.2, which are based on the assumption that a bus costs 500,000 euro and a fast charger 100,000. It is assumed that every driven km costs 1.5 kWh. This number is calculated based on the average speed of a bus in a city. The energy consumption of the trips as well as for the deadhead trips is based on this assumption. The average speed for the deadhead trips in the timetable of Eindhoven is assumed to be 25 km/h. The deadhead trips for the timetable Rotterdam are mostly arbitrary chosen. There is no information available on these deadhead trips for this timetable. The schedule is included for the sole purpose of showing that timetables with the size of the timetable of Rotterdam can be solved in reasonable time. The results for different cities are shown below:

City	Costs	No. buses	No. fast chargers	Lower bound no. buses	Double trips [%]	Integrality gap [%]	No. Arcs	Computational time
Le Havre	15.2	15	1	15	4.46	0.67	1491	00:16:30
Bordeaux	33.6	33	3	33	5.38	0.5	3829	00:51:41
Eindhoven	39.2	38	6	30	2.5	2.31	9451	06:57:48
Rotterdam	56.2	55	6	43	10.4	7.37	10717	10:57:18

Table 6.1: *Results of the simulations for four cities*

Double trips are trips that are assigned twice in a schedule to different buses. The integrality gap is the gap between the minimum RMP solution found and the integer solution, i.e., the integrality gap is calculated by the value of the integer solution divided by the value of the fractional solution minus

1 times 100. The number of arcs is the total number of arcs in the graph. The Gantt chart for the schedule of Bordeaux is shown in the Figure 6.1. The Gantt charts of the schedule of Le Havre and

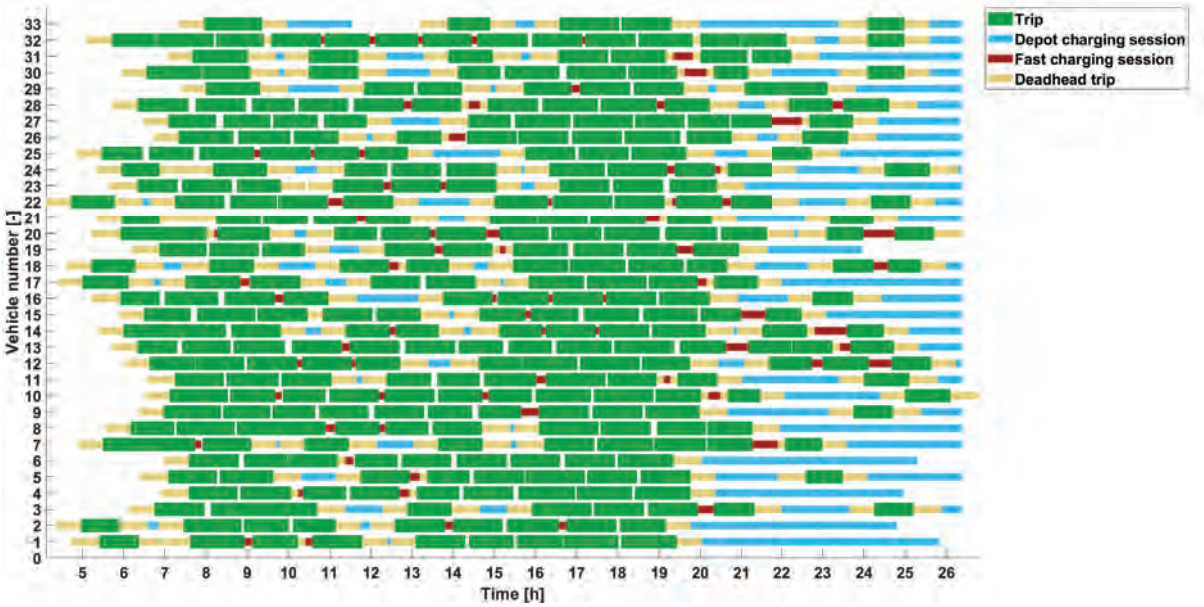


Figure 6.1: *Gantt chart: Timetable Bordeaux*

the schedule of Eindhoven can be found in Appendix E. The number of buses used for Le Havre and Bordeaux is equal to the lower bound. The number of buses used in Eindhoven and Rotterdam is above the lower bound. The currently used schedule in Eindhoven consists of 40 buses and 6 fast chargers. The model uses thus fewer buses, than currently are used. The schedule of the model cannot be completely compared with the current schedule due to the assumptions, but it gives a good indication. The reason why in Eindhoven the model cannot find a schedule close to the lower bound of the number of buses is because the fast chargers are placed in an unfavourable place. All fast chargers are placed at the depot, which is only in the neighbourhood of the central station. This means that only from the central station a bus can go charging without losing a lot of time and energy for driving the deadhead trip to the depot. It is more profitable to place fast chargers on multiple places around the city or directly at the central station. The reason for the deviation to the lower bound for the number of buses in the case of Rotterdam is most likely, because of the arbitrary deadhead trips. Timetable Rotterdam without deadhead trips is the same as Timetable 7. The lower bound for the number of buses used is reached for Timetable 7.

The lower bound for the number of fast chargers is always zero. This makes it difficult to estimate whether the number of chargers used is close to the optimal solution. It is known that for the current schedule for Le Havre 15 buses and two fast chargers are used. The real schedule cannot be compared fully with the schedule made due to the assumptions, but it gives an indication that the number of fast chargers used is acceptable. For Bordeaux the real schedule is not known, the max deviation from the optimum solution is $33.6/33 = 1.8\%$.

The bottleneck of the computational time of the model is still the subproblem. For the simulation of Eindhoven the label-correcting algorithm used 20303 seconds out of the 25068 seconds of the total simulation, which is equal to 81.0% of the computational time. The LP solver of the RMP used 3025 seconds and takes 12.1% of the computational time. The column management took 315 seconds and takes 1.2% of the computational time. The 1425 seconds left are mainly used for the creation of the constraints. This is 5.7% of the computational time.

The number of trips that are driven twice in a schedule is large. The double driven trips exist because some charging sessions are free of costs. These are the charging sessions from the slow charger. Thus, if there is enough time in a schedule a bus can take a slow charging session and then drive an extra trip without influencing the costs of the objective function. One out of the pair of the double driven trips can be seen as deadhead trip. In case that this is not allowed three other solutions can reduce the double driven trips from occurring. The first one is by adding a small cost to the dual variables of the

slow charging sessions each iteration. This means that charging is not free any more and therefore free trips cannot be driven anymore. The second solution is to delete all columns that contain a trip that is already in the final solution. The last option is to add energy costs to the objective function of the RMP.

6.2 Reduction of the double trips driven

The timetable of Le Havre is used to study the difference in double trips driven for different methods. In the first simulation no method is implemented to prevent the double trips. In the second simulation the arcs from and to the trips that are already in the final solution are deleted from the subproblem. (The default case of the model.) In the third simulation all columns are deleted that contain trips that are already in the final solution. The trip variables are practically deleted from the RMP. In the last simulation a small value is added to the dual variables for the charging sessions of the slow charger each iteration to prevent unnecessary charging. The results are shown in the table below:

Method:	Costs	RMP costs	Double trips %	Computational time
No method	33.8	33.37	11.39	00:48:14
Delete arcs in subproblem	33.6	33.4	5.38	00:51:41
Delete arcs and columns	34.6	33.47	0	02:49:33
Add small value to the dual variables of each charging sessions	33.8	33.64	6.65	03:20:44

Table 6.1: *Results for multiple methods to reduce the number of double driven trips*

The deletion of the arcs in the subproblem and the deletion of the columns which have those arcs included leads to a higher computational time. This increases with the number of buses used, because multiple times a lot of columns are deleted then. There are 0% double trips driven. This is as expected.

It has been expected that adding a small value to the dual variables that are linked to the charging session would lead to 0% double driven trips, which is not the case. The reason for this is that there can also be vehicle tasks made without the use of a charging session. The initial 100% SoC of a bus is seen as free energy. This means that double trips can still be driven freely in some cases when not enough trips are in the vehicle task to reach the minimum SoC.

The computational time when using no method to reduce the number of double driven trip is the lowest for the schedule of Bordeaux. This is not as expected. The deletion of the arcs in the subproblem makes the subproblem smaller, which makes it easier to solve. The two options are simulated for the Timetable of Le Havre and Timetable 7 to see if this is also the case for other timetables.

Timetable	Method:	Costs	RMP costs	Double trips %	Computational time
Le Havre	No method	15.2	15.072	8.91	00:19:19
Le Havre	Delete arcs in subproblem	15.2	15.098	4.46	00:16:30
Timetable 7	No method	44	43	19.16	04:31:36
Timetable 7	Delete arcs in subproblem	43	43	5.84	02:18:39

Table 6.2: *Results for Timetable Le Havre and Timetable 7*

In both cases the method of deleting arcs in the subproblem is faster. For the large Timetable 7, it is much faster. It might be that in the schedule of Bordeaux paths are earlier dominated, when no method is used. This means that fewer paths are created, which can lead to a lower computational time.

6.3 Comparison with the previously created models

In this section the current model is compared to the two previously created models of Wijnheijmer [3]. The previous created models are the Concurrent scheduler (CS) and a column generation model. The same conditions are used as in the thesis of Wijnheijmer [3]. The minimum charger time used is 0.833 minute. There is a one-minute delay between two trips. An unlimited number of chargers can be used. This is due to the difference in models. The model of Wijnheijmer constraints the number of chargers, while the current model minimizes the number of chargers. The results are shown in the table below:

Timetable name:	CS: No. buses	CS: Computational time	CG Wijnheijmer: No. buses	CG Wijnheijmer: Computational time	New model: No. buses	New model: Computational time
Timetable 1	5	00:00:16	5	00:00:30	3	00:00:09
Timetable 4	9	00:00:24	10	00:19:37	7	00:02:41
Timetable 7	52	00:02:03	-	-	43	02:12:58
Timetable 10	2	00:00:11	2	00:00:22	1	00:00:08

Table 6.1: *Comparison of the previous model versus the current model*

From Table 6.1 it can be concluded that for these settings the current model is better. In all cases the current model reaches the lower bound for the number of buses, while the other models do not reach the minimum bound in any of the cases. The computational time of the concurrent scheduler is lower for the larger timetables, but the computational time of the current model is in both cases still a reasonable time. One of the reasons why the CG model of Wijnheijmer does not reach the same quality of solutions is because a different rounding algorithm is used. Wijnheijmer uses the restricted master heuristic, while the current model uses a diving heuristic.

6.4 Performance of the diving heuristic versus the restricted master heuristic

In this section the performance of the diving heuristic compared to the restricted master heuristic is shown. The maximum total of columns in the active column pool is 1500. The results are simulated with the same assumptions as in section 6.1 and can be found below in Table 6.1:

Method:	Timetable:	Costs	Double driven trips [%]	Integrality gap [%]	Computational time
Restricted master heuristic	Le Havre	20.4	44.55	35.27	00:07:07
Diving heuristic	Le Havre	15.2	4.46	0.67	00:16:30
Restricted master heuristic	Bordeaux	40.8	33.54	22.12	00:23:34
Diving heuristic	Bordeaux	33.6	5.38	0.5	00:51:41

Table 6.1: *Comparison of rounding methods*

The conclusion that can be drawn from Table 6.1 is that rounding with the restricted master heuristic gives worse results than rounding with the diving heuristic, because the integrality gap is larger. This is also expected, since the restricted master heuristic assumes that the same columns are needed to obtain the optimal fraction solution and the optimal integer solution, which is often not true. The diving heuristic takes this fact into account by adapting the column pool based on which column is already

in the final solution. The second disadvantage is that the restricted master heuristic contains a large percentage of double driven trips.

6.5 Performance of the label-correcting algorithm compared to a MILP solver

The subproblem can also be solved with a MILP solver. It is interesting to study the difference in computational time compared to the label-correcting algorithm and the difference in costs of the solution. The CBC MILP solver is used to solve the subproblem and is implemented with the help of the OPTI toolbox [26]. The MILP formulation used to solve the subproblem with the MILP solver can be found in Appendix D. There are a few constraints not implemented. The constraint for minimum charge time and shift time are not implemented in MATLAB for the MILP solver. The MILP formulation does not include deadhead trips, therefore Timetables 4 and 7 are used to estimate the performance of the MILP solver compared to the label-correcting algorithm. The costs for the chargers are zero. Only fast chargers are used, and they replace the slow chargers at the depot. The warm start from the varying dominance rule is not used when simulating with the label-correcting algorithm. Arcs to and from trips that are already in the final solution are not deleted from the subproblem. The other parameters are the same as used in section 6.1. The result can be found in the table below:

Method	Timetable name:	Costs	RMP Costs	Computational time	No. of arcs
MILP	Timetable 4	7	7	00:28:24	1242
MILP	Timetable 7	-	43	11:55:09	16208
Label-correcting algorithm	Timetable 4	8	7.0319	00:05:51	1242
Label-correcting algorithm	Timetable 7	45	43	03:42:55	16208

Table 6.1: *Comparison between the results of solving the subproblem with a MILP solver and solving the subproblem with a label-correcting algorithm*

The simulation with the MILP solver has been stopped after 42909 seconds, since the simulation would have taken too long. At that point 4 variables have been rounded and it still would have to round at least 39 others (since that is the lower bound for the number of buses for Timetable 7). The simulation for the label-correcting algorithm has taken less time, it only has taken 13375 seconds. In both simulation the label-correcting algorithm is faster. There are two reasons for this. The first reason is that the label-correcting algorithm has to run through a number of steps equal to the number of arcs, while the number of steps in a MILP solver is $2^{No.\,arcs}$. The second reason is that the dominance rule is effective in deleting a large amount of possible paths.

The solution of the label-correcting algorithm does not reach the lower bound of 43 buses for Timetable 7. The schedule uses 2 buses more. The RMP does however reach the lower bound of 43. It is known from the previous section 6.3 that this lower bound is obtainable (note that in previous section there is a minimum charging time). This means that a suboptimal integer solution has been found due to rounding. The same holds for the results for Timetable 4, where the costs for the RMP are different. The label-correcting algorithm has been cut-off at a value of 7.03, which means that it converges slower than the MILP solver in this case. A reason why the convergence is different is that the MILP solver can create different columns. Although both the MILP solver and the label-correcting algorithm find a path with the lowest reduced costs, the paths do not have to be the same. There can be multiple paths with the same reduced costs. Another difference is that the label-correcting algorithm can add multiple columns to the RMP in one iteration. To conclude the label-correcting algorithm is a better option than the MILP solver. This is mainly because of the large difference in computational time. The label-correcting algorithm can provide the same optimal solution in both cases for the RMP, when no cut-off method is used.

6.6 The influence of the number of deleted arcs

The reduction of the graph can have a large influence on the performance of the model. The graph is reduced by implementing a maximum time for deadhead trips and a maximum idle time. In this section the Timetable of Bordeaux is used to study the influence of the number of deleted arcs. The maximum time for a deadhead trip and the maximum idle time is on default 10 minutes. The maximum deadhead trip time to a charger is 11 minutes, because the fast chargers are 11 minutes away in the Timetable of Bordeaux. In this section the three parameters are simultaneously varied to research the influence of the deleted arcs by the restrictions of these parameters. The results can be found in the table below:

Max. time restrictions [min]:	Costs	RMP costs	No. arcs	Computational time
0	49	48.8	3177	00:12:26
5	37.8	37.51	3368	00:22:27
10/11	33.6	33.4	3829	00:51:41
20	34.4	33.25	4352	01:01:05
-	33.2	33.22	44714	04:10:29

Table 6.1: *The influence of the number of deleted arcs on the resulting schedule for the Timetable of Bordeaux*

From the results in Table 6.1 it can be concluded that the deletion of the arcs has a large impact on results and computational results. The number of steps that must be gone through in the label-correcting algorithm is equal to the number of arcs in the graph as explained earlier in Chapter 4. Someone may think that therefore the relation between the computational time and the number of arcs has to be linear. There are two reasons why this is not the case. The first one is that the number of steps is not the only influence factor in the computational time. The labels of the paths are stored in matrices. The size of the matrices has a large influence on the computational time in the MATLAB script. The second reason is that in the current MATLAB script (Appendix F) all new updated paths are checked with the dominance rule with the already existing paths in the node one by one. Thus, one new path is checked with all other already existing paths in the node at once, if it is dominated or if it dominates one of the existing paths in the node. Instead of that all new updated paths are checked at once with all the already existing paths in the node. However, the storage is done for all paths at once. Concluding, the computational time depends on the number of arcs, but also depends on the number of possible paths created in the label-correcting algorithm. A recommendation is to check all updated paths at the same time with the already existing paths in the node. This leads to a decrease in computational time.

The influence of the number of arcs on the costs can be explained as follows: The more arcs there are in the graph the more possible paths can be created. This means that extra paths can be created that can have high negative reduced costs, which can lead to a schedule with lower costs. At some point this effect of more arcs leading to better paths does not hold anymore, since all paths that already have high negative reduced costs can already be made in the graph. Then adding new arcs to the graph does not create new paths with higher negative reduced costs. For example in Bordeaux the deadhead trip to the fast chargers from a trip location is in a few places 11 minutes. Increasing the maximum deadhead time from 5 minutes to 11 minutes has a large impact, while increasing it from 11 minutes to 20 minutes has not much impact, since no new charger locations can be reached.

Looking at the RMP costs it can be concluded that at least one charger can be used less, when no time restrictions are in place. However, from a practical point of view the schedules with no time restrictions are not executable. A long idle time is often not possible due to zero or limited parking space. It also means that drivers are doing nothing for a long period of time. Long deadhead trips are often also not appreciated. One of the disadvantages is that driving long deadhead trips leads to extra energy consumption. The energy costs are not specifically taken into account in this model, but by limiting the allowable deadhead trip duration it is partly taken into account.

6.7 Performance of the warm starts

In this part the benefits of the implemented warm starts are shown. The warm starts have been discussed in section 5.6. For the simulation the same parameters are used as described in section 6.1. In the first simulation both warm starts are used. In the second simulation the warm start of assigning one shift per bus is turned off. In the third simulation the warm start of varying the dominance rule is turned off. The results are shown below in Table 6.1.

	Timetable:	Costs	Computational
Warm start	Le Havre	15.2	00:16:30
No shifts	Le Havre	15.2	00:17:26
No varying dominance rule	Le Havre	15.2	00:22:13
Warm start	Bordeaux	33.6	00:51:41
No shifts	Bordeaux	33.6	00:58:18
No varying dominance rule	Bordeaux	33.6	01:09:27

Table 6.1: *Influence of the warm starts on the computational time*

From Table 6.1 it can be concluded that both warm starts reduce the computational time. The varying dominance rule reduces the computational time the most.

6.8 Summary

In this chapter the results of the model have been given. First the results for different cities have been shown. The schedules for the timetable of Le Havre and the timetable of Bordeaux give good results. The lower bound for the number of buses is reached. The third schedule for the timetable of Eindhoven uses more buses than the lower bound, but uses fewer buses than currently are used in real time. The schedule of Rotterdam can be simulated within reasonable time. The results appear to be poor for this schedule. The main reason for this is because the deadhead trips are chosen arbitrary. The best method to prevent the double trips is to delete the trips from the problem that are already in the final solution. This method is not the default setting, because the computational time is too high, especially for larger schedules. Therefore, the method that deletes the trips that are in the final solution from the subproblem is implemented. In section 6.3 the current model has been compared to the previous models as far as possible. An unlimited amount of chargers can be used without any costs. The current model performs for each timetable better than the previous methods [3]. The current model reaches the lower bound for the number of buses in all cases. The Concurrent scheduler is faster in creating a schedule, but the current model can also create all schedules in reasonable time. In section 6.4 the diving heuristic has been compared to the restricted master heuristic. The maximum column pool is set to 1500. The restricted master heuristic is faster, but performs worse. It has been concluded that the diving heuristic is the better option to use.

In section 6.5 it has been shown that the label-correcting algorithm solves the subproblem more efficiently than the MILP solver. The influence on the size of the graph has been discussed in section 6.6. The number of deleted arcs has a large influence on the computational time. The relation between the number of arcs and the computational time is not linear. The computational time also depends on the number of paths created. The reasons for this are the increased size of the matrices and the checking of updated paths one by one whether they are dominated or not. In the last section the influence of the two warm starts have been shown, which both reduce the computational time. The varying dominance rule decreases the computational time the most.

To conclude this chapter, the results look promising. The number of buses used are often equal to the lower bound, but it is sometimes not reached for timetables where it is known that it is possible to reach the lower bound. This is probably due to rounding errors from the diving heuristic. The results in

such cases are one or two buses off the lower bound. The lower bound for number of fast chargers is too conservative to draw conclusions from on the used number of fast chargers in the created schedules. The number of fast chargers has to be compared to a real schedule to draw a conclusion. The number of fast chargers used for Le Havre and Eindhoven are equal to the number used in real time. This comparison is only an estimation, since some assumptions have been made in this model. The computational time is not an issue for the currently created schedules. Solving the subproblem is still the bottleneck of the current model.

Chapter 7

Conclusion and recommendations

The goal of this project is to create a tool for VDL that can create schedules for electric buses based on given timetables within a reasonable time. These schedules have to meet the requirements stated in section 1.2 to create a schedule that can be used in practice. In this chapter a conclusion is drawn on the created tool. At the end some recommendations are given to improve the current tool.

7.1 Conclusion

There are three main problems in this projects. The first main problem is to create a schedule from a timetable that can be used in practice. The second main problem is to minimize the number of buses and fast chargers used in the schedule. The third main problem is to create a schedule in a reasonable time. The project has as a starting point the column generation model of Wijnheijmer [2]. In Chapter 3 multiple algorithms have been introduced to accelerate the subproblem of the model of Wijnheijmer [2]. These are a greedy algorithm, a genetic algorithm and a diving heuristic. The greedy algorithm is not accurate enough. The genetic algorithm is too slow and does not give an accurate solution. One of the main problems is that a lot of infeasible solutions are created. The diving heuristic is also too slow and infeasible solutions occurred. The conclusion has been drawn that a different approach is needed. Therefore, the multiple step column generation method has been introduced. This method subdivides the subproblem in multiple subproblems and places the subproblems in series. The problem with this method is that the subproblems can not be solved in series. Each time a subproblem is solved, the RMP has to be solved first to update the dual variables correctly before the next subproblem can be solved. The method cannot be formulated correctly and again a new approach is taken.

The successful and final approach is based on describing the subproblem with the help of a graph. The size of the graph is reduced by applying time constraints on deadhead trips and idle time of a bus. The subproblem is transformed to a shortest path problem (SPP), whereby the reduced costs are seen as the distance. The subproblem is in specific an elementary resource constrained shortest path problem (ERC-SSP). The shortest path can be found by a label-correcting algorithm. This new method has been implemented in the new model, which has been described in Chapter 4. The new model has the objective to create a schedule from a timetable with a minimum amount of buses and a minimum amount of fast chargers. The new model is based on a column generation method, whereby the subproblem is rewritten to an SPP. It includes most of the requirements of VDL given in section 1.2. Two requirements are missing. Currently, the charge rate follows a linear curve, while the charge curve is in reality non-linear. The second requirement is that the number of chargers are not restricted at a location. This has been explained in section 5.1. Restricting the number of chargers at a location leads to many infeasible rounded up variables by the diving heuristic, which leads to poor results. For the same reason it is not restricted that trips can only be assigned once in the schedule. Schedules can contain between 0% and 10% of trips that are been driven twice. One of the pair of trips can be seen as a deadhead trip. An option is implemented that restricts that trips can be assigned more than once. This option is disabled by default because it leads to longer computational time. In the other sections of Chapter 5 additional parts are described on how to successfully implement the model.

In section 6.1 the results of the new model have been shown for different cities. One important as-

sumption has been made for all the schedules and that is that the energy consumption per kilometre is 1.5 kWh/km. The schedules for the cities of Le Havre meet the lower bound for the number of buses. The lower bound for the number of fast chargers is too conservative to obtain a good estimation for the minimum number of fast chargers needed. A better estimation can be obtained by comparing the created schedules with schedules that are used in practice. The schedule for Eindhoven uses 38 buses and 6 fast chargers. Although the number of buses is a bit off the lower bound for the number of buses, it uses fewer buses than the current schedule used in Eindhoven, which uses 40 buses and 6 fast chargers. The goal of the project is to be able to simulate the schedule of Rotterdam within reasonable time. It is shown that this schedule can be created within reasonable time. In general, the results of most schedules look promising. The quality of the solution is often good. Sometimes more buses than needed are used. This is mainly due to rounding the wrong decision variable with the diving heuristic.

7.2 Recommendations

The recommendations are divided in two parts. The first part has as topic the reduction of the computational time of the model. The second part describes recommendations for the implementation of the missing two requirements and improvements in the quality of the solution.

The bottleneck of the model is solving the subproblem. Therefore, the recommendations are based on improving the computational time of the subproblem. The graph can be improved. Currently, two lines of charger nodes are created, when there are two types of chargers at one location. This can be improved by only using one line of charger nodes with in-between the nodes multiple arcs. One arc describes the use of one of the charger types, while the other describes the use of another charger type. This reduces the number of arcs. The second recommendation is to create a more efficient script for the label-correcting algorithm, which has been originally written to be understandable instead of immediately focussing on computational efficiency. A possible improvement is to check all newly updated paths with the already existing paths in a node at once. This now happens one by one as is described in section 6.6. The second possible improvement is to use for loops that are solved in parallel with the help of the Parallel computing toolbox of MATLAB. In the current script it is not possible to implement parallel computing efficiently. The third improvement is to improve the storage of paths in the matrices. The last recommendation is to check whether the current model suffers a lot from unstable dual variables as mentioned in section 5.2. It is very likely that the model does suffer from this due to that the columns contain a lot of rows. A recommendation is to implement stabilization methods on the dual variables. This leads to a faster convergence of the RMP solution and thus fewer iterations are needed to solve the RMP to optimality.

The second part of recommendations is about the implementation of the missing requirements and improving the quality of the solution. There are two requirements that are missing. The first requirement is to use a non-linear charge curve to determine the charge rate, which can be easily implemented in the current model. In the label-correcting algorithm the SoC of a path is known, before a charging session. The charge rate e_{charge} can be changed based on the current SoC of the path. This can be done by approximating the non-linear curve with piecewise linear functions. The charge rate can be constant throughout a charging session due to the fact that the charging sessions are often in steps of one minute. This is an accurate approximation of the non-linear charge curve. The second requirement is that it is sometimes handy for the sales department of VDL Bus & Coach to restrict the number of chargers at a certain location. The diving heuristic performs poorly with such a constraint. It might be an option to use a branch-and-price method. This method finds the optimal integer solution, which is thus also sometimes better than the solution from the diving heuristic. A constraint that restricts the chargers can be implemented, because there are no rounding errors. The sign in the constraint (4.7) can be an equal sign for the same reason. The main question is whether the branch-and-price algorithm is fast enough. It is recommended that first the computational time is reduced of the subproblem, before a branch-and-price algorithm is implemented. The last recommendation is to find a solution for long charging sessions (i.e., a long time interval between charger nodes). For small problems a solution can lead to the use of fewer chargers. The problem currently is that the label-correcting algorithm can only choose to use the full charging session or not use the charging session at all. For long charging sessions

this is not ideal. A solution is to create more charger nodes to split the long charging session in multiple small charging sessions. A second option is to adjust the percentage of the charging session r_{pcl} in a column, after the column has been created by the label-correcting algorithm.

Bibliography

[1] ZeEUS eBus Report #2 (2017). Retrieved from: https://zeeus.eu/uploads/publications/documents/zeeus-report2017-2018-final.pdf

[2] Monhemius, M.A., (2019). Solving an Electric Vehicle Scheduling Problem using Mixed Integer Linear Programming. DC 2019.047, Eindhoven University of Technology, Department of Mechanical Engineering, Dynamics and Control Group, Eindhoven, The Netherlands. Coach: Lefeber, E..

[3] Wijnheijmer, J.W.M., (2020). Development of a Vehicle scheduling Tool for Large-Scale Electric Bus Transit Networks. DC 2020.017, Eindhoven University of Technology, Department of Mechanical Engineering, Dynamics and Control Group, Eindhoven, The Netherlands, MSc Thesis. Supervisor: Nijmeijer, H.; Coach: Lefeber, E..

[4] Adler, J.D., (2014). Routing and Scheduling of Electric and Alternative–Fuel Vehicles. Arizona State University, Phoenix, United States, PHD Thesis. Retrieved from: https://repository.asu.edu/attachments/134788/content/Adler_asu_0010E_13619.pdf

[5] Sassi, O. & Oulamara, A., (2017). Electric vehicle scheduling and optimal charging problem: complexity, exact and heuristic approaches. International Journal of Production Research, 55(2), 519–535. Retrieved from: https://doi.org/10.1080/00207543.2016.1192695

[6] Bunte, S. & Kliewer, N., (2009). An overview on vehicle scheduling models. Public Transport, 1(4), 299–317. Retrieved from: https://doi.org/10.1007/s12469-010-0018-5

[7] Pereira, M.L.B.A., (2019). The Vehicle Scheduling Problem of Electric Buses. Universidade Do Porto, Porto, Portugal, MSc Thesis. Supervisor: Dias, M.T.G.. Retrieved from: https://repositorio-aberto.up.pt/bitstream/10216/122669/2/357012.pdf

[8] Perumal, S.S.G., Dollevoet, T.A.B, Huisman, D, Lusby, R.M, Larsen, J. & Riis, M., (2020). Solution Approaches for Vehicle and Crew Scheduling with Electric Buses. Econometric Institute Research Papers, EI-2020-02. Retrieved from: http://hdl.handle.net/1765/123963

[9] Chao, Z. & Xiaohong, C. (2013). Optimizing battery electric bus transit vehicle scheduling with battery exchanging: Model and case study. Procedia – Social and Behavioral Sciences, 96, 2725–2736. Retrieved from: https://doi.org/10.1016/j.sbspro.2013.08.306

[10] Teng, J., Fan, W.D. & Chen, T., (2020). Integrated approach to vehicle scheduling and bus timetabling for an electric bus line. Journal of Transportation Engineering, Part A: Systems, 146(2). Retrieved from: https://doi.org/10.1061/JTEPBS.0000306

[11] Desaulniers, G., Desrosiers, J. & Solomon, M.M., (2005). Column generation (Ser. Gerad 25th anniversary series). Springer. Retrieved from: https://doi.org/10.1007/b135457

[12] Lübbecke, M.E., (2011). Column generation. Wiley encyclopedia of operations research and management science. Retrieved from: https://doi.org/10.1002/9780470400531.eorms0158

[13] Lübbecke, M.E. & Desrosiers, J., (2005). Selected topics in column generation. Operations research, 53(6), 1007–1023. Retrieved from: https://doi.org/10.1287/opre.1050.0234

[14] Li, J.Q., (2014). Transit bus scheduling with limited energy. Transportation Science, 48(4), 521–539. Retrieved from: https://doi.org/10.1287/trsc.2013.0468

[15] Sundin, D., (2018). Scheduling of Electric Buses with Column Generation. Linköping University, Department of Mathematics, Linköping, Sweden . Supervisor: Quttineh, N.. Retrieved from: http://www.diva-portal.org/smash/get/diva2:1283727/FULLTEXT01.pdf

[16] van Kooten Niekerk, M., (2018). Optimizing for Reliable and Sustainable Public Transport. Utrecht University , PHD thesis at . Retrieved from: https://dspace.library.uu.nl/bitstream/1874/364146/1/vKootenNiekerk.pdf

[17] Posthoorn, C., (2016). Vehicle Scheduling of Electric City Buses. Delft University of Technology, Department of Applied Mathematics, MSc thesis. Supervisor: Aardal, K.I. & Post, H.N.. Retrieved from: https://repository.tudelft.nl/islandora/object/uuid%3A0a0b2596-e908-475c-b8d7-48e4b6bbb37d

[18] Joncour, C., Sadykov, R., Vanderbeck, F., Michel, S., & Sverdlov, D. (2010). Column generation based primal heuristics. Electronic Notes in Discrete Mathematics, 36(C), 695–702. Retrieved from: https://doi.org/10.1016/j.endm.2010.05.088

[19] Barnhart, C., Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W. & Vance, P.H., (1998). Branch-and-price: Column generation for solving huge integer programs. Operations research, 46(3), 316–329. Retrieved from: https://doi.org/10.1287/opre.46.3.316

[20] Feillet, D. (2010). A tutorial on column generation and branch-and-price for vehicle routing problems. 4OR, 8(4), 407–424. Retrieved from: https://doi.org/10.1007/s10288-010-0130-z

[21] Desaulniers, G., Desrosiers, J. & Solomon, M. M. (2002). Accelerating strategies in column generation methods for vehicle routing and crew scheduling problems. In Essays and surveys in metaheuristics, 309–324. Springer, Boston, MA. Retrieved from: https://doi.org/10.1007/978-1-4615-1507-4_14

[22] Mittelman, H.D. (2020, 22 December). Benchmarks for optimization software. Arizona state university, Phoenix, United states. Retrieved from: http://plato.asu.edu/bench.html

[23] Irnich, S. & Desaulniers, G. (2005). Shortest path problems with resource constraints. In Column generation (pp. 33–65). Springer, Boston, MA. Retrieved from: https://doi.org/10.1007/0-387-25486-2_2

[24] Feillet, D., Dejax, P., Gendreau, M. & Gueguen, C. (2004). An exact algorithm for the elementary shortest path problem with resource constraints: application to some vehicle routing problems. Networks, 44(3), 216–229. Retrieved from: https://doi.org/10.1002/net.20033

[25] Forrest, J., Hall, J., (2020, 22 December). CLP solver. Retrieved from: https://projects.coin-or.org/Clp

[26] Curie, J., (2020, 22 December). Opti toolbox: A free MATLAB Toolbox for Optimization. Retrieved from: https://inverseproblem.co.nz/OPTI/

[27] Vance, P. H., Barnhart, C., Johnson, E. L. & Nemhauser, G. L. (1997). Airline crew scheduling: a new formulation and decomposition algorithm. Operations Research, 45(2), 188–200. Retrieved from: https://www.jstor.org/stable/171737

Appendix A

Timetables

In this section of the appendix the information of the timetables used in this thesis are given. Timetables 4 and 7 are used in Chapter 3 and all the timetables are used in Chapter 6. Each timetable has one depot, where there are as many slow chargers as there are buses.

Name	No. Trips	No. Locations	No. fast Charger locations	Charge rate slow chargers (kW)	Charge rate fast chargers (kW)	Battery capacity (kWh)	Max. SoC (%)	Min. SoC (%)	Deadhead trips
Timetable 1	14	1	0	230	-	216	90	10	-
Timetable 4	203	1	0	230	-	216	90	10	-
Timetable 7	1096	1	0	230	-	216	90	10	-
Timetable 10	13	1	0	230	-	216	90	10	-
Timetable Le Havre	202	4	1	50	420	288	90	10	Accurate
Timetable Bordeaux	316	7	3	50	330	420	90	20	Accurate
Timetable Eindhoven	977	9	1	50	270	180	90	20	Assumed buses drive 25 km/h
Timetable Rotterdam	1096	21	1	50	230	216	90	10	Partly arbitrary chosen

Table A.1: *Information on the timetables used in this thesis*

Name	Lower bound buses	Lower bound fast chargers	Max. deadhead trip time	Max. deadhead trip time to charge	Max. idle time
Timetable 1	3	1	-	-	-
Timetable 4	7	1	-	-	-
Timetable 7	43	2	-	-	-
Timetable 10	1	1	-	-	-
Timetable Le Havre	15	0	10	10	10
Timetable Bordeaux	33	0	10	11	10
Timetable Eindhoven	30	0	17	27	10
Timetable Rotterdam	43	0	10	10	10

Table A.2: *Information on the timetables used in this thesis*

The deadhead trips for the Timetable of Rotterdam are arbitrary chosen, since the length of the deadhead trips are unknown. The main reason to use the timetable of Rotterdam is to show that the model can simulate such large models in a reasonable time.

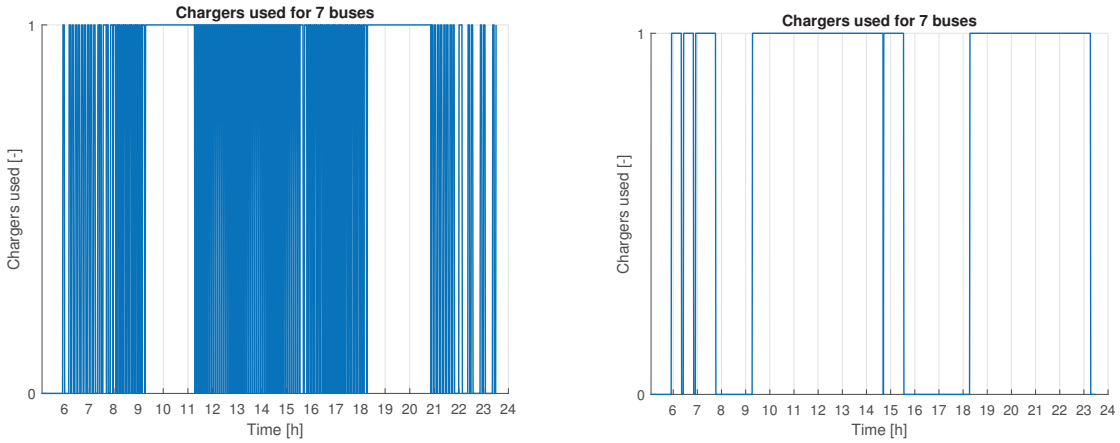
Appendix B

Explanations of different failed attempts to solve the eVSP

In this Appendix the attempts that were unsuccessful to solve the eVSP of VDL are explained. The first algorithm explained is an algorithm based on an energy lower bound. The other methods in this appendix were already partly addressed in Chapter 3, but are explained in depth in this appendix. These are the greedy algorithm, the genetic algorithm, the diving heuristic and the multi-step column generation model.

B.1 Algorithm based on an energy lower bound

The combination of both lower bounds on the number of buses in section 3.1 can give information on when it is profitable to let a bus charge. The number of chargers used at each time point can be found with the help of the model for creating the lower bound based on energy. In the model it is calculated how many of the fixed number of chargers are used each minute. The model starts with the minimum number of buses needed according to the lower bound of simultaneously driven trips, when the lower bound of simultaneously driven trips has a higher value. All charging sessions are at the moment one minute long, but the duration can be changed. The following figures can be found for timetable 4:



(a) *Timetable 4: Number of chargers used in the system per minute*

(b) *Timetable 4: Number of chargers used in the system per minute with charging sessions of 25 minutes*

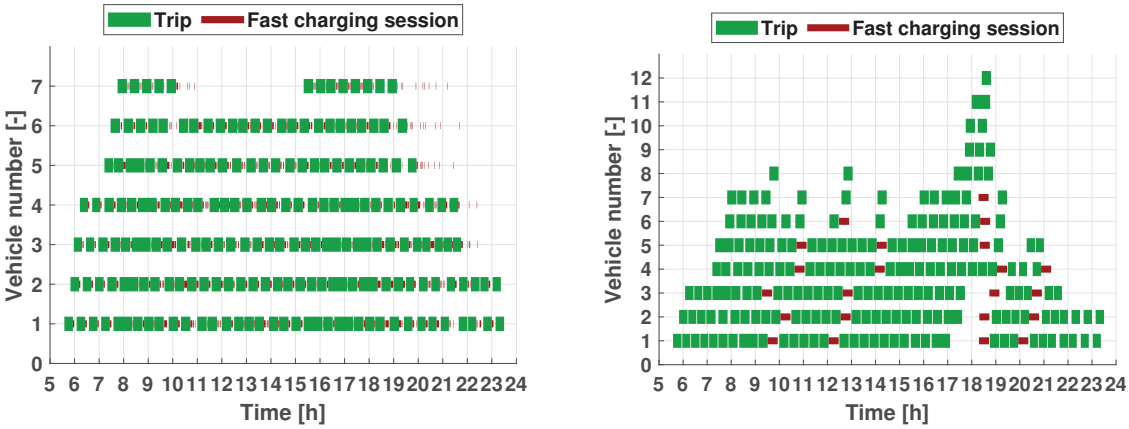
Figure B.1: *Number of chargers used as function of time in the combined model of the two lower bounds for the number of buses*

This information can be used to create a schedule with the help of an algorithm. In Figure B.1a every minute that a charger is used can be seen as a charging session. The charging sessions can be assigned to buses just as trips can be assigned. The trips and charging sessions are ordered in a list based on start time. The algorithm starts with one bus. Then the first item of the list is taken. The trip is assigned to

the bus, if that is possible. In the case that it is not possible, a new bus is added to the problem and the trip is added to this bus. Then the next item is taken from the list. In case that it is a trip, it is again assigned to bus one, if possible or otherwise it assigned to bus two. In case that both options are not possible the trip is assigned to a new bus. This process continues till all trips are assigned. Charging sessions are assigned to an idle bus with the lowest SoC.

An option is also to assign trips to the bus with the highest SoC at the moment instead of assigning trips to buses in a fixed order. This method of assigning trips works better, when there are no charging sessions, but it performs worse, when charging sessions are taken into account. Another problem is that it creates a long break time between two trips, which is not preferable. Therefore, this method of assigning trips to buses is not implemented.

The results of the new algorithm are shown in two Gantt graphs in Figure B.2. The solution in Figure



(a) *Timetable 4: Schedule for timetable 4 with charging sessions of one minute*

(b) *Timetable 4: Schedule for timetable 4 with charging sessions of 25 minutes*

Figure B.2: *Results of the Algorithm based on an energy lower bound*

B.2a reaches the lower bound of timetable 4 of the minimum number of buses that should be used. This means an optimal solution is found. The problem with the solution is that charging sessions are used of one minute. This is not a feasible result. Therefore, a simulation has ben done with charging sessions of 25 minutes. The result can be found in Figure B.2b. The results are poor. Better results were found, when using timetable 7 instead of timetable 4. The lower bound of 43 buses is reached, when charging sessions of one minute are used. In total 47 buses were used, when the charging session is 25 minutes, which is still a very good result. There are a couple of problems with the algorithm. The first problem is that the algorithm needs to work for every schedule. The second problem is that implementing and making efficient choices for deadhead trips is difficult to implement in this algorithm. It can be concluded that this algorithm is not sufficient to use due to the mixed results and has not enough potential to be extended.

B.2 Greedy algorithm

In this section the greedy algorithm of section 3.2.2 is presented. The greedy algorithm consists of 3 parts as explained in the main body of the report. The goal of this section is to make clear how the greedy algorithm works. In this section each step of the greedy algorithm is given.

Algorithm 2: A greedy algorithm

```
[v_new ] = function Greedy_algorithm(π, θ,σ and ϵ);
%% Part 1 choose trips based on max reduced costs;
Find first trip with π_t > 0;
while there are still trips after the latest chosen trip, which are compatible with the latest chosen
 trip do
    if e_current + e_t(new) < e_min then
        e_current = e_max;
        Update σ and ϵ;
    else
        Find the trips that are not compatible with the current trip;
        Compare π_t and choose the trip with the highest π_t ;
        Add the trip with with the highest π_t to v;
        Adjust SoC of the bus;
        Find the next trip, which is compatible with the previously assigned trip with π_t > 0;
        if there is no trip with π_t > 0 that starts after the previous assigned trip then
            break
```

%% Part 2 minimize charging and replace charge spots in between a gap to two trips;
while *the SoC is above* SoC_{min} *at the end of the* v **do**
    Find last charging spot by finding $\sigma = 1$;
    Empty ϵ in the current time-block ;
    if $\epsilon = 0$ **then**
        $\sigma = 0$ for this the current time-block;
    Update the SoC of the v;
% Relocate charging sessions **if** *there is any* $\sigma = 1$ *AND* $\theta > 0$ **then**
    while *there are any* $\sigma = 1$ *AND* $\theta > 0$ **do**
        Make a list L_σ of all indexes of $\sigma = 1$ AND $\theta > 0$ ordered from a high value for θ to a low value of θ;
        for $l_\sigma \in L_\sigma$ **do**
            Find all $\sigma = 0$ inbetween two assigned trips, which are before the charging session of l_σ ;
            for *all* $\sigma = 0$ *inbetween two assigned trips* **do**
                if $\sigma_{new} < \sigma_{l_\sigma}$ **then**
                    if *the energy level does not exceed e_b_max of the bus* **then**
                        $\sigma_{new} = \sigma_{l_\sigma}$;
                        $\epsilon_{new} = \epsilon_{l_\sigma}$;
                        $\sigma_{l_\sigma} = 0$;
                        $\epsilon_{l_\sigma} = 0$;
                        break ;
%% Part 3 Place charging session back on more favourable positions and delete the trips planned on these positions ;
if *there is any* $\sigma = 1$ *AND* $\theta > 0$ **then**
    while *there is any* $\sigma = 1$ *AND* $\theta > 0$ **do**
        Make a list L_σ of all indexes of $\sigma = 1$ AND $\theta > 0$ ordered from a high value for θ to a low value oof θ;
        for $l_\sigma \in L_\sigma$ **do**
            $i = 1$; **while** *there are assigned trips that are driven before the charging session* l_σ **do**
                Determine the time interval of the assigned i number of trips before the charging session l_σ ;
                if *the time interval of i trips is larger than the charging session* **then**
                    if *deleting these trips and placing the charging session on that spot is cheaper than the current solution* **then**
                        Delete the i trips;
                        Delete the old charging session;
                        Add the new charging session;
                        break;
                $i = i + 1$;

In the algorithm is π_t the dual variable associated with a trip, σ is whether a time-block is used for charging or not, ϵ is the amount of energy charged in a time-block, θ is the dual variable associated with the costs for σ and l_σ is a list that contains all σ, which are equal to one.

B.3 Genetic algorithm

In this section the genetic algorithm of Chapter 3 is given. The algorithm is explained in section 3.2.3. In this section the genetic algorithm is presented.

Algorithm 3: Genetic Algorithm

```
[V_new ] = function Genetic Algorithm(V',π_t,θ and ρ);
Choose number of generations g;
Choose number of tournaments ts;
Choose number of participants ps;
for 1:g do
    Choose a certain number of vehicle tasks v from V';
    Split each chosen v in 4 chromosomes;
    Calculate reduced costs for all chromosomes;
    % selection
    for each of the 4 chromosomes and the chose v do
        for 1:ts do
            for 1:ps do
                Assign a chromosome or a v depending on the tournament to participant p_{ts_ps};
            end
            for 1:ps do
                Create intervals for each participant with the following equation;
                p_{ts_ps} = sum(reduced_cost_{p_{ts_{ps-1}}} + reduced_cost_{p_{ts_ps}})/∑ reduced_costs_{p_ts};
                Choose a random number between 0 and 1;
                Choose p_{ts_ps} which is linked to the interval that captures the random number;
            end
        end
    end
    % start of crossover
    for each of the chromosomes do
        for each p_{ts_win} do
            Pick randomly a vehicle task from all tournament winners ;
            Combine the tournament winner of the chromosome of p_{ts_win} with the vehicle task to
             create v_new;
        end
    end
    Save all v_new to use for the new generations ;
    Save the best v_new to use as final columns ;
    if this is the last generation then
        Calculate the reduced costs ;
        Save all v_new with reduced costs;
        Add the columns V_new to the RMP
    end
end
if two trips are planned, while they are not compatible at the border of transition from chromosome
 to chromsome then
    Delete one trip from the schedule
end
if the SoC of the bus does not meet the constraints then
    Delete the v
end
```

In the algorithm is v a vehicle task, $p_{ts_{ps}}$ is a participant of tournament number ts with participant number ps, g is generation number. This genetic algorithm could be improved significantly, but the main reason for this algorithm is to test the potential of a genetic algorithm. The algorithm appeared to have not much potential, therefore the algorithm has not been improved. The algorithm did not work for two reasons: There are too many infeasible columns and the columns did not have high enough negative reduced costs.

B.4 Diving heuristic

The diving heuristic is the last of three heuristics implemented to solve the subproblem of Wijnheijmer [3]. The results are explained in section 3.2.4. The algorithm is presented below:

Algorithm 4: Diving heuristic for the subproblem

```
[v_new] = function Diving_heuristic_subproblem(π_t,θ,ρ);
Relax the integer constraints in the subproblem. Solve the subproblem with a LP solver ;
while not all δ_t variables have an integer value do
    Find all the integer variables equal to one if there are no new integer variables then
        Find all variables below one ;
        Find the indexes of the variable with the highest value ;
        Fix this variable δ_t to one with constraints in the subproblem ;
    end
    Fix all previous fixed variables δ_t with help of equality constraints;
    Solve the subproblem;
    if solution is infeasible then
        Constraint the previous fixed variable δ_t to zero ;
    end
end
while not all σ_ζ have an integer value do
    Find all σ_ζ equal to one if there are no new integer variables then
        Find all variables below one ;
        Find the indexes of the variable with the highest value ;
        Fix this variable σ_ζ to one with constraints ;
    end
    Fix all fixed variables with help of equality constraints;
    Solve the subproblem;
    if solution is infeasible then
        Constraint the previous fixed variable σ_ζ to zero ;
    end
end
```

B.5 The formulation of the multiple step column generation method

In section 3.3 the method of using a multiple step column generation model is explained. It is stated that the model could not work. In this section an explanation is given why the model does not work. First, the formulation of the RMP of the model is given. The formulation of the RMP is based on the paper of Vance et al. [27]. The formulation is as follows:

$$\text{obj min} \quad \sum_{d\in D} c_d^D u_d^D + \sum_{v\in V} c_v^V u_v^V + \sum_{s\in S} c_c^S u_s^S \tag{B.1}$$

$$\text{subject to} \quad \sum_{d:t\in d} u_d^D = 1 \quad \forall t \in T \tag{B.2}$$

$$\sum_{v:d\in v} u_v^V = u_d^D \quad \forall d \in D \tag{B.3}$$

$$u_{l_{ci}}^{L_{ci}} = u_v^V \quad \forall l_{ci} : v \in L_{ci} \tag{B.4}$$

$$\sum_{s:l_{ci}\in s} u_s^S = u_{l_{ci}}^{L_{ci}} \quad \forall l_{ci} \in L_{ci}, v \in V \tag{B.5}$$

$$u_d \in \{0,1\} \quad \forall d \in D \tag{B.6a}$$

$$u_v \in \{0,1\} \quad \forall v \in V \tag{B.6b}$$

$$u_{l_{ci}} \in \{0,1\} \quad \forall l_{ci} \in L_{ci} \tag{B.6c}$$

$$u_s \in \{0,1\} \quad \forall s \in S, \tag{B.6d}$$

where c_d are the costs for a shift, u_d is 0 or 1 depending on if shift d is used, c_v are the costs for a vehicle task, u_v is 0 or 1 depending on, if vehicle task v is used, c_c are the costs per charger, u_s is 0 or 1 depending on if the charger task s is used or not and l_{ci} are the charger intervals.

The objective function is to minimize the costs of the shifts, the costs of the vehicle tasks and the costs of the charger tasks. A vehicle task is a day task for a bus and a charger task is a day task for a charger. The first constraint (B.1) states that each trip has to be present once in a chosen shift. The second constraint (B.2) states that each duty has to be assigned to one vehicle task. The third constraint (B.3) couples every charge interval to a vehicle task interval. Charge intervals are time intervals between two shifts in a vehicle task. The fourth constraint (B.4) states that each charge interval has to present one in a charger task.

To simplify the model the costs for the shifts can be left out for now. Constraints (B.1) and (B.2) can be combined. Constraint (B.3) and constraint (B.4) can also be combined. To reduce the size of the formulation a block structure is used. The constraint can be written as follows:

$$\begin{bmatrix} A_{TD} & 0 & 0 & 0 \\ -I_D & A_{DV} & 0 & 0 \\ 0 & A_{LV} & -I_L & 0 \\ 0 & 0 & -I_L & A_{LS} \end{bmatrix} \begin{bmatrix} u_d^D \\ u_v^V \\ u_l^{L_{ci}} \\ u_s^S \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0. \end{bmatrix} \tag{B.7}$$

The formulation can be made smaller by substitution:

$$\begin{bmatrix} A_{TD}A_{DV} & 0 \\ A_{LV} & -A_{LS} \end{bmatrix} \begin{bmatrix} u_v^V \\ u_s^S \end{bmatrix} = \begin{bmatrix} 1 \\ 0. \end{bmatrix} \tag{B.8}$$

The new RMP would be as follows:

$$\text{obj min} \quad \sum_{v \in V} c_v^V u^V + \sum_{s \in S} c_s u^S \tag{B.9}$$

$$\text{subject to} \quad A_{TD}A_{DVi}u^V = 1 \quad \forall t \in T \tag{B.10}$$

$$A_{LV}u^V = A_{LS}u^s \quad \forall l_{ci} \in L_{ci} \tag{B.11}$$

$$0 \leq u_v^V \leq 1 \quad \forall v \in V \tag{B.12a}$$

$$0 \leq u_s^S \leq 1 \quad \forall s \in S. \tag{B.12b}$$

The dual is created to find the dual variables:

$$\text{obj max} \quad \pi_t \tag{B.13}$$

$$\sum_{t \in T} (A_{T_D}A_{DV})^T \pi_t - \sum_{l_{ci} \in L_{ci}} A_{LV}^T \pi_{l_{ci}} \leq c_v \quad v \in V \tag{B.14}$$

$$- \sum_{l_{ci} \in L_{ci}} A_{LS}\pi_{l_{ci}} \leq c_s \quad s \in S \tag{B.15}$$

$$\pi_t \geq 0 \quad \forall t \in T \tag{B.16a}$$

$$\pi_{l_{ci}} \geq 0 \quad \forall l_{ci} \in L_{ci}, \tag{B.16b}$$

where π_t is the dual variable of constraint B.9, $\pi_{l_{ci}}$ is the dual variable of constraint (B.10). There is a dual variable π_t for each trip, where $\pi_{l_{ci}}$ gives a dual variable per charge interval. The new column would include a vehicle task and a charger task. The reduced costs for such a column is as follows:

$$c_v - \sum_{t \in T} \delta_t \pi_t + \sum_{l_{ci} \in L_{ci}} \delta_c \pi_{l_{ci}}, \tag{B.17}$$

where δ_c is 1 if a charging interval is taken in the new charging task and δ_t is 1 if a trip is assigned to the new vehicle task. The objective should be less than zero, to obtain a new column that improves the objective function. The first subproblem has the reduced costs (B.17) as objective function. In the subproblem a vehicle task and charger task is chosen to be sent as a column to the RMP. The charger task should be created on the basis of the vehicle tasks already present in the RMP and the newly chosen vehicle task in the subsubproblem. Vehicle tasks contain namely the charger intervals that have to be assigned to a charger task. A dual variable $\pi_{l_{ci}}$ is connected to each charger interval. The newly chosen vehicle task however has been made in the subsubproblem and can contain a new charging interval l_{ci}, which has not been priced yet by the RMP. The new vehicle task can thus not be taken into account in the new charger task. The new vehicle task first has to be added to the RMP before a dual variable $\pi_{l_{ci}}$ can be found for to the new charging interval l_{ci}. This is one of the problems this model had, next to the other problems. The problem could be solved by creating the vehicle tasks and charger tasks in parallel instead of in series. Thus, one subproblem creates a vehicle task and another subproblem creates the charger task. It has been however decided to not implement this option. The main reason is the complexity of the model, the possible large loss in quality of the solution due to using multiple (relaxed) subproblems and it is difficult to implement deadhead trips.

Appendix C

Charger connection time and Non-linear charging

This section describes the implementation of the charger connection time in the label-correcting algorithm explained in section 3.4 and section 4.4.2. The second part of the section explains how a non-linear charging rate can be implemented. At the moment the charging rate is still linear.

It takes time to connect the charger to the bus. In most cases this takes one minute. This has to be taken into account. This constraint is difficult to implement in the label-correcting algorithm. It can be easy implemented by adding one extra minute to the time of all deadhead trips from trip nodes to charger nodes.

One of the requirements of VDL (1.2) that is not implemented is the requirement that the charging rate of the chargers should follow a non-linear charging curve instead of a linear arc. At the moment e_{charge} is a constant value per charger location and thus follows a linear charging curve. The non-linear charging curve can be approximated by piecewise linear functions. Each piecewise linear function describes e_{charge} per interval, where the interval is determined based on the % SoC in the battery. For example from 5% SoC until 10% SoC the value for e_{charge} is 1% SoC per minute and for 10% SoC until 15% SoC the value for e_{charge} is 1.1% SoC per minute. The SoC of a path is known before it takes a charger to charger arc (i.e., a charging session). The SoC at the beginning of a charging sessions can determine, which linear charging curve has to be used to determine e_{charge}. e_{charge} can stay constant during a charging session, since most charging sessions are small. In this way an approximation can be made from the non-linear charging curve. An approximation of the non-linear charge curve is not implemented in the current model due to limited time.

Appendix D

MILP formulation for the subproblem

The subproblem of the RMP is solved in section 6.5 with a MILP solver. The subproblem has to be written in a MILP formulation. This MILP formulation is shown here:

$$\text{obj min} \quad c_v - \sum_{t \in T} \pi_t a_{i,j} + \sum_{p_{c_l} \in P_{c_l}, c_l \in C_l} \pi_{p_{c_l}} r_{pcl}, \tag{D.1}$$

$$\sum a_{N_{start}j} = 1 \tag{D.2}$$

$$\sum a_{jN_{end}} = 1 \tag{D.3}$$

$$\sum_{i \in I} a_{i,j} = \sum_{I \in I} a_{j,i} \quad \forall j \in N_t + 1, j \in N_{p_{cl}} - 1 \tag{D.4}$$

$$SoC_{N_{start}} = SoC_{max} \tag{D.5}$$

$$SoC_{N_t} \leq SoC_{N_{q_t}} - e_t a_{N_{q_t},N_t} + 100(1 - a_{N_{q_t},N_t}) \quad \forall t \in T, q_t \in Q_t \tag{D.6}$$

$$SoC_{N_{p_{cl}}} \leq SoC_{N_{m_t}} + 100(1 - a_{N_{m_{pcl}},N_{p_{cl}}}) \quad \forall p_{cl} \in p_{cl}, c_l \in C_l, m_{pcl} \in M_{pcl} \tag{D.7}$$

$$SoC_{N_{p_{cl}}} \leq SoC_{N_{p-1_{cl}}} + r_{pcl} e_{charge_{cl}} dt_{pcl-p-1_{cl}} \quad \forall p_{cl} \in P_{cl}, c_l \in C_l \tag{D.8}$$

$$a_{p-1cl,cl} \geq r_{pcl} \quad \forall p_{cl} \in P_{cl}, c_l \in C_l \tag{D.9}$$

$$a_{i,j} = \{0, 1\} \quad \forall i \in N_t, i \in N_{p_{cl}}, j \in 1 + N_t, j \in N_{p_{cl}} \tag{D.10a}$$

$$SoC_{min} \leq SoC_i \leq SoC_{max} \quad \forall i \in N \tag{D.10b}$$

$$0 \leq r_{pcl} \leq 1 \quad \forall p_{cl} \in P_{cl}, c_l \in C_l, \tag{D.10c}$$

where SoC_i keeps track of the SoC of a path at node i, N are all nodes, Q_t are all nodes that an outgoing arc towards trip t, M_{pcl} are all trips nodes that have an outgoing arc to cl.

The objective function (D.1) is the reduced costs function (4.15). The first constraint (D.2) states that only one outgoing arc is allowed from the starting node of the model. The second constraint (D.3) states that only one ingoing arc is allowed from the end node of the model. The third constraint (D.4) states that all other arcs must have one ingoing and one outgoing arc. The next constraint (D.5) states that the SoC of the path at the starting node is equal to the maximum SoC. The fifth constraint (D.6) states that the SoC at a trip node has to be lower than the energy costs of the trip plus the SoC of the node m which has an outgoing arc to that node. Constraint (D.7) states that the SoC at a charger node has to be lower than the SoC of all trip nodes it has an ingoing arc from. Constraint (D.8) states that the SoC of a charger node is lower or equal to the SoC in the previous charger node plus the SoC added by the charging session. Constraint (D.9) states that a charging session can only be used, when the linked charger to charger arc is one. Constraint (D.10a) states that an arc can only be one or zero. The constraint (D.10b) states that the SoC of at each node has to be higher or equal than the minimum SoC and lower or equal to the maximum SoC. The last constraint states that 0% until 100% can be used of a charging session.

Appendix E

Gantt charts of schedules of Le Havre and Eindhoven

In this appendix the Gantt charts are shown of the schedules simulated in section 6.1. First the graph of Le Havre is given in Figure E.1. The blue dots are the depot nodes, the red nodes are the charger

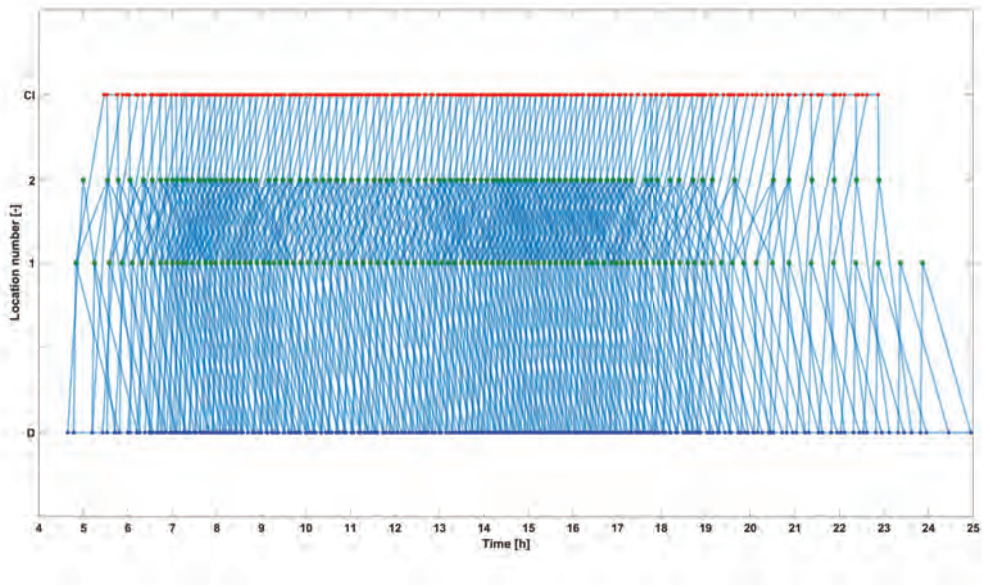


Figure E.1: *Graph: Timetable Le Havre*

nodes and the green nodes are the trip nodes. The lowest line of blue dots are the depot nodes. The Gantt chart of Le Havre is given in Figure E.2. The Gantt chart of Eindhoven is shown in Figure E.3.

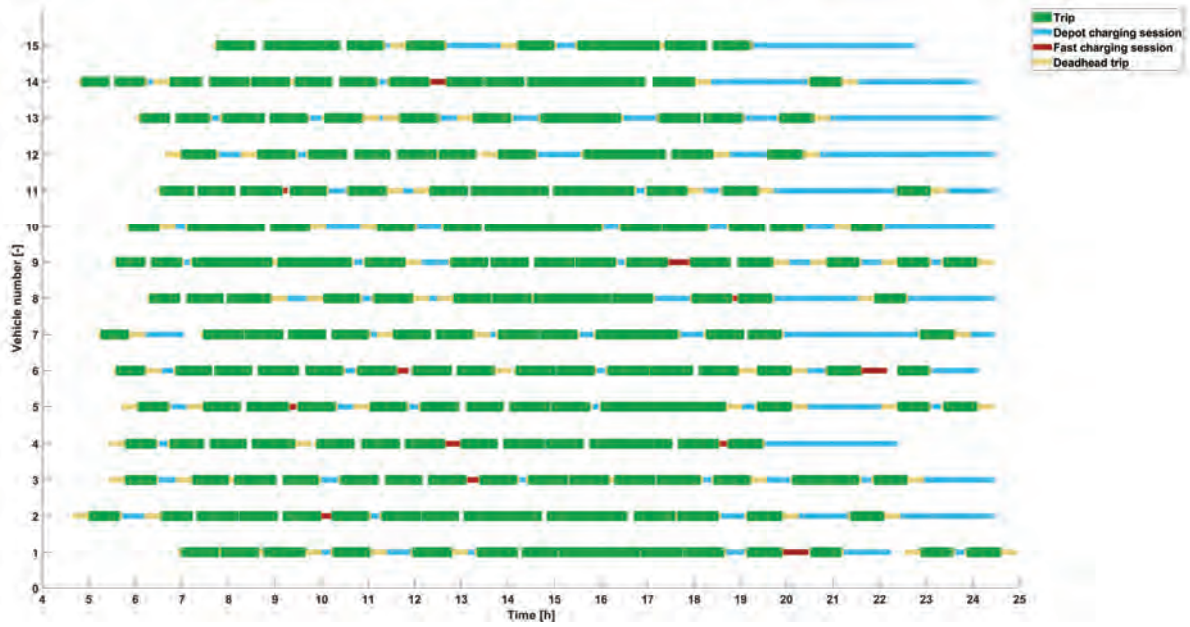


Figure E.2: *Gantt chart: Timetable Le Havre*

Figure E.3: *Gantt chart: Timetable Eindhoven*

Appendix F

MATLAB scripts

In this appendix the code is provided that is used to obtain the results in Chapter 6. The code is based on the model described in Chapter 4 and Chapter 5. For each file a short explanation is given on what the file does.

Main file

The main file is the file, where the input has to be given. This is done with a excel sheet and partly by manually filling in the input. The goal of the main file is to transfer information between the different functions. These functions are creation of the graph, the initiation of the model (i.e., the warm start for shifts), (the creation of the constraints for the subproblem for the MILP solver) the column generation method and the results.

```
%% VDL ETS - Electric fleet Scheduler
% Author: Xander Ouwerkerk
% Date: 1-02-2021



clear all; close all; clc;

%% Input Excel file
TimeTable_name  = ['b']; % Abbreviation of the name
excelfile       = ['TimeTable_',TimeTable_name,'.xlsx']; % excel file
[TimeTable, headers_t] = xlsread(excelfile,'TimeTable');
[All_Locations, headers_l] = xlsread(excelfile,'All_Locations');
%% Extract information from Excel file
idx1            = find(strcmpi(headers_t(1,:),'Dist'))-1;
Dist_t          = TimeTable(:,idx1);   %Distence driven trip
idx2            = find(strcmpi(headers_t(1,:),'Start'))-1;
ht_start        = TimeTable(:,idx2);   %Begin time of a trip
idx3            = find(strcmpi(headers_t(1,:),'End'))-1;
ht_end          = TimeTable(:,idx3);   %End time of trip
l_start2        = headers_t(2:end,1); %Name start location
l_end2          = headers_t(2:end,4); %Name end location
Dist_d          = xlsread(excelfile,'d'); % distance deadhead trip
h               = xlsread(excelfile,'t'); % Deadhead trip tme

% Convert time
ht_start    = ceil(ht_start*24*60);
ht_end      = ceil(ht_end*24*60);
h           = ceil(h); % Already in minutes

% Determine number of trips
n_t = size(ht_start,1);

% From location name to a number
loc_start = zeros(1,n_t);
loc_end = zeros(1,n_t);
for ii = 1:n_t
    loc_start(ii) = find(strcmp(strtrim(l_start2{ii}),strtrim(headers_l(2:end,1))) ≠ 0);
```

```
39      loc_end(ii)    = find(strcmp(strtrim(l_end2{ii}),strtrim(headers_l(2:end,1))) ~= 0);
40  end
41
42  % Determine number of charging locations
43  nr_loc_charg = size(h,1) - max(loc_start);
44  % Assumed that every location is used
45  % Give location numbers to charging locations
46  if isempty(nr_loc_charg) == 0
47      for xx = 1:nr_loc_charg
48          loc_charge(xx) = max(loc_start) + xx;
49      end
50  end
51  %% Manual Input
52  % energy variables
53  e_max           = 420;                  % Battery capacity (kWh)
54  e_charge(1)     = 50/60/(e_max/100);    % Charge rate Depot (SoC %)
55  for xx = 2:nr_loc_charg                 % For each fast charging location
56  e_charge(xx)    = 330/60/(e_max/100);
57  end
58  SoC_min         = 20;                   % Minumum allowable (SoC %)
59  SoC_max         = 90;                   % Maximum allowable (SoC %)
60  C_cl(1)         = 0;                    % Costs slow charger (0≤c≤1)
61  for xx = 2:nr_loc_charg
62      C_cl(xx) = 0.2;                     % Costs fast chargers(0≤c1≤1)
63  end
64  c_v             = 1;                    % Costs per bus (0<c_b ≤1)
65  min_s_t         = 0;                    % minimum shift time (min)
66  min_c_t         = 5;                    % minimum charge time (min)
67  depot           = 1;                    % number of chargers types at depot
68  %1 is for the slow charger (default and recommended for computational purposes)
69  %2 would mean a fast charger on the depot
70  % Simulation parameters
71  max_columns                = 1500;   % Maximum allowable columns in the active column ...
        pool (default 1500)
72  max_deadhead_time          = 10;     % Maximum deadhead trip time to trips (default 10 min)
73  max_idle_time              = 10;     % Maximum allowable idle time (default 10 min)
74  max_deadhead_time_charger = 10;      % Maximum allowable deadhead trip to charger ...
        (default 10 min)
75  maxiter                    = 350000; % Maximum iterations for RMP (default infinite)
76  nr_prev_val                = 50;     % Number of increase in obj function over an x ...
        amount of iterations
77  opts = optiset('solver','clp','maxiter',1000000,'maxtime',100000);  % Settings LP solver
78  opts2 = optiset('solver','cbc','maxiter',1000000,'maxtime',100000); % Settings MILP solver
79
80
81  %% Information red out of the excel file
82  % Energy consumption
83  e_d             = 1.5;               % Energy consumption [kWh/km]
84  e_t             = (Dist_t/1000)*e_d; % Energy consumption trips[kWh]
85  e_h             = (Dist_d/1000)*e_d; % Energy consumption deadhead trips[kWh]
86  e_t             = e_t/(e_max/100);   % SoC consumption per trip
87  e_h             = e_h/(e_max/100);   % Soc consumption per deadhead trip
88
89
90  %% Create graph + obtaing information of the graph + create list
91  [x,A_list_c,A_list_r,comp,charge_line_all,h_s,h_e,charge_line_2,pcl...
92      ,dt_2,dt_all,x_loc,x_loc_trip,x_loc_charger...
93      ,x_loc2,x_loc_size,all_loc,all_loc2,s_loc,s_loc_trip,s_loc_charger...
94      ,s_loc2,s_loc_size,s_loc_c_tot,x_loc_tot,...
95      time_nodes,A_list_c_sub,A_list_r_sub,G,loc_X,...
96      loc_Y,n_d,n_lc,I,graphplot]...
97      = Graph(n_t,ht_start,ht_end,loc_start,...
98      loc_end,h,nr_loc_charg,loc_charge...
99      ,max_deadhead_time,max_idle_time,max_deadhead_time_charger);
100
101 %% Inital solution + warm start of shifts
102 [V] = Initialization(n_d,n_t,A_list_c,A_list_r,pcl,...
103     SoC_max,SoC_min,e_t,graphplot...
104     ,comp,ht_start,ht_end,loc_end,loc_start,h,e_h,loc_charge,time_nodes);
105
106 %% Column generation
107 % Inital variables needed
108 rounding = 1;
```

```
rounding_sub = 1;
fxd_variables_cg = []; % array to fix variables to one
Obj_val_prev(1:nr_prev_val) = 0; % nr_prev_val previous iterations
V_prev = 0; % previous column created
V_Store = zeros(size(V,1),1); % V back up storage
V_final =[]; % final solution
no_column_management = 1; % after rounding no column management
R_nr = 1;
iter(R_nr) = 0;


%% Initialization for using the Diving heuristic

left_t_final(1:n_t) = 0;
left_d_final(1:pcl(2)) = 0;
left_npcl_final(1:pcl(end)) = 0;% left_c is the number of chargers used
b2_fix(1:pcl(end),1) = 0; % initialize b2_fix
b1_fix(1:n_t,1) = 1;

A_sub = [];b_sub =[];Aeq_sub = [];beq_sub=[];lb_sub=[];ub_sub=[];prob2=[];
% Constraints subproblem LP solver
% [A_sub,b_sub,beq_sub,Aeq_sub,lb_sub,ub_sub,prob2]...
% = ConstraintsSubproblem(n_t,...
% n_lc,n_d,all_loc,all_loc2,x_loc,A_list_r,...
%     A_list_c,e_t,s_loc_trip,s_loc_charger,...
%     dt_2,e_charge,min_c_t,pcl,SoC_max,SoC_min,nr_loc_charg);

%% CG + Rounding
[V_final, iter, RMP_Obj_val,b2_fix,V,uv] = ...
    Column_generation(x,A_list_c,A_list_r,pcl...
    ,dt_2,x_loc,A_list_c_sub,A_list_r_sub,n_d,n_lc,V,...
    opts,opts2,rounding,rounding_sub,fxd_variables_cg,nr_prev_val,...
    min_c_t,min_s_t,Obj_val_prev,maxiter,V_Store,max_columns,...
    V_final,e_h,I,no_column_management,left_npcl_final,...
    left_d_final,left_t_final,R_nr,iter,...
    n_t,nr_loc_charg,A_sub,b_sub,beq_sub,Aeq_sub,lb_sub,ub_sub...
    ,c_v,C_cl,b1_fix,b2_fix,e_t,ht_start,ht_end,SoC_min,SoC_max,...
    e_charge,loc_charge,loc_start,loc_end,prob2,depot);

%% Save variables
name = ['TimeTable_',TimeTable_name];
save(name ,'V_final','x_loc','h_e','h_s','charge_line_2',...
    'dt_2','A_list_c','A_list_r','ht_end','ht_start','n_t',...
    'pcl','h','loc_charge','loc_start','loc_end','nr_loc_charg','iter',...
    'e_h','SoC_max','SoC_min','c_v','C_cl','RMP_Obj_val','b2_fix',...
    'n_d','charge_line_all','e_t',...
    'e_charge','e_max','graphplot','time_nodes');
%% Results
[buses_used,Costs,percentage_double,trips_double,...
    total_energy,double_deadhead,integrality_gap,bus_min]...
    = Results(V_final,x_loc,h_e,h_s,charge_line_2,...
    dt_2,A_list_c,A_list_r,ht_end,ht_start,n_t,...
    pcl,h,loc_charge,loc_start,loc_end,nr_loc_charg, iter,...
    e_h,SoC_max,SoC_min,c_v,C_cl,RMP_Obj_val,b2_fix,...
    n_d,charge_line_all,e_t,e_charge,e_max,graphplot,time_nodes);
```

Creation of the graph

In this function the graph is created. First the compatibility matrix is created to determine, if trips can be driven by the same bus or not. Then the charger nodes are determined, based on the start and end times of the trips. The arcs between the nodes are created after this step. All arcs are stored in a matrix x. A list is created from this matrix. The list is ordered based on start time. The list is split into two lists: *A_list_r* and *A_list_c*. *A_list_r* contains all nodes with the outgoing arcs to the nodes in *A_list_c*. As last the graph is plotted.

```
function [x,A_list_c,A_list_r,comp,...
    charge_line_all,h_s,h_e,charge_line_2,pcl...
    ,dt_line_2,dt_all,x_loc,x_loc_trip,x_loc_charger...
```

```
    ,x_loc2,x_loc_size,all_loc,all_loc2,s_loc,s_loc_trip,s_loc_charger...
    ,s_loc2,s_loc_size,s_loc_c_tot,x_loc_tot,...
    time_nodes,A_list_c_sub,A_list_r_sub,G,loc_X,...
    loc_Y,n_d,n_lc,I,graphplot]...
    = Graph(n_t,ht_start,ht_end,...
    loc_start,loc_end,h,nr_loc_charg,l_charge...
    ,max_deadhead_time,max_idle_time,max_deadhead_time_charger)
%% Create graph
%% Compatibility matrix -> if two trips can be driven in sequence
comp  = zeros(n_t);
comp(1:n_t,1:n_t) = 1;
for i = 1:n_t
    for j = i+1:n_t
        if ht_end(i) + h(loc_end(i),loc_start(j)) ≤ ht_start(j)  % time between two ...
            trips  + h(l_end(i),l_start(j))
            if h(loc_end(i),loc_start(j)) < max_deadhead_time % if the deadhead trip ...
                is smaller than 15 minutes
                if ht_end(i)+ h(loc_end(i),loc_start(j)) + max_idle_time ≥ ht_start(j) ...
                    % + h(l_end(i),l_start(j)) no longer than standing still 5min
                    comp(i,j) = 0;
                end
            end
        end
    end
end
comp = comp - tril(comp);

%% create charge nodes
charge_line_all = []; % time of charge nodes
for xx = 1:nr_loc_charg
    % update start time charge nodes with deadhead trips fo each charger
    ind = sub2ind(size(h),repmat(l_charge(xx),1,numel(loc_start)),loc_start);
    h_s{xx} = ht_start-h(ind)'; % start time with deadhead trip taken into account
    ind2 = sub2ind(size(h),loc_end,repmat(l_charge(xx),1,numel(loc_end)));
    h_e{xx} = ht_end+h(ind2)'; % end time with deadhead trip taken into account
    charge_line_2{xx} = sort(unique([h_s{xx} h_e{xx}]))'; % Sort time, only unique values
    % only unique charge nodes, so no charge nodes with same time
    if xx > 1 % for non depot chargers
        del1 = find(charge_line_2{xx}≤charge_line_2{1}(1)); % find all nodes before ...
            the first depot node
        del2 = find(charge_line_2{xx}≥charge_line_2{1}(end)); % find all nodes before ...
            the first depot node
        delh_e1 = find(h_e{xx}≥h_e{1}(end))'; % find every end point after last depot node
        delh_s1 = find(h_s{xx}≤h_s{1}(1))';% find every  point before first depot node
        delete = [del1 del2];
        delete_h = [delh_e1 delh_s1];
        charge_line_2{xx}(delete) = []; % delete all nodes before start end time of ...
            depot nodes
        h_e{xx}(delete_h) = []; % delete these nodes also from the time list
        h_s{xx}(delete_h) = [];
    end
    % note that start and end node are not needed sometimes, due to the max deadhead time
    % these nodes are not deleted due to time
end
charge_line = charge_line_2{1};  %unique charging nodes


%% Create matrix A to describe graph
x = zeros(n_t+size(charge_line,2)); % create A matrix

% Arcs between trip
for ii = 1:n_t
    for jj = ii+1:n_t
        if comp(ii,jj) == 0 % if trips are time compatible
            x(ii,jj) = 1;
        end
    end
end
% Create lines chargers
cc = 0;
for jj = 2:nr_loc_charg+1
    for ii = 1:size(charge_line_2{jj-1},2)
        x(n_t+cc+1,n_t+cc+2) = 1;
```

```
71          cc = cc + 1;
72          pcl(jj) = cc; % Number of charging sessions
73      end
74      x(n_t+cc+1-1,n_t+cc+2-1) = 0; % to get subdivision between charger locs
75  end
76  x(:,end) =[]; % delete last column
77  % create lines chargers trips
78
79  % from chargers to trips arcs
80  for ii = n_t:-1:1
81      for kk = 2:nr_loc_charg+1
82          for jj = n_t+pcl(kk)-pcl(kk-1):-1:n_t+1
83              if charge_line_2{kk-1}(jj-(n_t)) + h(l_charge(kk-1),loc_start(ii)) ≤ ...
                    ht_start(ii)
84                  % Determine which charger node one arc is drawn to a trip node
85                  if kk == 2 % for depot
86                      x(jj,ii) = 1;
87                      break % Break to prevent multiple arcs to the trip node
88                  else
89                      if h(l_charge(kk-1),loc_start(ii)) ≤ max_deadhead_time_charger
90                          % Limit max deadhead trip time implied
91                          x(jj+pcl(kk-1),ii) = 1;
92                          break
93                      else
94                          break
95                      end
96                  end
97              end
98          end
99      end
100 end
101 % to chargers from trips
102 for ii = 1:n_t
103     for kk = 2:nr_loc_charg+1
104         for jj = n_t+1:n_t+pcl(kk)-pcl(kk-1)
105             if ht_end(ii) + h(loc_end(ii),l_charge(kk-1)) ≤ ...
                    charge_line_2{kk-1}(jj-(n_t)) % needs to be changed ≤ -> <
106                 % Determine from which charger node an arc is drawn to a
107                 % trip node
108                 if kk == 2
109                     x(ii,jj) = 1;
110                     break; % prevent multiple arcs to one trip node
111                 else
112                     if h(loc_end(ii),l_charge(kk-1)) ≤ max_deadhead_time_charger
113                         x(ii,jj+pcl(kk-1)) = 1; %-1
114                         break; % prevent multiple arcs to one trip node
115                     else
116                         break
117                     end
118                 end
119             end
120         end
121     end
122 end
123 % all charge nodes that are not connected to one end point and one start
124 % point delete
125 charge_line_all = [charge_line_2{1}];
126 for kk = 3:nr_loc_charg+1 % For all fast charger locations
127     idx_del = 0;
128     % Start Sweep (begin of the schedule)
129     for jj = n_t+1:n_t+pcl(kk)-pcl(kk-1)
130         jj = jj - idx_del;
131         if isempty(find(x(:,jj+pcl(kk-1)))) || isempty(find(x(jj+pcl(kk-1),:)))
132             %  if one row is completely empty is completely empty in the
133             %  matrix A_list
134             x(jj+pcl(kk-1),:) = []; % delete this charger node
135             x(:,jj+pcl(kk-1)) = []; % delete this charger node
136             charge_line_2{kk-1}(jj-(n_t)) = []; % delete charger node
137             pcl(kk:end) = pcl(kk:end) - 1; % update c_t
138             idx_del = idx_del+1; % to determine deleted rows columns
139
140         end
141     end
```

```
142
143     %sweep the otherway around (end of the schedule)
144     for jj = n_t+pcl(kk)-pcl(kk-1):-1:n_t+1
145         if isempty(find(x(:,jj+pcl(kk-1)))) || isempty(find(x(jj+pcl(kk-1),:)))
146             % if one row is completely empty
147             x(jj+pcl(kk-1),:) = [];
148             x(:,jj+pcl(kk-1)) = [];
149             charge_line_2{kk-1}(jj-(n_t)) = []; % delete charger node
150             pcl(kk:end) = pcl(kk:end) - 1; % update c_t
151
152         end
153     end
154     charge_line_all = [charge_line_all charge_line_2{xx}];
155 end
156
157
158
159 %% Collect information of the graph
160
161 %% Important: List used throughout the script
162 [A_list_r,A_list_c] = find(x);
163 % Create two arrays, one with all starting nodes and one with all ending
164 % nodes
165
166 n_lc = pcl(end)-nr_loc_charg; % number of charging intervals
167 for ii = 2:size(charge_line,2)
168     dt_line_2{1}(ii-1) = charge_line(ii) - charge_line(ii-1);
169 end % charge intervals
170 diff_charge_line = dt_line_2{1}(1:end);
171 dt_all = dt_line_2{1}(1:end)';
172 for ii = 2:nr_loc_charg
173     for jj = 2:size(charge_line_2{ii},2)
174         dt_line_2{ii}(jj-1) = charge_line_2{ii}(jj) - charge_line_2{ii}(jj-1);
175         % adjusted for non depot
176     end
177     dt_all = [dt_all;dt_line_2{ii}(1:end)'];
178 end
179
180
181 n_d = size(find(x),1);
182 x_loc_size = 0;
183 for ii = 1:n_t
184     x_loc{ii} =  find(A_list_c == ii);  % find all location to a trip node
185     x_loc_trip{ii} = x_loc{ii}(A_list_r(x_loc{ii})≤ n_t);% find trip to trip
186     x_loc_charger{ii} = x_loc{ii}(A_list_r(x_loc{ii}) > n_t); % find charger to trip
187     x_loc2{ii} =  find(A_list_r == ii);  % find all location from a trip node
188     x_loc_size = x_loc_size+numel(x_loc{ii});
189     all_loc{ii} = x_loc{ii}; % all locations to a node
190     all_loc2{ii} = x_loc2{ii}; % all location from a node;
191 end
192 s_loc_size = 0;
193 for ii = n_t+1:pcl(end)+n_t
194     s_loc{ii} =  find(A_list_c == ii); % find all location to a charge node
195     s_loc_trip{ii} = s_loc{ii}(A_list_r(s_loc{ii})≤ n_t); % find trip to charger
196     s_loc_charger{ii} = s_loc{ii}(A_list_r(s_loc{ii}) > n_t); % find charger to ...
            charger 9->10
197     s_loc2{ii} =  find(A_list_r == ii); % find all location from a charge node
198     s_loc_size =  s_loc_size+numel(s_loc{ii});
199     all_loc{ii} = s_loc{ii}; % all locations to a node
200     all_loc2{ii} = s_loc2{ii}; % all location from a node;
201 end
202 s_loc_c_tot = find(A_list_r > n_t & A_list_c > n_t); % find all locations charger to ...
        charger
203 x_loc_tot = find(A_list_c ≤ n_t); % find all locations to a trip node
204
205 %% Create list for label-correcting algorithm
206 time_nodes = [ht_end];
207 for xx = 1: nr_loc_charg
208     time_nodes = [time_nodes;charge_line_2{xx}']; % all possible time nodes
209 end
210
211 % Create list for subproblem
212 % list is ordered here
```

```
t1 = A_list_r ≤ n_t; % trips
t2 = A_list_r > n_t & A_list_r ≤ pcl(2); % depot
t3 = A_list_r > pcl(2); % charger
xx(t1,1) = 1; % determine secondary order
xx(t2,1) = 3;
xx(t3,1) = 2;
yy = [time_nodes(A_list_r)];
S = [xx, yy];
[¬, I] = sortrows(S, [2 1]);

A_list_c_sub = A_list_c(I);
%in the subproblem a different list is used than in the rest of the model
A_list_r_sub = A_list_r(I);

%% plot the Graph
[s,t] = find(x);
G = digraph(s,t);
figure;
loc_Y = [];
r = 0;
loc_X = [ht_start'];
for xx = 1:nr_loc_charg
    loc_X = [loc_X charge_line_2{xx}];
end
for ii = 1:n_t
    loc_Y = [loc_Y loc_start(ii)];
end
loc_Y = [loc_Y repmat(-1,1,pcl(2))];
for xx = 2:nr_loc_charg
    loc_Y = [loc_Y repmat(l_charge(xx-1),1,pcl(xx+1)-pcl(xx))];
end

graphplot = plot(G,'XDATA',loc_X/60,'YDATA',loc_Y,'LineWidth',3);
highlight(graphplot,1:n_t,'NodeColor',[0 0.5 0],'MarkerSize',7)
highlight(graphplot,n_t+1:n_t+numel(charge_line_2{1}),...
    'NodeColor','b','MarkerSize',5)
highlight(graphplot,n_t+numel(charge_line_2{1})+1:size(loc_X,2),...
    'NodeColor','r','MarkerSize',5)
xlabel('Time [h]','fontweight','bold','fontsize',18)
ylabel('Location number [-]','fontweight','bold','fontsize',18)
axis([floor(min(loc_X)/60) ceil(max(loc_X)/60) -2 max(loc_Y)+1])


set(gca,'xtick',floor(min(loc_X)/60)...
    :1:ceil(max(loc_X)/60),'ytick',1:max(loc_Y))
end
```

Initial solution

The initial solution of one trip per one vehicle task, where every trip is part of one vehicle task each, is created. Then the warm start based on one shift per vehicle task is created. The shifts are created in the function warm_start_shift.

```
function [V] = Initialization(n_d,n_t,A_list_c,A_list_r,pcl,...
    SoC_max,SoC_min,e_t,graphplot...
    ,comp,ht_start,hr_end,loc_end,loc_start,h,e_h,loc_charge,time_nodes)

%% Create inital solution
% option one assign one trip to one bus
V = zeros(n_d,n_t); % Create V' matrix
for ii = 1:n_t % for each tripnode
    for jj = [find(ii == A_list_c)]'
        % find charger node that has an outgoing arc to the trip node
        if n_t ≤ A_list_r(jj) && n_t+pcl(2) > A_list_r(jj)
            % The starting node has to be a depot node
            V(jj,ii) = 1; % add arc to V
            loc_back = [find(ii == A_list_r)]'; % find location back to depot node
            for zz = loc_back % Find arc back to depot line
                if A_list_c(zz) > n_t && n_t+pcl(2) ≥ A_list_c(zz)
```

```
                    V(zz,ii) = 1; % add arc
                    break;
                end
            end
            for yy = [find((A_list_c > n_t) & (n_t+pcl(2) ≥ A_list_c))]'
                % use depot line to complete vehicle task
                if A_list_c(yy) ≠ [(A_list_r(jj)+1):A_list_c(zz)]
                    % depot nodes in the interval of the trip cannot be
                    % taken
                    if A_list_r(yy) ≠ [1:n_t]
                        if A_list_c(yy) ≠ [1:n_t]
                            V(yy,ii) = 1;
                        end
                    end
                end
            end
        end
    end

end


V_w2 =[]; % if option two is off
%% option two: one shift per bus
[Shift] = Warm_start_shift(SoC_max,SoC_min,e_t,n_t,...
    comp,ht_start,hr_end,loc_end,loc_start,h,e_h,loc_charge);
% function to set-partition all trips in shifts
V_w2 = zeros(n_d,size(Shift,2));
% Create initial matrix
for ii = 1:size(Shift,2) % for each tripnode
    Shift{ii} = sort(Shift{ii}); % sort the shifts on time
    for kk = 1:size(Shift{ii},2) % for every trip
        if kk == 1 % if it is the first trip
            for jj = [find(Shift{ii}(kk) == A_list_c)]'
                %  find arc from depot line to trip
                if n_t ≤ A_list_r(jj)&& A_list_r(jj) ≤ n_t+pcl(2)
                    % if arc is to a depot node
                    V_w2(jj,ii) = 1; %% ii is vehicle task indexing
                    loc_begin = A_list_r(jj); % remember depot node
                    break
                end
            end
        end
        if kk < size(Shift{ii},2) % from trip to trip
            loc_next = [find(Shift{ii}(kk) == A_list_r)]';
            % find trip to trip arc
            for bb = loc_next
                if  A_list_c(bb) == Shift{ii}(kk+1)
                    if A_list_c(bb) ≤ n_t
                        V_w2(bb,ii) = 1;
                    end
                end
            end
        end


        if kk == size(Shift{ii},2) % from last trip node back to depot line
            loc_back = [find(Shift{ii}(kk) == A_list_r)]';
            % find arcs from last trip node
            for zz = loc_back
                if A_list_c(zz) > n_t && A_list_c(zz) ≤ n_t+pcl(2)
                    % if depot node
                    V_w2(zz,ii) = 1;
                    loc_end = A_list_c(zz); % return depot node
                end
            end
        end
    end
    % create depot line
    for yy = [find(A_list_c > n_t)]'
        % for all arcs to charger nodes
        if A_list_c(yy) ≠ [loc_begin+1:loc_end]
            % if the arcs are not within the shift interval
```

```
            if A_list_c(yy) > n_t && A_list_c(yy) ≤ n_t+pcl(2)
                if A_list_r(yy) > n_t && A_list_r(yy) ≤ n_t+pcl(2)
                    % if the arcs are on the depot line
                    V_w2(yy,ii) = 1;
                end
            end
        end
    end
end

V = [V V_w2]; % Add warm start to initial solution
V(end+1:end+pcl(end),:) = 0; % add rpcl for chargers that are not used

% plot warm start
% Do not use for large models !
% for jj = 1:size(V_w2,2)
%     [c,r] = find(V_w2(1:n_d,jj));
%
%     [s_c,I] = sort(time_nodes(A_list_c(c)));
%     c = d_var_c(c(I));
%     c(2:end+1) = c;
%     c(1) = n_t+1;
%
%     highlight(graphplot,c,'edgecolor',[rand(1) rand(1) rand(1)],...
%         'LineWidth',3)
% end
```

Creation of shifts

```
function [Shift] = Warm_start_shift(SoC_max,SoC_min,e_t,n_t,...
    comp,h_start,h_end,l_end,l_start,h,e_h,l_charge)
%% input minimum shift size
min_usage = SoC_max - SoC_min-20;
% SoC consumption per shift
comp = triu(comp)'+comp; % mirror
E = SoC_max-SoC_min-min_usage;
% SoC of the bus should stay above E
%% start
Chosen = 0; % already assigned trips
Shift_count = 0; % number of shifts
ft_not_chosen = 0;  % while not every trip is assigned
ii = 0;

% Start algorithm
while Shift_count == 0
    ii = ii + 1; % Add new shift
    energy(ii) = SoC_max - SoC_min; % energy to be used
    Select_ft_s = n_t+1; % Selection first trip new shift
    %Start at last trip
    while ft_not_chosen == 0 % While new trip is not chosen in new shift
        Select_ft_s = Select_ft_s - 1 ; % Update selection trip
        [¬,I] = sort(h_end); % sort of trips list to h_end
        Select_ft = I(Select_ft_s); % Select latest trip
        if Select_ft ≠ Chosen % if trip has not been assigned
            Shift{ii} = [Select_ft]; % create new shift
            Chosen = [Chosen Select_ft]; % Update assigned trip list
            energy(ii) = energy(ii) - e_t(Select_ft)...
                - e_h(l_end(Select_ft),l_charge(1)); % energy update
            break % New trip is assigned to a new shift
        end
    end
    while energy(ii) > E % while energy limit is not reached
        Select = find(h_end + ...
            h(l_end(1:n_t),l_start(Shift{ii}(end)))<h_start(Shift{ii}(end)));
        % possbile trips start later than current trip in shift
        if isempty(Select) == 1 % if no trips are available anymore
            break % End
        end
        [¬,I] = sort(h_end(Select),'descend'); % order trips on end time
        Select_nt = Select(I); % select new trip
```

```
        for new_trip = [Select_nt]'
            if new_trip ≠ Chosen
                % if trip is already assigned to another bus
                if comp(new_trip,Shift{ii}(end)) == 0
                    % compatibility check
                    if energy(ii) - e_t(new_trip)...
                            - e_h(l_end(new_trip),l_start(Shift{ii}(end)))...
                            - e_h(l_charge(1),l_start(new_trip))> 1
                        % - energy trip - deadhead trip - deadhead trip
                        % from depot
                        % energy feasibility check
                        Shift{ii} = [Shift{ii} new_trip]; % Update shift
                        Chosen = [Chosen new_trip]; % Update assigned trips
                        energy(ii) = energy(ii) - e_t(new_trip)...
                            - e_h(l_end(new_trip),l_start(Shift{ii}(end)));
                        % Update energy of shift
                        break
                    end
                end
            end
        end
        if ismember(new_trip,Shift{ii}) == 0
            break
        end
        if new_trip == Select_nt(end)
            break
        end
    end
    if size(Chosen,2) > n_t
        break
    end


end
```

Constraint for the MILP subproblem

In this function the constraints for the MILP subproblem are created. The model-based method of MATLAB is used, since this method is easier to implement code for difficult constraints. This function is only used, when the subproblem is solved with a MILP solver.

```
function [A_sub,b_sub,beq_sub,Aeq_sub,lb_sub,ub_sub,prob2] = ...
    ConstraintsSubproblem(n_t,...
    n_lc,n_d,all_loc,all_loc2,x_loc,A_list_r,...
    A_list_c,e_t,s_loc_trip,s_loc_Charger,...
    dt_2,e_charge,min_c_t,pcl,SoC_max,SoC_min,nr_loc_charg);
%% Constraint problem-based sub problem
% create optimalization problem
prob2 = optimproblem('ObjectiveSense','minimize');

% variables
a = optimvar('Ax',n_d,1,'Type','continuous','LowerBound',0,'UpperBound',1);
S = optimvar('S',n_t+pcl(end),1,'Type','continuous','LowerBound',0,'UpperBound',SoC_max);
r_pcl = optimvar('XC',pcl(end),1,'Type','continuous','LowerBound',0,'UpperBound',1);

% initial matrices
prob2.Constraints.A1_sub = optimconstr(size(n_t,2));
prob2.Constraints.A2_sub = optimconstr(size(n_t,2));
A3_sub = sparse(1,n_d+n_t+pcl(end)+pcl(end));
prob2.Constraints.A4_sub = optimconstr(size(n_t,2));
prob2.Constraints.A5_sub = optimconstr(size(n_t,2));
prob2.Constraints.A6_sub = optimconstr(size(n_t,2));

% constraint 1 inflow = outflow
for ii = 1:n_t+pcl(end)
    prob2.Constraints.A1_sub(ii) = ...
        sum(a(all_loc{ii})) == sum(a(all_loc2{ii}));
end
```

```
28
29 % Constraint 2 start node has SoC = SoC_max
30 A3_sub(1,n_d+1+n_t) = 1;
31 b3_sub(1,1) = SoC_max;
32
33 % Constraint 3 SoC of trips
34 counter = 0;
35 %  all to trip
36 for ii = 1:n_t
37     rr = [];
38     jj = size(x_loc{ii},1);
39     rr(1:jj,1) = ii; % to get the appropiate size
40     dd = counter+1:counter+jj;
41     counter = counter + jj;
42     prob2.Constraints.A2_sub(dd) = ...
43         S(rr) ≤ S(A_list_r(x_loc{ii}))-(e_t(rr).*a(x_loc{ii})) + 100*(1-a(x_loc{ii}));
44 end
45
46 % Constraint 4+5 S for charging spots
47 counter = 0;
48 counter2 = 0;
49 for kk = 2:nr_loc_charg+1 % for all charger locations
50     for ii = n_t+pcl(kk-1)+1:n_t+pcl(kk) % for all charging sessions
51         % trip to charger
52         rr = [];
53         jj = size(s_loc_trip{ii},1);
54         rr(1:jj,1) = ii; % obtain the appropiate size
55         dd = counter+1:counter+jj;
56         counter = counter + jj;
57         if isempty(rr) == 0 % S_end < S_t
58             prob2.Constraints.A4_sub(dd) = ...
59                 S(rr) ≤ S(A_list_r(s_loc_trip{ii})) + 100*(1-a(s_loc_trip{ii}));
60         end
61
62
63         % charger to charger
64         rr = [];
65         jj = size(s_loc_Charger{ii},1);
66         rr(1:jj,1) = ii; % obtain the appropiate size
67         dd = counter+1:counter+jj;
68         counter = counter + jj;
69         if isempty(rr) == 0 % S_end < S_start + e*x for charging spots
70             prob2.Constraints.A4_sub(dd) = ...
71                 S(rr) ≤ S(A_list_r(s_loc_Charger{ii})) + ...
                        (dt_2{kk-1}(ii-n_t-pcl(kk-1)-1)*e_charge(kk-1))....
72                 *r_pcl(A_list_c(s_loc_Charger{ii})-n_t);
73
74
75             counter2 = counter2 + jj;
76             dd = counter2+1:counter2+jj;
77             prob2.Constraints.A6_sub(dd) = S(A_list_r(s_loc_Charger{ii})) +...
78                 (dt_2{kk-1}(ii-n_t-pcl(kk-1)-1)*e_charge(kk-1))....
79                 *r_pcl(A_list_c(s_loc_Charger{ii})-n_t) ≤ SoC_max ;
80             % not 100% necessary constraint
81         end
82     end
83 end
84
85 %% Constraints 6 only charging allowed when the linked arc is taken
86 for ii  = n_t+1:n_t+pcl(end) % only for one charger location capable/ can be changed
87     if isempty(s_loc_Charger{ii}) == 0
88         prob2.Constraints.A5_sub(ii-n_t) = a(s_loc_Charger{ii}) ≥ r_pcl(ii-n_t);
89     end
90 end
91
92
93 %% Storing constraints
94 % transfer model-based -> problem based
95 problem = prob2struct(prob2);
96 Aineq_sub = problem.Aineq;
97 Aeq_sub = problem.Aeq;
98 bineq_sub = problem.bineq;
99 beq_sub = problem.beq;
```

```
beq_sub(n_t+1,1) = -1; %inflow one arc allowed
beq_sub(n_t+pcl(2),1) = 1; % outflow one arc allowed

% lower/upperbound
lb1_sub(1:n_d,1) = 0;
ub1_sub(1:n_d,1) = 1;
lb2_sub(1:n_t+pcl(end),1) = SoC_min;%SoC_min-1e-3; % error margin
ub2_sub(1:n_t+pcl(end),1) = SoC_max; % of label correcting algorithm
lb3_sub(1:pcl(end),1) = 0;
ub3_sub(1:pcl(end),1) = 1;
% merge
Aeq_sub = [Aeq_sub; A3_sub;];
beq_sub = [beq_sub;b3_sub;];

A_sub = [Aineq_sub;];
b_sub = [bineq_sub;];

lb_sub = [lb1_sub;lb2_sub;lb3_sub];
ub_sub = [ub1_sub;ub2_sub;ub3_sub];
end
```

Column generation model

In this function the column generation model is run and the diving heuristic is run. The column generation model consists of multiple parts. The first part is the Rounding_fix_arc.m function, which fixes the rounded up variables in the column generation model. The second part is the column management. This part is run every 5 iterations. It reduces the active column pool. The third part creates the two sets of constraints for the RMP. These are made with the help of array indexing in the solver-based method of MATLAB. The next part consists of solving the LP and writing the dual variables to the correct values. Then the subproblem has to be solved. First, the parameters for the subproblem are changed based on which of the dominance rules are currently active. Then the function Label_correcting_algorithm.m is run, which gives as output the columns created in the label-correcting algorithm and their reduced costs. Then the columns are added to the active column pool, if they meet the conditions. In case none of the columns meet the conditions the for loop is stopped and the function Diving_heuristic.m is run, which rounds a variable. The model is finished, when it has spotted that all trips are assigned to at least one bus. This check is performed after the Rounding_fix_arc.m function has run. Then the Column_generation.m function returns to the main file.

```
function [V_final, iter, RMP_Obj_val,b2_fix,V,uv] = ...
    Column_generation(x,A_list_c,A_list_r,pcl...
    ,diff_charge_line_2,x_loc,A_list_c_sub,A_list_r_sub,n_d,n_lc,V,...
    opts,opts2,rounding,rounding_sub,fxd_variables_cg,nr_prev_val,...
    min_c_t,min_s_t,Obj_val_prev,maxiter,V_Store,max_columns,...
    V_final,e_h,I,no_column_management,left_npcl_final,...
    left_d_final,left_t_final,R_nr,iter,...
    n_t,nr_loc_charg,A_sub,b_sub,beq_sub,Aeq_sub,lb_sub,ub_sub...
    ,c_v,C_cl,b1_fix,b2_fix,e_t,ht_start,ht_end,SoC_min,SoC_max,...
    e_charge,loc_charge,loc_start,loc_end,prob2,depot)
iterations_method1 = 10000; % Number iterations used first dominance rule
iterations_method2 = 10000; % Number iterations used second dominance rule
method             = 1;     % starting method
del_trips          = [];    % initial variable to delete arcs from subproblem

%% Start Column generation model + Diving heuristic
while rounding == 1
    % Start column generation
    for i = 1:maxiter
        iter(R_nr) = iter(R_nr) + 1; % Iterations every rounding round

        n_v = size(V,2);   % Number of vehicle tasks in V
        n_end = size(V,1); % Number of variables in V

        %% Implement results diving heuristic in Column generation model
        if rounding == 1
            fxd = fxd_variables_cg; % Transmit index new fixed variable
            if rounding_sub == 1 % To determine if new variable is fixed
```

```
            else
                if isempty(fxd) == 0 % if there is a fixed variable
                    [V,V_Store,b2_fix,fxd...
                        ,x,A_list_r,A_list_c,n_d,n_t...
                        ,left_npcl_final,left_d_final,...
                        left_t_final,b1_fix,V_final,del_trips]...
                        = Rounding_fix_arcs(...
                        V,V_Store,n_t,n_d,fxd,...
                        A_list_c,A_list_r,b2_fix,x,...
                        left_npcl_final,left_d_final,left_t_final,...
                        pcl,b1_fix,V_final,...
                        A_list_c_sub,A_list_r_sub);
                    % Function to implement diving heurstics results
                    no_column_management = 3;
                    % No column management before RMP is solved again
                    rounding_sub = 1; % no new fixed variable anymore
                    n_v = size(V,2); % update number of columns
                    % Determine if simulation if finished
                    if b1_fix == 0
                        %if there are no more non-assigned trips
                        disp('rounding finished') % model is done
                        rounding = 0; % end while loop
                        break
                    end
                    disp('out')

                end
            end
            fxd_variables_cg = []; % Delete fixed variable
        end

        %% Column management

        if i > 5 && no_column_management == 1
        % Not for the first 5 iterations and not after a variable has been
        % rounded, you want to use column management one iteration after rounding
        % because of the possible warm start
            if rem(i,5)==0 && no_column_management == 1
                % Each 5th iteration column management is used
                if size(V,2) > size(V_new,2)*5
                    % if V is larger than the columns created
                    % in the last 5 iterations
                    V_changed = V(:,1:end-size(V_new,2)*5);
                    % Columns of the last 5 iterations are left out
                    [V_changed,V_Store] = column_mangement(V_changed,...
                     V_Store,c_v,pi_tau,pi_pcl,...
                     n_lc,max_columns,n_d,x_loc,n_t,uv(1:size(V_changed,2)),pcl);
                    % Readd the columns
                    V = [V_changed V(:,end-size(V_new,2)*5+1:end)];
                end
                V = [V V_final];
                n_v = size(V,2); % update number of columns
                no_column_management = 1;
                disp('in')

            end
        end

        if no_column_management > 1
            % if no column management is allowed this iteration
            no_column_management = no_column_management - 1;
        end
        %% Constraints RMP
        % Constraint 1
        A1 = zeros(n_t,n_v+nr_loc_charg);
        % Number of columns + charger locations
        % Create inital matrix constraints
        for ii = 1:n_t % every trip is driven once
            if size(x_loc{ii},1) > 1
                % if there are more than one arcs to drive the trip
                pp = find(any(V(x_loc{ii},:) == 1));
                % find arcs to trip t
            else
```

```
                pp = find(V(x_loc{ii},:) == 1);
            end
            A1(ii,pp) = 1; % update matrix
            %b is updated with b1_fix
        end

        % constraint 2 number of chargers at each charging session
        A2 = zeros(pcl(end),n_v+nr_loc_charg);
        % Create inital matrix
        counter = 0;
        for ii = 2:nr_loc_charg+1 % for each charger location
            for jj = pcl(ii-1)+1:pcl(ii) % for each charging session
                pp = (1:n_v)'; % for all columns
                kk = ones(size(pp,1),1).*pp;
                idx_v = find(V(n_d+jj,kk) ~= 0);
                % find charger used for each column
                if isempty(idx_v) == 0
                    A2(jj,idx_v) = -V(n_d+jj,idx_v); % update chargers used
                    A2(jj,n_v+ii-1) = 1;
                    % Number of chargers at charger location
                    % b is updated by b2_fix
                end
            end

        end


        % lowerbound/upperbound
        lb1 = zeros(n_v,1);
        ub1 = zeros(n_v,1);
        lb1(1:n_v,1) = 0; % lower bound vehicle tasks
        ub1(1:n_v,1) = 1; % upper bound vehicle tasks
        for ii = 1:nr_loc_charg % nr of charger locations
            lb2(ii,1) = 0; % min number of chargers at a locaiton
            ub2(ii,1) = 1000000;% maxa number of chargers at a certain location
        end

        % Costs
        f1 = zeros(n_v,1);
        f1(1:n_v,1) = c_v; % costs for a vehicle tasks
        for xx = 1:nr_loc_charg
            f2(xx,1) = C_cl(xx); % costs for a charger
        end

        % Combine matrices
        A = [-A1;
             -A2;
            ];
        b = [-b1_fix;
             -b2_fix;
            ]; %%% NOTE orders matters, because of the dual variables
        lb = [lb1;lb2;];
        ub = [ub1;ub2;];
        f  = [f1;f2];
        Aeq = [];
        beq = [];

        %% Solve RMP with the CLP solver
        OPTLP = opti('f',f,'ineq',A,b,'eq',Aeq,...
            beq,'bounds',lb,ub,'options',opts);
        [uv,Obj_val,ef,lambda_RMP] = solve(OPTLP);

        % Infeasible solution check
        if ef < 1
            disp('Error: Infeasibile RMP')
        end
        % dual variables
        pi_tau      = lambda_RMP.Lambda.ineqlin(1:n_t); % trips
        pi_pcl      = -lambda_RMP.Lambda.ineqlin(end-pcl(end)+1:end);% depot
        pi_pcl(pcl(2)+1:pcl(end)) = pi_pcl(pcl(2)+1:pcl(end))+1e-4;
        % fast charging sessions
        % 1e-4 added to minimize charging sessions fast chargers

```

```
175
176         % primal upper bound or the current objective value of the RMP
177         disp(Obj_val)
178         RMP_Obj_val(iter(R_nr),R_nr) = Obj_val;
179
180
181
182         %% Solve subproblem
183
184         % turn off/on dominance rules
185         if i < iterations_method1 && method == 1    % Dominance rule 1
186             min_c_t_opt = 0;
187             min_s_t_opt = 0;
188             c_c2        = [1;];                     % only reduced costs
189         elseif method == 1
190             method = 2;
191         elseif i < iterations_method2 && method == 2 % Dominance rule 2
192             c_c2        = [1;2;];                    % reduced costs + SoC
193             min_c_t_opt = 0;
194             min_s_t_opt = 0;
195         elseif method == 2
196             method = 3;
197         else                                        % Dominance rule 3
198             min_c_t_opt = min_c_t;
199             min_s_t_opt = min_s_t;                  % all variables
200             c_c2        = [1;2;];
201             method = 3;
202         end
203         % The Label-correcting algorithm
204         tic
205
206         [V_new, obj_val_sub] = Label_correcting_algorithm(...
207             A_list_r,A_list_c,...
208             n_t,pi_tau,pi_pcl,e_t,ht_start,ht_end,...
209             A_list_c_sub,A_list_r_sub,SoC_min,SoC_max,e_charge,n_d...
210             ,pcl,loc_charge,loc_start,loc_end,e_h,diff_charge_line_2,min_c_t...
211             ,min_s_t,min_c_t_opt,min_s_t_opt,c_c2,I,depot,del_trips);
212
213         toc
214
215
216 %         % Solve the subproblem with a MILP solver
217 %         % function ConstraintsSubproblem has to on
218 %         tic
219 %         % costs
220 %         f_sub = zeros(n_d+n_t+pcl(end)+pcl(end),1);
221 %         for ii = 1:n_t
222 %             for yy = 1:size(x_loc{ii},1)  % for every arc to node
223 %                 f_sub(x_loc{ii}(yy)) = pi_tau(ii);
224 %             end
225 %         end
226 %         % SoC at each node
227 %         for ii =  n_d+n_t+pcl(end)+1:n_d+n_t+pcl(end)+pcl(end)
228 %             f_sub(ii) = -1e-5; % prevent random values
229 %         end
230 %         f_sub(n_d+n_t+pcl(end)+1:n_d+n_t+pcl(end)+pcl(end)) = pi_pcl;
231 %         xtype(1:n_d) = [1:n_d];
232 %         OPTLP = opti('f',f_sub,'ineq',A_sub,b_sub,'eq',Aeq_sub,...
233 %             beq_sub,'bounds',lb_sub,ub_sub,'xtype',xtype,'options',opts2);
234 %         [V_new, obj_val_sub,ef_sub] = solve(OPTLP);
235 %         if ef_sub < 1
236 %             disp('Error: MILP subproblem')
237 %         end
238 %         toc
239
240
241
242         % Determine column with the highest negative reduced costs
243         obj_val_sub_max = min(obj_val_sub);
244         % take the best column from all created columns
245
246         %% Add new columns
247         if i == 1
```

```
248         % for first iteration update the previous objective function to
249         % have a large enough decrease in the objective value
250             Obj_val_prev(1:nr_prev_val) = 2*Obj_val;
251         end
252 
253         % Determine if the best column has negative reduced costs and the
254         % RMP is improved enough for dominance rule 1 and dominance rule 2
255         if   -c_v-obj_val_sub_max>0.01*Obj_val && ...
256                 (Obj_val_prev(nr_prev_val-10)/Obj_val) > 1.01...
257                 && (method == 1 || method ==2)
258             disp('reduced costs =')
259             disp(-c_v-obj_val_sub_max)
260             for xx = 1:size(V_new,2) % for every new column
261                 if -c_v-obj_val_sub(xx)>0.01*Obj_val
262                     % check for each individual column if it has reduced
263                     % costs
264 
265                     V(1:n_d,end+1) = round(V_new(1:n_d,xx));
266                     % add arcs and fix storage errors
267                     V(n_d+1:end,end) = V_new(end-pcl(end)+1:end,xx);
268                     % Add rcpl percentage of charging sessions used
269                 end
270             end
271             % Update improvement in RMP objective value
272             Obj_val_prev(nr_prev_val) = Obj_val;
273             Obj_val_prev(1:nr_prev_val-1) = Obj_val_prev(2:nr_prev_val);
274 
275             % For method 3, more strict conditions
276         elseif -c_v-obj_val_sub_max > 0.001*Obj_val &&...
277                 (Obj_val_prev(1)/Obj_val) > 1.005...
278                 && method == 3
279             disp('reduced costs =')
280             disp(-c_v-obj_val_sub_max)
281             for xx = 1:size(V_new,2) % for every new column
282                 if -c_v-obj_val_sub(xx)>0
283                     % check for each individual column on negative
284                     % reduced costs
285                     V(1:n_d,end+1) = round(V_new(1:n_d,xx));
286                     % add arcs and fix storage errors
287                     V(n_d+1:end,end) = V_new(end-pcl(end)+1:end,xx);
288                     % Add rcpl percentage of charging sessions used
289 
290                 end
291             end
292             % Update RMP objective function improvement
293             Obj_val_prev(nr_prev_val) = Obj_val;
294             Obj_val_prev(1:nr_prev_val-1) = Obj_val_prev(2:nr_prev_val);
295 
296         else % if conditions do not hold
297 
298             rounding_sub = 1; % for rounding, no new column;
299             if method < 3 % if not all dominance rules are used
300                 % use the next one
301                 method = method + 1;
302                 rounding = -1; % no rounding
303                 break
304             else
305                 rounding = 1; % round variable
306                 break
307             end
308 
309 
310         end
311         disp('Iterations:')
312         disp(iter(R_nr))
313 
314     end
315 
316     %% MILP rounding
317     % Rounding with a MILP solver, not advised to use
318 %         if rounding == 1
319 %                         x_type(1:n_v+nr_loc_charg) = 1:n_v+nr_loc_charg;
320 %         %                 x_type(size(uv,1)-1) = size(uv,1)-1;
```

```
%                         OPTLP = opti('f',f,'ineq',A,b,'eq',Aeq,...
%                             beq,'bounds',lb,ub,'xtype', x_type,'options',opts2);
%                         [uv,Obj_val,ef] = solve(OPTLP);
%                         V_final = V(:,uv(1:n_v)==1);
%                          break
%         end

    %% Rounding with the diving heuristic
    if rounding == 1 % if rounding should happen
        [uv,obj_val_sub,ef,lambda_RMP,fxd_variables_cg] ...
            = Diving_heuristic(f,A,b,lb,ub,Aeq,beq, ...
            n_v,uv,opts);
        R_nr                    = R_nr + 1; % New Rounding Round
        iter(R_nr)              = 0; % zero iterations
        V_new                   = 0;
        disp('new round rounding')
        maxiter                 = maxiter;
        % Maximum number of extra columns rounded
        rounding_sub            = 0;
        % Column has been round and should be implemented in the CG
        method                  = 1;  % reset Dominance rules
        nr_prev_val             = 20;
        % after first rounding session number of steps to keep track of
        % improvent
    end
    Obj_val_prev(1:nr_prev_val) = 10000;
    % reset RMP objective value tracking
    if rounding == -1 % if methods are changed no rounding
        rounding = 1;
    end

end
```

Fixing the rounded variables

```
function [V,V_Store,b2_fix,fxd...
    ,x,A_list_r,A_list_c,n_d,n_t...
    ,left_npcl_final,left_d_final,...
    left_t_final,b1_fix,V_final,del_trips]...
    = Rounding_fix_arcs(...
    V,V_Store,n_t,n_d,fxd,A_list_c,A_list_r,b2_fix,x,...
    left_npcl_final,left_d_final,left_t_final,pcl,b1_fix,V_final,...
    A_list_c_sub,A_list_r_sub)
%% Update constraint auv ≥ 1
% fxd is the idx of the fixed variable
% Extract all arcs and transform them to which nodes are vissited
delete_var = A_list_c(find(V(1:n_d,fxd)>0));
% determine nodes used
delete_var2 = delete_var ≤ n_t; % for only trips
delete_var3 = unique(delete_var(delete_var2));
for xx = 1:delete_var3 % for all trips that are driven
    left_t_final(delete_var3) = 1; % update left_t auv ≥ 1-left_t
end
% Update the b for constraint for trips
b1_fix(delete_var3) = 1 - left_t_final(delete_var3);

%% Update charger constraints

% charger rounding
delete_var_d = find(V(n_d+1:n_d+pcl(2),fxd)>0);
% find depot slow charger arcs used
delete_var_c = find(V(n_d+pcl(2)+1:end,fxd)>0)...
    +pcl(2);
% find faast charger used

% for charger depots
if isempty(delete_var_d) == 0
    for xx = delete_var_d'
        left_d_final(xx) = left_d_final(xx) + V(n_d+xx,fxd); % Arc charged
        % update slow chargers used
```

```
    end
end
% charger locs
if isempty(delete_var_c) == 0
    for xx = delete_var_c'
        left_npcl_final(xx) = left_npcl_final(xx) + V(n_d+xx,fxd);% Arc charged
        % update fast chagers used at charging session xx
    end
end
b2_fix(1:pcl(2)) = left_d_final;
% change number of slow charger used in b value of constraints
b2_fix(pcl(2)+1:pcl(end)) =...
    left_npcl_final(pcl(2)+1:pcl(end));
% update fast chargers used in b value of the constraints

V_final = [V_final V(:,fxd)];
% Add new column to the final solution V_final

fxd = []; % delete fxd index


%% Option delete trips from the subproblem
% Update Trips deleted from subproblem
del_trips1 = find(left_t_final == 1); % find trips used
del_trips = unique([find(ismember(A_list_c_sub,del_trips1));...
    find(ismember(A_list_r_sub,del_trips1))]);
% find index arcs used, transport to label-correcting algorithm


%% Option two, delete all columns with trips in V_final
% % Delete columns and add new initial solution to make  RMP feasible
V(:,logical(sum(V(del_trips,:)==1))) = [];
% delete columns that have the same trips
V_Store(:,logical(sum(V_Store(del_trips,:)==1))) = [];
% delete columns that have the same trips
V_start = zeros(n_d,n_t);
for ii = 1:n_t
    for jj = [find(ii == A_list_c)]'

        if n_t ≤ A_list_r(jj) && n_t+pcl(2) > A_list_r(jj)
            V_start(jj,ii) = 1;
            loc_back = [find(ii == A_list_r)]';
            for zz = loc_back
                if A_list_c(zz) > n_t && n_t+pcl(2) ≥ A_list_c(zz)
                    V_start(zz,ii) = 1;
                    break;
                end
            end
            for yy = [find((A_list_c > n_t) & (n_t+pcl(2) ≥ A_list_c))]'
                if A_list_c(yy) ≠ [(A_list_r(jj)+1):A_list_c(zz)]
                    if A_list_r(yy) ≠ [1:n_t]
                        if A_list_c(yy) ≠ [1:n_t]
                            V_start(yy,ii) = 1;
                        end
                    end
                end
            end
        end
    end

end
V_start(end+1:end+pcl(end),:) = 0;
V = [V V_final V_start];




end
```

Column management

```
function [V,V_Store] = column_mangement(V,V_Store,c_v,pi_tau,pi_pcl,...
    n_lc,max_columns,n_d,x_loc,n_t,uv,pcl)
new_reduced_costs = 0;
re_store = [];% from V_Store to V
stored = []; % from V to V_store



%% Active column ppol
if size(V,2) > max_columns
% max number of columns threshold

% Calculation of the reduced costs for every column
ii =  1:size(V,2);
rc_c(ii) = 0;
rc_t(ii) = 0;
for jj = 1:n_t
    for yy = 1:size(x_loc{jj},1)
        rc_t(ii) = rc_t(ii) + V(x_loc{jj}(yy),ii)*pi_tau(jj); % trips
    end
end
for jj = 1:pcl(end)
    rc_c(ii) = rc_c(ii) + V(n_d+jj,ii)*pi_pcl(jj); % Chargers
end

new_reduced_costs(ii) = rc_c(ii)+rc_t(ii);
% combined reduced costs from both dual variables without cv
    [sorted_nrc,I] = sort(new_reduced_costs,'descend');
    % sort from low to high negative reduced costs
    nr_stored = size(V,2)-max_columns; % determine columns too many
    idx_not_stored2 = find(uv(I)); % Find columns used in last RMP solution
    idx_not_stored = unique([idx_not_stored2;]);
    % columns that are used in final solution

    I(idx_not_stored) = []; % delete columns that are in the RMP solution
    if nr_stored < numel(I)
        stored = I(1:nr_stored); % Store columns
    else
        stored = I; % Store all columns
    end


%% restore from V_store to V
% Determine reduced costs
ii =  1:size(V_Store,2);
rc_c2(ii) = 0;
rc_t2(ii) = 0;
for jj = 1:n_t
    for yy = 1:size(x_loc{jj},1)
        rc_t2(ii) = rc_t2(ii) + V_Store(x_loc{jj}(yy),ii)*pi_tau(jj);
    end
end
for jj = 1:n_lc
    rc_c2(ii) = rc_c2(ii) + V_Store(n_d+jj,ii)*pi_pcl(jj);
end

new_reduced_costs2(ii) = rc_c2(ii)+rc_t2(ii);

re_store = find(new_reduced_costs2 < -c_v); % send columns to second pool
if numel(re_store) > ceil(max_columns/100)
 % limit the number of columns added back to one percent
   re_store = re_store(1:ceil(max_columns/100));
end


end



%% delete columns
if isempty(stored) == 0
    V_Store = [V_Store V(:,stored)]; % replace columns
```

```
73      V(:,stored) = []; % delete columns
74  end
75  %% restore columns
76  if isempty(re_store) == 0 % cannot be empty
77      V = [V V_Store(:,re_store)]; % update V_store
78      V_Store(:,re_store) = []; % delete replaced columns
79  end
80
81
82  %% delete columns V_store when full
83  if size(V_Store,2) > ceil(n_t/2)
84      % threshold cannot become too high
85      ii =  1:size(V_Store,2);
86      rc_c2(ii) = 0;
87      rc_t2(ii) = 0;
88      for jj = 1:n_t
89          for yy = 1:size(x_loc{jj},1)
90              rc_t2(ii) = rc_t2(ii) + V_Store(x_loc{jj}(yy),ii)*pi_tau(jj);
91          end
92      end
93      for jj = 1:n_lc
94          rc_c2(ii) = rc_c2(ii) + V_Store(n_d+jj,ii)*pi_pcl(jj);
95      end
96
97      new_reduced_costs2(ii) = rc_c2(ii)+rc_t2(ii);
98      [¬,II] = sort(new_reduced_costs2,'descend');
99      % sort from low to high all reduced costs
100     V_Store(:,II(1:ceil(n_t/2))) = []; %Delete columns
101 end
102 end
```

Label-correcting algorithm

```
1   function [V_new, obj_val_sub] =...
2       Label_correcting_algorithm(A_list_r,A_list_c,...
3       n_t,pi_tau,pi_pcl,e_t,ht_start,ht_end,...
4       A_list_c_sub,A_list_r_sub,SoC_min,SoC_max,e_charge,n_d...
5       ,pcl,loc_charge,loc_start,loc_end,e_h,dt_2,min_c_t...
6       ,min_s_t,min_c_t_opt,min_s_t_opt,c_c2,I,depot,del_trips);
7   %% Initialization
8   count = 0;                      % Number of paths deleted
9   label= cell(1,n_t+pcl(end));  % Number of cells = Number of nodes
10  for xx = 1:n_t+pcl(end)       % For each node. Starting label should always be dominated
11      label{xx}(1,1) = -100000; % Reduced costs
12      label{xx}(2,1) = 0;       % SoC (%)
13      label{xx}(3,1) = 0;       % Shift time (s)
14      label{xx}(4,1) = 0;       % Charging time (s)
15      label{xx}(5,1) = 0;       % Start of the path
16  end
17
18  % Start node
19  label{n_t+1}(1,1) = 0;
20  label{n_t+1}(2,1) = 100;     % Starting SoC of the bus
21  label{n_t+1}(3,1) = 0;
22  label{n_t+1}(4,1) = 0;
23  label{n_t+1}(5,1) = 0;
24
25  c_c = [1;2]; % variable to reference to index of reduced costs and soc
26  start_node = A_list_r_sub(1); % The starting node, is always n_t+1
27  end_node = A_list_c_sub(end); % End node of the depot
28  SoC_max_slow = 100;          % Maximum SoC for slow chargers
29
30  if min_s_t == 0                 % cannot be used with min shift time
31      A_list_r_sub(del_trips) = []; % delete trips already in the final solution
32      A_list_c_sub(del_trips) = []; % delete trips already in the final solution
33      I(del_trips) = [];           % update the transformation array -> See Graph for I
34  end
35
36
37
```

```
38
39
40  % Domination rules used in this algorithm
41  % A numbering is used to determine when a path is dominated or dominates
42  % Each domination rule can give a number when is true or not, in the
43  % case of minimum shift time or minimum charge time a third option is added
44  % , namely irrelevant
45  % The combination of the three number determines if a path is dominated or
46  % not
47  % 1+1+5 = 7;  1+1 = 2 jj dominates ii
48  % 1+2+5 = 8 2+1+5 = 8 1+2+0 = 3 2+1+0= 3 2+1+10 = 13 1+2+10
49  % = 13 1+1+10 = 12; 2+2+5 = 9; % no domination at all
50  % 2+2+0 = 4  2+2 + 10 = 14  1 ii dominates jj
51
52
53  %% Start of the label-correcting algrotihm
54  for kk = 1:size(A_list_r_sub,1)   % For every arc ordered on starting time nodes with ...
        outgoing arcs
55      if rem(kk,750) == 0          % Each x iteration, delete empty rows in the matrices
56          for mm = 1:n_t+pcl(end)  % For each cell
57              label{mm} = label{mm}(logical([ones(5,1);¬all(label{mm}(6:end,:) == 0,2)]),:);
58              % delete empty rows
59          end
60      end
61      % The row and columns change in size in the matrices of cell label
62      % The columns need to change most likely in size, the change in rows is a
63      % choice, either the number of rows is around equal to the number of
64      % arcs or varies with the length of total arcs taken. From early
65      % simulation varying rows appeared to be faster,  but there is probably
66      % a better way to store the labels
67
68
69      ii = A_list_r_sub(kk); % New Node outgoing arc
70      jj = A_list_c_sub(kk); % New Node ingoing arc
71
72      if jj > n_t %% to charger
73          if ii > n_t %% from charger
74              % Determine Charger location
75              for xx = 2:size(pcl,2)
76                  if jj ≤ pcl(xx)+n_t
77                      jj_c = jj - n_t - xx+1;
78                      jj_r = jj - n_t - pcl(xx-1)-1;
79                      % jj - trips-sum of charging sessions - 1 because
80                      % c_t is in nodes instead of arcs
81                      break; % break when charger location found
82                  end
83              end
84              %% if current c time reaches min_c_t
85              %% path needs to be rechecked on domination at the same node
86              loc_dom = []; % var to determine location of dominated paths
87              if numel(c_c2)>1 % if dominance rule uses SoC
88                  if any(label{ii}(4,1:end) ≥ min_c_t) == 1
89                      % if any path has a higher current charge time than the min charge ...
                            time
90                      f_p = find(label{ii}(4,1:end) ≥ min_c_t); % find paths above min_c_t
91                      for hh = 1:numel(f_p) % for each path
92                          idx = 1:size(label{ii},2); % All other paths in the cell
93                          idx(f_p(hh)) = []; % Delete chosen path from this array
94                          if isempty(idx) == 0
95                              loc= (label{ii}(c_c2(1),idx) ≤ label{ii}(c_c2(1),f_p(hh)))+1;
96                              % if reduced costs is equal or higher
97                              loc(2,:) = (label{ii}(c_c2(2),idx) ≤ ...
                                    label{ii}(c_c2(2),f_p(hh)))+1;
98                              % if SoC is equal or higher
99                              equal = find(abs(label{ii}(c_c2(1),idx) - ...
                                    label{ii}(c_c2(1),f_p(hh))) < 1e-5);
100                             % if reduced costs is equal with error margin
101                             equal2 = find(abs(label{ii}(c_c2(2),idx) - ...
                                    label{ii}(c_c2(2),f_p(hh))) < 1e-2);
102                             % if SoC is equal with error margin
103                             if isempty(equal) == 0 % if reduced costs is equal
104                                 loc(1,equal) = loc(2,equal);
105                                 % change based on result of SoC comparison
```

```
106                             end
107                             if isempty(equal2) == 0 % if SoC is equal
108                                 loc(2,equal2) = loc(1,equal2);
109                                 % change based on result of reduced costs comparison
110                             end
111                             loc(3,:) = (label{ii}(4,idx) < min_c_t_opt)*10;
112                             % if path has lower current c time than the
113                             % minimum
114                             zero = label{ii}(4,idx)==0;
115                             % zero values are not dominated
116                             if isempty(zero) == 0
117                                 loc(3,zero) = 0;
118                             end
119                             Dom = sum(loc,1); % Sum loc to determine domination
120                             if Dom ≠ c_c2(end) & Dom ≠ c_c2(end)+5 % if not dominated
121                                 loc_dom = [loc_dom find(Dom == c_c2(end)*2| Dom == ...
                                        c_c2(end)*2+10| Dom == 1)];
122                                 % find paths that are dominated
123                             elseif Dom == c_c2(end) & Dom == c_c2(end)+5 % if dominated
124                                 loc_dom = [loc_dom f_p(hh)];
125                             end
126                         end
127                     end
128                 end
129             end
130             if isempty(loc_dom) == 0 % if paths are dominated
131                 label{ii}(:,loc_dom) = [];  % delete label
132                 count = count + size(loc_dom,2); % count dominated paths
133             end
134
135
136
137             %% Idle arc at depot
138             Dom =[];
139             loc_dom = [];
140             update2 = []; % when a updated path is added to the label set of the node
141             for hh = 1:size(label{ii},2) % for each path in node ii
142                 if xx == depot+1 % only for the depot
143                     if label{ii}(4,hh) ≥ min_c_t || label{ii}(4,hh) == 0 ...
                            ||label{ii}(2,hh) > SoC_max_slow-1
144                         % if charge time is zero or larger than the minimum
145                         % charge time
146                         % assumption minim charge time doese not count when max SoC is ...
                                reached on the depot
147                         % charge limits
148                         loc= (label{jj}(c_c2(1),1:end) ≤ label{ii}(c_c2(1),hh))+1;
149                         if numel(c_c2)>1
150                             loc(2,:) = (label{jj}(c_c2(2),1:end) ≤ ...
                                    label{ii}(c_c2(2),hh))+1;
151                             equal = find(abs(label{jj}(c_c2(1),1:end) - ...
                                    label{ii}(c_c2(1),hh)) < 1e-5);
152                             equal2 = find(abs(label{jj}(c_c2(2),1:end) - ...
                                    label{ii}(c_c2(2),hh)) < 1e-2);
153                             if isempty(equal) == 0
154                                 loc(1,equal) = loc(2,equal);
155                             end
156                             if isempty(equal2) == 0
157                                 loc(2,equal2) = loc(1,equal2);
158                             end
159                             loc(3,:) = (label{jj}(4,1:end) < min_c_t_opt)*10;
160                             zero = label{jj}(4,1:end)==0;
161                             if isempty(zero) == 0
162                                 loc(3,zero) = 0;
163                             end
164
165                         end
166
167
168                         % determine if dominated
169                         Dom = sum(loc,1);
170                         if Dom ≠ c_c2(end) & Dom ≠ c_c2(end)+5 % if not dominated
171                             update2 = [update2 hh]; % add updated path to the list ...
                                    that is added to node jj
```

```
                                loc_dom = [loc_dom find(Dom == c_c2(end)*2| Dom == ...
                                    c_c2(end)*2+10| Dom == 1)];
                                % find label that is dominated
                            end
                        end
                    end
                end
                % Update matrix of label jj
                if isempty(loc_dom) == 0
                    label{jj}(:,loc_dom) = [];  % delete dominated paths
                    count = count + size(loc_dom,2);
                end
                if isempty(update2)==0 % Add updated paths
                    label{jj}(c_c,end+1:end+numel(update2)) = label{ii}(c_c,update2);
                    % Add reduced costs and SoC to new column
                    label{jj}(5:max(size(label{ii}(5:end,update2)),1)+5,...
                        end+1-numel(update2):end) = ...
                        [label{ii}(5:end,update2); I(kk)*ones(1,numel(update2))];
                    % Add path + new taken arc to the new column
                    label{jj}(3,end+1-numel(update2):end) = 0; % update current shift time
                    label{jj}(4,end+1-numel(update2):end) = 0; % update current charging time
                end



                %% Charging sessions Arc
                loc_dom = [];
                update  = []; % Updated paths added to the node
                r_pcl   = []; % To remember percentage charged
                for hh = 1:size(label{ii},2)
                    % For each path in node ii compare to all path in node jj at once
                    loc = (label{jj}(c_c2(1),1:end) ≤ label{ii}(c_c2(1),hh)...
                        - pi_pcl(jj-n_t))+1; % Dominance reduced costs

                    % SoC domination rule
                    energy_u = label{ii}(2,hh)...
                        +dt_2{xx-1}(jj_r)*e_charge(xx-1);

                    if xx == 2 % For depot slow chargers
                        if energy_u ≥ SoC_max_slow
                            % if charge exceed max SoC slow chargers
                            energy_u1 = SoC_max_slow;
                        else
                            energy_u1 = energy_u;
                        end
                    else % For fast chargers
                        if energy_u ≥ SoC_max
                            % if updated exceed max SoC for fast chargers
                            if label{ii}(2,hh) > SoC_max
                                % if SoC was already larger than maximum SoC
                                energy_u1 = label{ii}(2,hh);
                            else
                                energy_u1 = SoC_max;
                            end
                        else
                            energy_u1 = energy_u;
                        end
                    end
                    % energy_u1 is used to determine dominance rule for SoC
                    % energy_u had to be adjusted to get realistic values

                    if numel(c_c2)>1 % if dominance rule 2 is active
                        loc(2,:) = (label{jj}(c_c2(2),1:end) < energy_u1) +1;
                        equal = find(abs(label{jj}(c_c2(1),1:end) - ...
                            label{ii}(c_c2(1),hh)+pi_pcl(jj-n_t)) < 1e-5);
                        equal2 = find(abs(label{jj}(c_c2(2),1:end) - energy_u1) < 1e-2);
                        if isempty(equal) == 0 % if reduced costs is equal
                            loc(1,equal) = loc(2,equal);
                        end
                        if isempty(equal2) == 0
                            loc(2,equal2) = loc(1,equal2);
```

```
243                     end
244
245                     if label{ii}(4,hh) + dt_2{xx-1}(jj_r) < min_c_t_opt
246                         % if constraint is active for new label
247                         loc(3,:) = (label{jj}(4,1:end) ≤ label{ii}(4,hh)+ ...
                                dt_2{xx-1}(jj_r))*2+5;
248                         % it is better to finish earlier with the min charge time ...
                                constraint
249                         if xx == depot +1% if depot charger
250                             zero = label{jj}(4,1:end)==0;
251                             % zero values dominate
252                             loc(3,zero) = 5;
253
254                         end
255                     else
256                         loc(3,:) = (label{jj}(4,1:end) < min_c_t_opt)*10;
257                         if xx == depot+1 % if depot charger
258                             zero = label{jj}(4,1:end)==0;
259                             % zero values are not dominated
260                             loc(3,zero) = 0;
261                         end
262                     end
263
264                 end
265
266
267
268                 % Determine which labels are dominated or not not dominated
269                 Dom = sum(loc,1);
270                 if Dom ≠ c_c2(end) & Dom ≠ c_c2(end)+5
271                     % determine if new_label is not dominated
272
273                 % Check if full charging arc can be used!
274                     SoC_charged = -1;
275                     % The SoC charging is stored with a minus sign
276                     % For the reason that no node number is negative
277                     % -1 means 100% of the charger arc is used.
278                     % if SoC max is exceeded, it is made equal to SoC max
279                     % By adjusting SoC_charged
280                     if xx == 2 % if slow charger at depot
281                         if energy_u > SoC_max_slow % if max charging limit is exceed
282                             SoC_charged = ...
283                                 -1 + (energy_u - ...
                                        SoC_max_slow)/(dt_2{xx-1}(jj_r)*e_charge(xx-1));
284                             % determine new SoC charged, based on how much
285                             % energy_u exceedes SoC_max_slow
286                             if SoC_charged > 0
287                                 SoC_charged = 0;
288                             end
289                         end
290                     elseif energy_u > SoC_max % For fast chargers
291                         if label{ii}(2,hh) > SoC_max
292                             SoC_charged = 0;
293                         else
294                             SoC_charged = ...
295                                 -1 + (energy_u - ...
                                        SoC_max)/(dt_2{xx-1}(jj_r)*e_charge(xx-1));
296                             % Similair rule, only now for SoC_max
297                             if SoC_charged > 0
298                                 SoC_charged = 0;
299                             end
300                         end
301
302                     end
303
304                     % Update which paths needs to added or deleted in the
305                     % node
306                     if SoC_charged < -1e-1 % error margin
307                         update = [update hh]; % Add path hh from node ii to node jj
308                         r_pcl = [r_pcl SoC_charged]; % Which percentage charging is used
309                         loc_dom = [loc_dom find(Dom == c_c2(end)*2| Dom == ...
                                c_c2(end)*2+10| Dom == 1)];
310                         % find label that is dominated in node jj by path
```

```
311                         % ii
312                     end
313
314                 end
315
316             end
317
318             % update outside of loop, so all paths are updated in two steps
319             % First all dominated paths in jj are deleted, then all new
320             % updated paths from node ii are added to node jj
321             % Charging
322             if isempty(loc_dom) == 0
323                 label{jj}(:,loc_dom) = [];  % delete label
324                 count = count + size(loc_dom,2);
325             end
326             if isempty(update) == 0
327                 label{jj}(c_c,end+1:end+numel(update)) = ...
328                     [label{ii}(c_c(1),update);label{ii}(2,update)]...
329                     - [pi_pcl(jj-n_t)*ones(1,numel(update));r_pcl*...
330                     dt_2{xx-1}(jj_r)*e_charge(xx-1);];
331                 % Update first two variables
332                 label{jj}(5:max(size(label{ii}(5:end,update)),1)+4 ...
333                 ,end-numel(update)+1:end)...
334                     = label{ii}(5:end,update);
335                 % add path to new label
336                 label{jj}(size(label{ii}(1:end,update),1)+1,end-numel(update)+1:end) = ...
                        I(kk); % add new node to old path
337                 label{jj}(size(label{ii}(1:end,update),1)+2,end-numel(update)+1:end) = ...
                        r_pcl(1:numel(update)); % energy charged
338                 label{jj}(size(label{ii}(1:end,update),1)+3,end-numel(update)+1:end) = ...
                        I(kk); % there has been charged
339                 % The above three lines add to which node the path has gone
340                 % and how much percentage of the charging arc is needed
341
342                 % Update last 2 parameters
343                 label{jj}(3,end-numel(update)+1:end) = 0;
344                 label{jj}(4,end-numel(update)+1:end) = label{ii}(4,update) - ...
                        r_pcl(1:numel(update)) *dt_2{xx-1}(jj_r);
345                 % Update current charging time
346             end
347             %%  From trip to charger node
348         else
349             for xx = 2:size(pcl,2)
350                 if jj ≤ pcl(xx)+n_t
351                     % Determine which charger location is travelled to
352                     % for deadhead trip
353                     break;
354                 end
355             end
356             update = [];
357             loc_dom = [];
358             for hh = 1:size(label{ii},2)
359                 if label{ii}(3,hh) ≥ min_s_t || label{ii}(3,hh) == 0
360                     % min shift constraint
361                     if label{ii}(2,hh) - e_h(loc_end(ii),loc_charge(xx-1)) ≥ SoC_min
362                         % Minimum SoC constraint
363                         loc = (label{jj}(c_c2(1),1:end) ≤ label{ii}(c_c2(1),hh))+1;
364                         if numel(c_c2) > 1 % if SoC variable rule is active
365                             loc(2,:) =  (label{jj}(c_c2(2),1:end) ≤ ...
                                    label{ii}(c_c2(2),hh)...
366                                     - e_h(loc_end(ii),loc_charge(xx-1)))+1;
367                             equal = find(abs(label{jj}(c_c2(1),1:end) - ...
                                    label{ii}(c_c2(1),hh)) < 1e-5);
368                             equal2 = find(abs(label{jj}(c_c2(2),1:end) - ...
                                    label{ii}(c_c2(2),hh)...
369                                     + e_h(loc_end(ii),loc_charge(xx-1))) < 1e-2);
370                             if isempty(equal) == 0 % if reduced costs is equal
371                                 loc(1,equal) = loc(2,equal);
372                             end
373                             if isempty(equal2) == 0
374                                 loc(2,equal2) = loc(1,equal2);
375                             end
376                             if min_c_t_opt > 0
```

```
377                                 loc(3,:) = (label{jj}(4,1:end) ≤ 0)+5;
378                                 equal3 = find(label{jj}(4,1:end) == 0);
379                                 if isempty(equal3) == 0
380                                     loc(3,equal3) = 0;
381                                 end
382                             else
383                                 loc(3,:) = 0;
384                             end
385                         end
386
387                         % determine if dominated
388                         Dom = sum(loc,1);
389                         if Dom ≠ c_c2(end) & Dom ≠ c_c2(end)+5   % determine if ...
                                new_label is not dominated
390                             update = [update hh]; % add new label to the list
391                             loc_dom = [loc_dom find(Dom == c_c2(end)*2| Dom == ...
                                    c_c2(end)*2+10| Dom == 1)]; % find label that is dominated
392                         end
393                     end
394                 end
395             end
396
397             if isempty(loc_dom) == 0
398                 label{jj}(:,loc_dom) = [];  % delete label
399             end
400             if isempty(update) == 0
401                 label{jj}(c_c,end+1:end+numel(update)) = label{ii}(c_c,update)...
402                     - [0;e_h(loc_end(ii),loc_charge(xx-1))]; % Update first two variables
403                 label{jj}(5:size(label{ii}(5:end,update),1)+5,end+1-numel(update):end) ...
                        = ...
404                     [label{ii}(5:end,update);I(kk)*ones(1,numel(update))];% Update path
405                 label{jj}(3,end+1-numel(update):end) = 0;
406                 label{jj}(4,end+1-numel(update):end) = 0;
407             end
408         end
409         %% to trip
410     else
411         %% to trip from charger
412         if ii > n_t
413
414             for xx = 2:size(pcl,2)
415                 if ii ≤ pcl(xx)+n_t
416                     % Determine charger location for deadhead trip
417                     break;
418                 end
419             end
420             update = [];
421             loc_dom = [];
422             for hh = 1:size(label{ii},2)
423                 if label{ii}(4,hh) ≥ min_c_t ||  label{ii}(4,hh) == 0
424                     % Minimum charge time should be met or be zero
425                     if label{ii}(2,hh) - e_t(jj)- e_h(loc_charge(xx-1),loc_start(jj)) ≥...
                            SoC_min
426                         % Minimum SoC constraint
427                         loc = (label{jj}(c_c2(1),1:end) ≤ label{ii}(c_c2(1),hh)...
428                             - pi_tau(jj))+1;
429                         if numel(c_c2) > 1
430                             loc(2,:) = (label{jj}(c_c2(2),1:end) ≤ ...
                                    label{ii}(c_c2(2),hh)...
431                                 -e_t(jj) - e_h(loc_charge(xx-1),loc_start(jj)))+1;
432                             equal = find(abs(label{jj}(c_c2(1),1:end) - ...
                                    label{ii}(c_c2(1),hh)+pi_tau(jj)) < 1e-5);
433                             equal2 = find(abs(label{jj}(c_c2(2),1:end) - ...
434                                 label{ii}(c_c2(2),hh)+e_t(jj) + ...
                                        e_h(loc_charge(xx-1),loc_start(jj))) < 1e-2);
435                             if isempty(equal) == 0 % if reduced costs is equal
436                                 loc(1,equal) = loc(2,equal);
437                             end
438                             if isempty(equal2) == 0
439                                 loc(2,equal2) = loc(1,equal2);
440                             end
441                             if min_s_t_opt > 0
442                                 loc(3,:) = (label{jj}(3,1:end) ≤ 0)+5;
```

```
443                                 equal3 = find(label{jj}(3,1:end) == 0);
444                                 if isempty(equal3) == 0 %
445                                     loc(3,equal3) = 0;
446                                 end
447                             else
448                                 loc(3,:) = 0;
449                             end
450                         end
451                         % min_s_t is zero at the moment
452                         Dom = sum(loc,1);
453                         % determine if dominated
454                         if Dom ≠ c_c2(end) & Dom ≠ c_c2(end)+5 % determine if ...
                                new_label is not dominated
455                             update = [update hh];
456                             loc_dom = [loc_dom find(Dom == c_c2(end)*2| Dom == ...
                                    c_c2(end)*2+10| Dom == 1)]; % find label that is dominated
457                         end
458                     end
459                 end
460             end
461             if isempty(loc_dom) == 0
462                 label{jj}(:,loc_dom) = [];  % delete label
463                 count = count + size(loc_dom,2);
464             end
465             if isempty(update) == 0
466                 label{jj}(c_c,end+1:end+numel(update)) = label{ii}(c_c,update)...
467                     - [pi_tau(jj);e_t(jj)+e_h(loc_charge(xx-1),loc_start(jj))];
468                 label{jj}(5:size(label{ii}(5:end,update),1)+5,end+1-numel(update):end) ...
                        =...
469                     [label{ii}(5:end,update); I(kk)*ones(1,numel(update))];
470                 label{jj}(3,end+1-numel(update):end) = ht_end(jj)-ht_start(jj);
471                 % update current shift time
472                 label{jj}(4,end+1-numel(update):end) = 0;
473
474             end
475
476             %% from trip
477         else
478             e  = - e_t(jj) - e_h(loc_end(ii),loc_start(jj)) - SoC_min;
479             % to reduce computational time precalculation energy
480             loc_dom = [];
481             update = [];
482             for hh = 1:size(label{ii},2) % for each path in ii to path in node jj
483                 if label{ii}(2,hh)  ≥ -e  % SoC constraint
484                     loc = (label{jj}(c_c2(1),1:end) ≤ label{ii}(c_c2(1),hh)...
485                         - [pi_tau(jj)])+1;
486                     if numel(c_c2) > 1 % if SoC constraint is active
487                         loc(2,:) = (label{jj}(c_c2(2),1:end) ≤ label{ii}(c_c2(2),hh) - ...
                                e_t(jj)...
488                             - e_h(loc_end(ii),loc_start(jj)))+1  ;
489                         equal = find(abs(label{jj}(c_c2(1),1:end) - ...
                                label{ii}(c_c2(1),hh)+ [pi_tau(jj)]) < 1e-5);
490                         equal2 = find(abs(label{jj}(c_c2(2),1:end)...
491                             - label{ii}(c_c2(2),hh)+ e_t(jj) ...
                                    +e_h(loc_end(ii),loc_start(jj))) < 1e-2);
492                         if isempty(equal) == 0 % if reduced costs is equal
493                             loc(1,equal) = loc(2,equal);
494                         end
495                         if isempty(equal2) == 0
496                             loc(2,equal2) = loc(1,equal2);
497                         end
498                         if label{ii}(3,hh) + ht_end(jj)-ht_start(jj) < min_s_t_opt
499                             % if constraint is active for new label
500                             loc(3,:) = (label{jj}(3,1:end) ≤ label{ii}(4,hh))*2+5;
501                             % it is better to finish earlier with the min shift time ...
                                    constraint
502                         else
503                             loc(3,:) = (label{jj}(3,1:end) < min_s_t_opt)*10;
504                         end
505                     end
506                     Dom = sum(loc,1);
507
508                     % determine if dominated
```

```
509                     if Dom ≠ c_c2(end) & Dom ≠ c_c2(end)+5 % determine if new_label is ...
                            not dominated
510                         update = [update hh];
511                         loc_dom = [loc_dom find(Dom == c_c2(end)*2| Dom == ...
                                c_c2(end)*2+10|Dom == 1)];
512
513                     end
514                 end
515             end
516             % update for trip trip outside loop
517             if isempty(loc_dom) == 0
518                 label{jj}(:,loc_dom) = [];  % delete label
519                 count = count + size(loc_dom,2);
520             end
521             if isempty(update) == 0
522                 label{jj}(c_c,end+1:end+numel(update)) = label{ii}(c_c,update)...
523                     - [pi_tau(jj);e_t(jj)+e_h(loc_end(ii),loc_start(jj));];
524                 label{jj}(5:max(size(label{ii}(5:end,update),1))+5 ...
525                     ,end+1-numel(update):end)...
526                     = [label{ii}(5:end,update); I(kk)*ones(1,numel(update))];
527                 label{jj}(3,end+1-numel(update):end) = label{ii}(3,update) + ...
                        ht_end(jj)-ht_start(jj);
528                 % Update shift time
529                 label{jj}(4,end+1-numel(update):end) = 0;
530             end
531         end
532
533     end
534 end
535
536
537 %% Process results
538 dele = [];
539 yy = 0;
540 xxx = 0;
541 while size(label{end_node}(1,:),2)>0
542     % while there are still paths in the end node
543     yy = yy + 1;
544     new_path = find(label{end_node}(1,:) == max(label{end_node}(1,:)),1);
545     % Find path with highest reduced costs
546     new_arcs = nonzeros(label{end_node}(5:end,new_path));
547     % Find all arcs in the the new path (negative r_pcl is still included)
548
549     % Option to Delete charging sessions after last trip in the path
550     %         last_arc = find(label{end_node}(5:end,new_path)≤n_t...
551     %             &label{end_node}(5:end,new_path)>0,1,'last')+4; % find last driven trip
552     %         if isempty(last_arc) == 0
553     %             loc_c_end = find(label{end_node}(last_arc(end)+1:end,new_path) < ...
        0)+last_arc(end);
554     %             % find charging sesssions (C) after last trip
555     %             label{end_node}(loc_c_end,new_path) = 0;
556     %             % equal charging to zero
557     %         end
558
559     % Initial matrix
560     tt = zeros(n_d+pcl(end),1); % nr of variables, nr of arcs in chosen path
561     % Matrix to rewrite the path to the form of vehicle tasks v
562
563     % Storage of arcs in v
564     tt(ceil(new_arcs(new_arcs>0))) = 1; % Add all arcs in the path to v
565     % Ceil used for rounding errors
566     % Storage of rpcl in v
567     for ii = 1:size(new_arcs,1)-1 % for all arcs
568         if ii + 2 < size(new_arcs,1) % to not exceed index
569             % check for charger double numbers within the charge percentage used
570             if new_arcs(ii) == new_arcs(ii+2)
571                 % if two arcs are the same with a distance of 2
572                 tt(A_list_c(new_arcs(ii))-n_t+n_d) = -new_arcs(ii+1);
573                 % percentage of charging session that is that is used
574             end
575         end
576     end
577
```

```
578      V_new(:,yy) = sum(tt,2); % add tt to V_new
579      obj_val_sub(yy) = -label{end_node}(1,new_path);
580      % Determine objective function without cv
581
582      % Find similair columns, delete those
583      for nn = 1:size(label{end_node}(1,:),2) % for every path in the end node
584          diff{nn} = setdiff(label{end_node}(5:end,nn),label{end_node}(5:end,new_path));
585          delete_1 =  numel(diff{nn}(diff{nn}>0))/(numel(new_arcs(new_arcs>0))) ≤ 0.05;
586          % if path does not differ 5% from the previously added path
587          if delete_1 == 1
588              dele = [dele nn]; % Add path to delete list
589              xxx = xxx + 1;
590          end
591      end
592
593      label{end_node}(:,dele) = []; % delete similair paths
594      dele = [];
595  %   break % if only one column is chosen, the one with the highest negative
596  %   costs
597  end
598
599
600  end
```

Diving heuristic

```
1   function [uv,obj_val_sub,ef,lambda_RMP,fxd_variables] ...
2       = Diving_heuristic(f,A,b,lb,ub,Aeq,beq, ...
3       n_v,uv,opts)
4   % fxd_variables == fxd_variables_cg in this function
5
6   % Initial variables
7   ii = 0; % initally needed
8   fxd_variables_prev = []; % Old variable, now empty variable
9   % Could probably be removed, but this function was always very error prone
10  % Therefore the variable is not deleted, since it does not influence the
11  % performance
12  fxd_variables      = []; % new fixed variables
13  error_present      = 1;  % to make while loop, rounding will end when new RMP is feasible
14  count_errors       = 0;  % number of errors allowed per iteration
15  Dlt_array          = [];
16  % Variable that determines which columns are infeasbile to round up
17
18
19
20  while error_present == 1 % loop to secure feasbility for roundng
21
22
23
24  fxd_variables2 = find(abs(uv(1:n_v)-1)≤1e-3);
25  % Determine variables that are equal to one in the current RMP solution
26  for rr  = 1:size(fxd_variables2,1) % for every new variable
27      if isempty(fxd_variables_prev) == 1 % if no prev variables are fixed
28          rr = 1; % Determine which one is fixed
29          break % Only one new variable fixed, then stop
30      end
31      if ismember(fxd_variables2(rr),fxd_variables_prev) == 0
32          break
33      end
34  end
35  if isempty(fxd_variables2) == 0 % if there are no new integer variables fixed
36      if ismember(fxd_variables2(rr),fxd_variables_prev) == 1 && rr == ...
            size(fxd_variables2,1)
37          fxd_variables = fxd_variables_prev;
38      else
39          fxd_variables = [fxd_variables_prev; fxd_variables2(rr)];
40          % New fixed variable
41      end
42  end
43
```

```
44
45 %% Round new variable with highest fractional value
46 % to integer if no new integer is already found
47 if  numel(fxd_variables2) ≤ numel(fxd_variables_prev)
48         % for first iteration needed, either there are previous fixed
49         % variables or there are no new integer variables
50         if numel(fxd_variables) == numel(fxd_variables_prev)
51             % to prevent fixing to many variables at once, last sentence is for
52
53             lctn_b_rnd_ntgr = find(uv(1:n_v)<0.9999&uv(1:n_v)>0); % find continious ...
                   variables
54             % find all variable lower than 1, 0.9999 because of error % margin
55             vl_b_rnd_ntgr = max(uv(lctn_b_rnd_ntgr));
56             % find variable with the highest fractional value
57             lctn_b_rnd_ntgr2 = find(uv(1:n_v)== vl_b_rnd_ntgr,1,'last');
58             % find the location of one of the variables that has the maximum value.
59             % find indexes max variable
60             fxd_variables = [lctn_b_rnd_ntgr2; fxd_variables];
61             % new fixed variable
62
63         end
64
65 end
66
67 %% Feasibility test
68
69 % Create constraints with sub2ind method
70     if isempty(fxd_variables) == 0
71         % Indexing
72         fxd_c2 = find(fxd_variables > n_v);   % find fixed variables for chargers
73         fxd_c = fxd_variables(fxd_c2);        % find right index
74         fxd_t2 = find(fxd_variables ≤ n_v);  % find idx fixed for trips
75         fxd_t = fxd_variables(fxd_t2);        % find fxdvariables
76         % Creation of matrices
77         Aeq_fix = sparse(numel(fxd_variables)+numel(Dlt_array),size(A,2));
78         beq_fix = zeros(numel(fxd_variables)+numel(Dlt_array),1);
79         % Aeq needs to be predefined to use sub2ind
80         if isempty(fxd_c) == 0 % if there are no chargers yet fixed
81             ii = [];
82             ii(1:size(fxd_c,1),1) = 1:size(fxd_c,1);
83             index_c = sub2ind(size(Aeq_fix), ii, fxd_c(ii));
84             Aeq_fix(index_c) = 1/ceil(uv(fxd_c(ii)));
85             beq_fix(ii,1) = 1;
86         else
87             ii = [];
88         end
89         jj = [];
90         jj(1:size(fxd_t,1),1) = 1:size(fxd_t,1); % there are always trips fixed
91         index_t = sub2ind(size(Aeq_fix), numel(ii)+jj, fxd_t(jj));
92         % index to linear index Aeq needs to be predefined
93         Aeq_fix(index_t) = 1;    % fixed trip variables
94         beq_fix(numel(ii)+jj,1) = 1; % equal to one
95
96     end
97     if isempty(Dlt_array) == 0 % if infeasible variables are present
98         xx = [];
99         xx(1:size(Dlt_array,1),1) = 1:size(Dlt_array,1);
100        index_d = sub2ind(size(Aeq_fix), numel(ii)+numel(jj)+xx, Dlt_array(xx));
101        Aeq_fix(index_d) = 1; % fixed variables to zero
102        beq_fix(numel(ii)+numel(jj)+xx,1) = 0;
103    end
104
105    %combine Aeq and Aeqfix
106    Aeq2 = [Aeq;
107        Aeq_fix];
108    beq2 = [beq;
109        beq_fix];
110
111    % LP solver
112    % solve to find errors
113    OPT_sub1 = opti('f',f,'ineq',A,...
114        b,'eq',Aeq2,beq2,'bounds',lb,ub,...
115        'options', opts);
```

```
116      [uv,obj_val_sub,ef,lambda_RMP] = solve(OPT_sub1); % solve sub
117
118
119  %% Find infeasible variable and fix to zero
120      if ef < 1 &&  size(fxd_variables,1) ≠ size(fxd_variables_prev,1)
121          % If there is an error
122          change = setdiff(fxd_variables,fxd_variables_prev)';
123          % find newly fixed fxd_variable
124          count_errors = count_errors + 1; % count errors
125          % Round with MILP if so many errors occur
126  %         if count_errors > 1500 % number of errors allowed
127  %                 %use MILP rounding
128  %                 x_type(1:n_v) = 1:n_v;
129  %                 x_type(size(uv,1)-1) = size(uv,1)-1;
130  %                 OPTLP = opti('f',f,'ineq',A,b,'eq',Aeq,...
131  %                     beq,'bounds',lb,ub,'xtype', x_type,'options',opts2);
132  %                 [uv,Obj_val,ef] = solve(OPTLP);
133  %                 disp('diving heuristic does not work, MILP used')
134
135          % Limit on allowable errors
136          if count_errors == 5
137              fxd_variables = [];
138              disp('ERROR: RMP Rounding infeasible')
139              break % Stop rounding due to too many erros
140          else
141              xx = [];
142              for ii = size(change,1)
143                  xx(ii) = find(change(ii) == fxd_variables);
144                  % find indexes of fixed variables
145              end
146              del = find(change > n_v);% find all chargers
147              change(del) = []; % charging variables are not fixed.
148              Dlt_array = [Dlt_array; change];
149              % add prev fxd variables to delete list, so equal to zero
150              if isempty(xx) == 0 % if xx is empty, which should not happen
151                  fxd_variables(xx) = [];
152                  % delete new fixed variable from the list with fixed variables
153              end
154
155  %% recalculate uv with new fixed variable set to zero with LP Solver
156              if isempty(fxd_variables) == 0
157                  fxd_c2 = find(fxd_variables > n_v);   % find idx fixed for chargers
158                  fxd_c = fxd_variables(fxd_c2);        % find fxdvariables
159                  fxd_t2 = find(fxd_variables ≤ n_v);  % find idx fixed for trips
160                  fxd_t = fxd_variables(fxd_t2);        % find fxdvariables
161                  % Initial empty matrices
162                  Aeq_fix = sparse(numel(fxd_variables)+numel(Dlt_array),size(A,2));
163                  beq_fix = zeros(numel(fxd_variables)+numel(Dlt_array),1);
164                  % Aeq needs to be predefined to use sub2ind
165                  if isempty(fxd_c) == 0 % if there are no chargers yet fixed
166                      ii = [];
167                      ii(1:size(fxd_c,1),1) = 1:size(fxd_c,1);
168                      index_c = sub2ind(size(Aeq_fix), ii, fxd_c(ii));
169                      Aeq_fix(index_c) = 1/ceil(uv(fxd_c(ii)));
170                      beq_fix(ii,1) = 1;
171                  else
172                      ii = [];
173                  end
174                  jj = [];
175                  jj(1:size(fxd_t,1),1) = 1:size(fxd_t,1);
176                  % Fix trips
177                  index_t = sub2ind(size(Aeq_fix), numel(ii)+jj, fxd_t(jj));
178                  % index to linear index Aeq needs to be predefined
179                  Aeq_fix(index_t) = 1;    % fix trips to one
180                  beq_fix(numel(ii)+jj,1) = 1;
181
182              end
183              if isempty(Dlt_array) == 0
184              % created constraints for infeasible variables which need to be set to zero
185                  if isempty(fxd_variables) == 1
186                      ii = [];
187                      jj = [];
188                  end
```

```
                xx = [];
                xx(1:size(Dlt_array,1),1) = 1:size(Dlt_array,1);
                index_d = sub2ind(size(Aeq_fix), numel(ii)+numel(jj)+xx, Dlt_array(xx));
                Aeq_fix(index_d) = 1; % fixed variables to zero
                beq_fix(numel(ii)+numel(jj)+xx,1) = 0;
            end
            % Resolve LP solver
                Aeq2 = [Aeq;
                        Aeq_fix];
                beq2 = [beq;
                        beq_fix];
                OPT_sub1 = opti('f',f,'ineq',A,...
                    b,'eq',Aeq2,beq2,'bounds',lb,ub,...
                    'options', opts);
                [uv,obj_val_sub,ef,lambda_RMP] = solve(OPT_sub1);
        end
%% Feasible RMP

else
        error_present = 0; % to break while loop, if there is no error;
    end

end

end
```

Results

This function creates all plots based on the results of the column generation model. It contains a Gantt chart, which is created in a similar way as in the previous projects, a progress of the SoC during the day plot, a plot which shows the number of chargers used during the day and the progress of the objective function of the RMP plot. It also contains a function Lower_bound.m that creates the lower bounds for the problem.

```
function [nr_bus,Costs,percentage_double,trips_double,...
    total_energy,double_deadhead,integrality_gap,bus_min]...
    = Results(V_final,x_loc,h_e,h_s,charge_line_2,...
    dt_2,A_list_c,A_list_r,ht_end,ht_start,n_t,...
    pcl,h,loc_charge,loc_start,loc_end,nr_loc_charg, iter,...
    e_h,SoC_max,SoC_min,c_v,C_cl,RMP_Obj_val,b2_fix,...
    n_d,charge_line_all,e_t,e_charge,e_max,graphplot,time_nodes)


nr_bus = size(V_final,2);
% Determine number of buses used from V_final
buses_loc = 1:size(V_final,2);
% Create array from the total of buses used
%% Determine costs
Costs = nr_bus*c_v;
for xx = 2:nr_loc_charg+1 % for each charging location
Costs = Costs + ceil(max(b2_fix(pcl(xx-1)+2:pcl(xx))))* C_cl(xx-1);
% Round up no of chargers used, the highest value is the number of chargers used
end
%% Plots
% Plot path on graphs
% Advice to not use, due to plotting takes very long
% figure;
% for jj = 1:size(V_final,2)
% [c,r] = find(V_final(1:n_d,jj));
%
% [s_c,I] = sort(time_nodes(A_list_c(c)));
% c = A_list_c(c(I));
% c(2:end+1) = c;
% c(1) = n_t+1;
%
% highlight(graphplot,c,'edgecolor',[rand(1) rand(1) rand(1)],'LineWidth',3)
% hold on;
% end
```

```
35
36 % Plot the Gantt chart for the schedule
37 trips_double(1:n_t) = 0; % Determine the number of trips driven double
38
39 figure
40 hold on
41 grid on
42
43 set(gcf,'position',[10,10,550,550]); % Increase size
44 set(gca,'LooseInset',get(gca,'TightInset'),'fontweight','bold', 'fontsize',18)
45 % Eliminate white space around figure
46
47 axis([floor(min(h_s{1})/60) ceil(max(h_e{1})/60) 0 nr_bus+1])
48 xlabel('Time [h]','fontweight','bold')
49 xticks(floor(min(h_s{1})/60):1:ceil((max(h_e{1})/60)))
50
51 ylabel('Vehicle number [-]','fontweight','bold')
52 yticks(0:1:nr_bus)
53
54
55 for ii = 1:nr_bus
56     for kk = 1:n_t   % For each trip
57         for jj = [x_loc{kk}]' % For all arcs that go to trip kk
58             if V_final(jj,buses_loc(ii)) == 1 % if bus drives trip
59                 start_tr = ht_start(kk)/60;
60                 end_tr = ht_end(kk)/60;
61                 trips_double(kk)  = trips_double(kk) + 1;
62                 % Update double driven trips
63                 trip_plot = plot([start_tr end_tr],[ii ii],...
64                     '-','Color',[0 0.6196 0.0431],'DisplayName','Trip','LineWidth',16);
65 %                 text((start_tr),ii,...
66 %                 num2str(kk),'FontSize',16); % user friendly number of all
67 %                 trips
68                 break
69             end
70         end
71     end
72 end
73
74 percentage_double = sum(trips_double)/n_t;
75 % Calculate the percentage of trips that are driven double
76
77 %Plotting depot slow charging sessions
78 for ii = 1:nr_bus
79     for xx = 2 % depot
80         for kk = 2:pcl(xx)-pcl(xx-1) % for each charging sessions
81             pp = kk+pcl(xx-1)+n_d; % the rpcl index number
82             if V_final(pp,buses_loc(ii)) > 0
83                 start_d = charge_line_2{xx-1}(kk-1)/60;
84                 end_d = (charge_line_2{xx-1}(kk)-(dt_2{xx-1}(kk-1)* ...
85                     (1-V_final(pp,buses_loc(ii)))))/60;
86                 depot_plot = plot([start_d end_d],[ii ii],...
87                     '-','Color',[0.2549 0.7255 0.9098],'DisplayName','Depot Charging ...
                        session','LineWidth',9);
88             end
89         end
90     end
91
92  % Plotting fast charging sessions
93     sequence = 0; % for writing text (location charger used) on the correct place
94     for xx = 3:nr_loc_charg+1 % for fast charger locations
95         for kk = 1:pcl(xx)-pcl(xx-1)  % for all charging sessions
96             pp = kk+pcl(xx-1)+n_d; % the rpcl index number
97             if V_final(pp,buses_loc(ii)) > 0 % if is in the final solution
98                 start_c = charge_line_2{xx-1}(kk-1)/60;
99                 end_c = (charge_line_2{xx-1}(kk)-(dt_2{xx-1}(kk-1)* ...
100                    (1-V_final(pp,buses_loc(ii)))))/60;
101                charge_plot = plot([start_c end_c],[ii ii],...
102                    '-','Color',[0.64 0 0],'DisplayName','Fast charging ...
                        session','LineWidth',9);
103                sequence = sequence+1;
104            elseif sequence > 0
105 %             text((beginning(1)+ending(end))/2-1/60,ii,...
```

```
%             num2str(xx-2)); % text for location fast charging sessions
            sequence = 0;
            end
            if sequence > 0 % for text writing and placing in the middle of a
                % sequence of charging sessions
%                 beginning(sequence) = start_c;
%                 ending(sequence) = end_c;
            end
        end
    end
end

% Plot deadhead trips
double_deadhead(1:n_d) = 0; % Determine double deadhead trips
for ii = 1:nr_bus % for each bus
    for kk = 1:n_d % for each arcs
        if  V_final(kk,buses_loc(ii)) == 1 % if bus drives arc
            loc1 = A_list_c(kk); % end location
            loc2 = A_list_r(kk); % start location
            if loc1 > n_t
                if loc2 > n_t
                    % from charger to charger
                    for xx = 2:nr_loc_charg+1 % find correct loc charger
                        if loc1 ≤ pcl(xx)+n_t
                            loc3 = xx-1;
                            loc5 = loc1 - n_t - pcl(xx-1);
                            break
                        end
                    end
                    for xx = 2:nr_loc_charg+1 % find correct loc charger
                        if loc2 ≤ pcl(xx)+n_t
                            loc4 = xx-1;
                            loc6 = loc2 - n_t - pcl(xx-1);
                            break
                        end
                    end
                    start_t = charge_line_2{loc4}(loc6)/60;
                    end_t = start_t + (h(loc_charge(loc4),loc_charge(loc3)))/60;
                else % trip to charger
                    for xx = 2:nr_loc_charg+1 % find correct charger location
                        if loc1 ≤ pcl(xx)+n_t
                            loc3 = xx-1;
                            loc5 = loc1 - n_t - pcl(xx-1);
                            break
                        end
                    end
                    start_t = ht_end(loc2)/60;
                    end_t = start_t + h(loc_end(loc2),loc_charge(loc3))/60;
                end
            else % for charger to trip
                if loc2 > n_t
                    for xx = 2:nr_loc_charg+1 % find correct location
                        if loc2 ≤ pcl(xx)+n_t
                            loc4 = xx-1;
                            loc6 = loc2 - n_t - pcl(xx-1);
                            break
                        end
                    end
                    start_t = charge_line_2{loc4}(loc6)/60;
                    end_t = start_t + h(loc_charge(loc4),loc_start(loc1))/60;
                else % for trip to trip arc
                    start_t = ht_end(loc2)/60;
                    end_t = start_t + h(loc_end(loc2),loc_start(loc1))/60;
                end
            end
            hold on;
            double_deadhead(kk) = double_deadhead(kk)+1;
            % calculate double deadhead trips
            deadhead_plot = plot([start_t end_t],[ii ii],...
            '-','Color',[0.9020 0.8196 ...
                0.5059],'DisplayName','Deadheadtrip','LineWidth',9);
        end
    end
```

```
178 end
179 % legend([trip_plot depot_plot charge_plot deadhead_plot]...
180 %     ,{'Trip','Depot charging session','Fast charging session','Deadhead trip'},...
181 %     'Location', 'northeastoutside','fontweight','bold','fontsize',18)
182
183
184
185 %% The progress of the SoC during the day plot
186 x_all = find(A_list_c ≤ n_t);
187 figure
188 for pp = 1:nr_bus % for all buses used
189     t_e = 0; % trip energy
190     eta = 0; % charge energy
191     d_e = 0; % deadhead trip energy
192
193     % variables for deadhead trips
194     c1 = 1;
195     c2 = 1;
196     c3 = 1;
197     c4 = 1;
198     out_dead1 = [];
199     in_dead1 = [];
200     out_dead2 = [];
201     in_dead2 = [];
202     out_dead3 = [];
203     in_dead3 = [];
204     out_dead4 = [];
205     in_dead4 = [];
206
207     % Find deadhead trips
208     loc = find(V_final(1:n_d,buses_loc(pp)));
209     for ii = 1:numel(loc)
210         if A_list_c(loc(ii)) > n_t
211             if A_list_r(loc(ii)) > n_t % out charger in charger
212                 out_dead1(c1,1) = A_list_r(loc(ii))-n_t;
213                 for xx = 2:nr_loc_charg+1 % find correct loc
214                     if out_dead1(c1,1) ≤ pcl(xx)
215                         out_dead1(c1,2) = xx-1;
216                         break
217                     end
218                 end
219                 in_dead1(c1,1) = A_list_c(loc(ii))-n_t;
220                 for xx = 2:nr_loc_charg+1 % find correct loc
221                     if in_dead1(c1,1) ≤ pcl(xx)
222                         in_dead1(c1,2) = xx-1;
223                         break
224                     end
225                 end
226                 c1 = c1 + 1;
227             else %out trip in charger
228                 out_dead2(c2) = A_list_r(loc(ii));
229                 in_dead2(c2,1) = A_list_c(loc(ii))-n_t;
230                 for xx = 2:nr_loc_charg+1 % find correct loc
231                     if in_dead2(c2,1) ≤ pcl(xx)
232                         in_dead2(c2,2) = xx-1;
233                         break
234                     end
235                 end
236                 c2 = c2 + 1;
237             end
238         else
239             if A_list_r(loc(ii)) > n_t %out charger in trip
240                 out_dead3(c3,1)= A_list_r(loc(ii))-n_t;
241                 for xx = 2:nr_loc_charg+1 % find correct loc
242                     if out_dead3(c3,1) ≤ pcl(xx)
243                         out_dead3(c3,2) = xx-1;
244                         break
245                     end
246                 end
247                 in_dead3(c3) = A_list_c(loc(ii));
248                 c3 = c3 + 1;
249             else %out trip in trip
250                 out_dead4(c4) = A_list_r(loc(ii));
```

```
                in_dead4(c4) = A_list_c(loc(ii));
                c4 = c4 + 1;
            end
        end
    end



    for xx = 2:nr_loc_charg+1 % for each charger locations
        chargers2{xx} = [];
    end
    trips = find(V_final(x_all,buses_loc(pp)) == 1);
    % find arcs driven to trips
    trips2 = A_list_c(trips);
    chargers = find(V_final(n_d+1:end-1,buses_loc(pp)) > 0);
    % find chargers used
    for yy = 1:numel(chargers)
        for xx = 2:nr_loc_charg+1
            if chargers(yy) ≤ pcl(xx)
                chargers2{xx} = [chargers2{xx} chargers(yy)-pcl(xx-1)];
                break
            end
        end
    end

    for hh = min(h_s{1}):max(h_e{1}) % for every minute of the day
        % deadhead trips SoC consumption
        if isempty(out_dead1) == 0
            d1 = find(charge_line_all(out_dead1(:,1)) == hh);
            if isempty(d1) == 0
                for ii = 1:numel(d1)
                    d_e = d_e + e_h(loc_charge(out_dead1(d1(ii),2)), ...
                        loc_charge(in_dead1(d1(ii),2)));
                end
            end
        end
        if isempty(out_dead2) == 0
            d2 = find(ht_end(out_dead2) == hh);
            if isempty(d2) == 0
                for ii = 1:numel(d2)
                    d_e = d_e + ...
                        e_h(loc_end(out_dead2(d2(ii))),loc_charge(in_dead2(d2(ii),2)));
                end
            end
        end
        if isempty(out_dead3) == 0
            d3 = find(charge_line_all(out_dead3(:,1)) == hh);
            if isempty(d3) == 0
                for ii = 1:numel(d3)
                    d_e = d_e + ...
                        e_h(loc_charge(out_dead3(d3(ii),2)),loc_start(in_dead3(d3(ii))));
                end
            end
        end
        if isempty(out_dead4) == 0
            d4 = find(ht_end(out_dead4) == hh);
            if isempty(d4) == 0
                for ii = 1:numel(d4)
                    d_e = d_e + ...
                        e_h(loc_end(out_dead4(d4(ii))),loc_start(in_dead4(d4(ii))));
                end
            end
        end


        % trips SoC consumption
        trips3 = find(ht_start(trips2) == hh);
        if isempty(trips3) == 0
            for ii = 1:numel(trips3)
                t_e = t_e + e_t(trips2(trips3(ii))); % SoC consumption update trips
            end
        end

```

```
321         % chargers SoC charged
322         for xx = 2:nr_loc_charg+1
323             chargers3{xx} = find(charge_line_2{xx-1}(chargers2{xx}-1) == hh); % start ...
                    of charge session
324             if isempty(chargers3{xx}) == 0
325                 eta = eta + dt_2{xx-1}(chargers2{xx}(chargers3{xx})-1)...
326                     *e_charge(xx-1)*...
327                     V_final(end-pcl(end)+pcl(xx-1)+chargers2{xx} ...
328                     (chargers3{xx}),buses_loc(pp));
329             end
330         end
331         SoC_level(pp,hh) = 100 - t_e - d_e + eta; % bus starts with 100 % SoC
332     end
333     total_energy(pp) = t_e+d_e; % Total energy charged
334     xx = (h_s{1}(1):h_e{1}(end))/60;
335     plot(xx,SoC_level(pp,h_s{1}(1):h_e{1}(end)),'LineWidth',2)
336     grid on
337     hold on
338 end
339 set(gca,'FontSize',16)
340 xlabel('Time [h]','FontWeight','bold')
341 ylabel('SoC of bus [%]','FontWeight','bold')
342 %% No. of chargers used plot
343 figure
344 for xx = 2:nr_loc_charg+1
345     counter = 0;
346     for ii = pcl(xx-1) + 2: pcl(xx)
347         c_p_m(xx-1,charge_line_2{xx-1}(ii-pcl(xx-1)-1)+1:...
348             charge_line_2{xx-1}(ii-pcl(xx-1))) ...
349             = b2_fix(ii)*ones(dt_2{xx-1}(ii-pcl(xx-1)-1),1)...
350             +zeros(dt_2{xx-1}(ii-pcl(xx-1)-1),1); % charger per minute
351
352
353     end
354     plot((charge_line_2{xx-1}(1):charge_line_2{xx-1}(end))/60,...
355         c_p_m(xx-1,charge_line_2{xx-1}(1):charge_line_2{xx-1}(end))...
356         ,'LineWidth',2)
357     grid on;
358     hold on;
359 end
360 set(gca,'FontSize',16,'FontWeight','bold')
361 xlabel('Time [h]')
362 ylabel('Number of chargers')
363
364 %% plot RMP
365 figure
366 plot(1:iter(1),RMP_Obj_val(1:iter(1),1),'LineWidth',3);
367 grid on;
368 hold on;
369 set(gca,'FontSize',18)
370 xlabel('Iterations [-]','FontSize',16,'FontWeight','bold')
371 ylabel('RMP value','FontSize',16,'FontWeight','bold')
372 %%
373 integrality_gap = Costs/min(RMP_Obj_val(:,1));
374 %% plot lowerbound
375 [bus_min,trips_per_time_cont] = Lower_bound(n_t,ht_start,...
376     ht_end,nr_bus,SoC_max,SoC_min,e_t,e_charge,e_max,c_p_m,nr_loc_charg);
```

Lower bounds

```
1  function [bus_min,trips_per_time_cont] = Lower_bound(n_t,ht_start,...
2      ht_end,nr_bus,SoC_max,SoC_min,e_t,e_charge,e_max,c_p_m,nr_loc_charg)
3
4  %% lower bound for buses based on trips per time continious
5  trips_per_time_cont = zeros(ht_end(end),1); % create zero matrix
6  for nr_t = 1:n_t % for each trip
7       for pp = ht_start(nr_t):ht_end(nr_t)-1 % for each time interval of a trip
8             trips_per_time_cont(pp) = trips_per_time_cont(pp) + 1;
9             % +1 trip in this time interval if a trip is driven during this
10            % time
```

```
11      end
12  end
13
14
15  bus_min = max(trips_per_time_cont); % determine min buses needed to drive all trips
16
17  %% lower bound based on energy in the system for chargers
18  nr_bus       = nr_bus; % Number of buses used in final schedule
19  n_s          = -1;         % Starting chargers = 0, if -1 is filled in
20  energy_error = 1;          % Variable that determines if lower bound is complete
21  while energy_error == 1    % While algorithm is not finished
22      n_s = n_s + 1;         % + 1 new fast charger
23      energy(ht_start(1)) = SoC_max * nr_bus; % calculate max energy at the start of the day
24      Chargers_used = zeros(ht_end(end),1);   % Number of chargers used
25      minu = 0; % minutes of duration charging session min 0 = 1 minute
26      for ii = min(ht_start):1:max(ht_end) % for each minute of the schedule
27          exceed = 0;
28          hh = find(ht_start ≤ ii);
29          trips = find(ht_end(hh) ≥ ii)'; % find trips that are driving at the moment
30          tripss = find(ht_start == ii);   % equal so the trip is finished
31          e = 0;
32          for uu = tripss(1:end) % determine energy lost
33              e(uu) = e_t(uu);   % energy loss
34          end
35          energy(ii) = energy(ii) - sum(e); % update total energy
36
37          if ii > ht_start(1)
38              % if charging time is shorter than already passed time
39              if energy(ii) < SoC_max*nr_bus
40                  % if the current energy is lower than the maximum energy
41                  if nr_bus > size(trips,2)
42                      % if there are more buses then trips driven at the tmoment
43                      if n_s > 0 % if there are fast chargers
44                          if nr_bus > size(trips,2)
45                              energy(ii) = energy(ii) + n_s * (max(e_charge)); % update ...
                                    energy
46                              Chargers_used(ii) = n_s; % determine chargers used
47                          else % if not all chargers can be used but only a part
48                              nr_charge = nr_bus - size(trips,2); % Determine how many ...
                                    chargers are used
49                              energy(ii) = energy(ii) + nr_charge * (max(e_charge)); % ...
                                    update energy
50                              Chargers_used(ii) = nr_charge; % update chargers used
51                          end
52                          if energy(ii) > e_max* nr_bus % energy exceed max SoC
53                              Energy_exceeded = energy(ii) - e_max* nr_bus;  % calculate ...
                                    energy exceeded
54                              Chargers_exceeded = Energy_exceeded/(max(e_charge));
55                              %  rewrite the energy to chargers
56                              Chargers_used(ii) = Chargers_used(ii) - ...
                                    floor(Chargers_exceeded+1e-3); % error factor
57                              energy(ii) = e_max* nr_bus; % update energy
58                              exceed = 1; % if exceeded no slow chargers needed
59                          end
60                      end
61                      if exceed == 0 % if energy is not exceed by fast chargers
62                          if nr_bus > size(trips,2) + n_s % if all fast chargers can be used
63                              nr_charge = nr_bus - n_s - size(trips,2); % determine no. ...
                                    of slow chargers used
64                              energy(ii) = energy(ii) + nr_charge * (e_charge(1)); % ...
                                    update energy
65                              Chargers_used2(ii) = nr_charge; % Number of slow chargers used
66                          end
67                      end
68                      if energy(ii) > e_max* nr_bus % energy exceed max SoC
69                          Energy_exceeded = energy(ii) - e_max* nr_bus;  % calculate ...
                                energy exceeded
70                          Chargers_exceeded = Energy_exceeded/(e_charge(1));
71                          %  rewrite the energy to chargers
72                          Chargers_used2(ii) = Chargers_used2(ii) - ...
                                floor(Chargers_exceeded+1e-3); % error factor
73                          energy(ii) = e_max* nr_bus; % update energy
74                      end
```

```
                end
            end
        end
        if energy(ii) < SoC_min*nr_bus % if energy is lower then min then new bus needed
            energy_error = 1; % Rerun algorithm with one extra charger
            break
        end
        energy(ii+1) = energy(ii);
    end
    if energy(ii) > SoC_min*nr_bus
        % if at the end of the day enough energy is left in the schedule
        break % end algorithm n_s is the lower bound for the number of chargers
    end
end



%% plots
% Plot lower bound for the number of buses
figure
hold on
grid on
xlabel('Time [h]')
ylabel('Vehicle number [-]')
plot((1:numel(trips_per_time_cont))/60,trips_per_time_cont)
axis([floor(min(ht_start)/60) ceil(max(ht_end)/60) 0 max(trips_per_time_cont)])
xticks(floor(min(ht_start)/60):1:ceil((max(ht_end)/60)))
yticks(0:1:max(trips_per_time_cont))

% Plot energy plot
figure
hold on
grid on
title(['kWh left at each time point for ',num2str(nr_bus),' buses'])
xlabel('Time [h]')
ylabel('Energy [kWh]')
plot((1:numel(energy))/60,energy,'LineWidth',1)
axis([floor(min(ht_start)/60) ceil(max(ht_end)/60) 0 ...
    ceil(max(energy)/10^floor(log10(max(energy))))*10^floor(log10(max(energy)))])
% round to order of maginitude on the y axis
xticks(floor(min(ht_start)/60):1:ceil((max(ht_end)/60)))

% Plot lower bound number of chargers used
figure
hold on
grid on
title(['Chargers used for ',num2str(nr_bus),' buses'])
xlabel('Time [h]')
ylabel('Chargers used [-]')
if any(Chargers_used) > 0 % if chargers are used
    plot((1:numel(Chargers_used))/60,Chargers_used,'LineWidth',1) % to hours
    hold on
end
% Plot number of slow charger used
% if any(Chargers_used2) > 0 % if chargers are used
%     plot((1:numel(Chargers_used2))/60,Chargers_used2,'LineWidth',1) % to hours
%     axis([min(ht_start)/60 max(ht_end)/60 0 max(Chargers_used2)])
%     xticks(floor(min(ht_start)/60):1:ceil((max(ht_end)/60)))% to hours
%     yticks(0:1:max(Chargers_used2))
% end



% %% lower bound buses based on energy for buses based on chargers
% available
% %%lower bound chargers
% n_s = 0;
% nr_bus = buses_used-1;
% for xx = 2:nr_loc_charg
% n_s = n_s + max(c_p_m(xx-1)); % Used chargers
% end
% n_slow = nr_bus; % number of slow chargers
```

```
% energy_error = 1; % determine if enough busses are available
% exceed = 0;
% while energy_error == 1 % while the energy is not below the min energy
%     nr_bus = nr_bus + 1; % + new charger
%     energy(ht_start(1)) = SoC_max * nr_bus; % calculate max energy
%     Chargers_used = zeros(ht_end(end),1); % chargers used
%     minu = 0; % minutes of duration charging session min 0 = 1 minute
%     for ii = ht_start(1):1:ht_end(end) % for each minutes
%         exceed = 0;
%         hh = find(ht_start ≤ ii);
%         trips = find(ht_end(hh) ≥ ii)';
%         tripss = find(ht_start == ii); % equal so the trip is finished
%         e = 0;
%         for uu = tripss(1:end) % determine energy lost per trip per time unit
%             e(uu) = e_t(uu); % energy loss
%         end
%         energy(ii) = energy(ii) - sum(e); % update energy
%
%         if ii > ht_start(1)
%             % if charging time is shorter than already passed time
%             if energy(ii) < SoC_max*nr_bus
%                 % if the current energy is lower than the maximum energy
%                 if nr_bus > size(trips,2)
%                  % if there are more buses then trips driven at the tmoment
%                     if n_s > 0 % if there are fast chargers
%                         if nr_bus > size(trips,2)
%                             energy(ii) = energy(ii) + n_s * (max(e_charge)); % ...
    update energy
%                             Chargers_used(ii) = n_s; % determine chargers used
%                         else % if not all chargers can be used but only a part
%                             nr_charge = nr_bus - size(trips,2);
%                             energy(ii) = energy(ii) + nr_charge * (max(e_charge)); % ...
    update energy
%                             Chargers_used(ii) = nr_charge;
%                         end
%                         if energy(ii) > e_max* nr_bus % energy exceed max SoC
%                             Energy_exceeded = energy(ii) - e_max* nr_bus;  % ...
    calculate energy exceeded
%                             Chargers_exceeded = Energy_exceeded/(max(e_charge));
%                             %                     rewrite the energy to chargers
%                             Chargers_used(ii) = Chargers_used(ii) - ...
    floor(Chargers_exceeded+1e-3); % error factor
%                             energy(ii) = e_max* nr_bus; % update energy
%                             exceed = 1; % if exceeded no slow chargers needed
%                         end
%                     end
%                     if exceed == 0 % if energy is not exceed by fast chargers
%                         if nr_bus > size(trips,2) + n_s % if all n_s chargers can be ...
    used
%                                 nr_charge = nr_bus - n_s - size(trips,2);
%                                 energy(ii) = energy(ii) + nr_charge * (e_charge(1)); ...
    % update energy
%                                 Chargers_used2(ii) = nr_charge;
%                         end
%                     end
%                     if energy(ii) > e_max* nr_bus % energy exceed max SoC
%                         Energy_exceeded = energy(ii) - e_max* nr_bus;  % calculate ...
    energy exceeded
%                         Chargers_exceeded = Energy_exceeded/(e_charge(1));
%                         %                     rewrite the energy to chargers
%                         Chargers_used2(ii) = Chargers_used2(ii) - ...
    floor(Chargers_exceeded+1e-3); % error factor
%                         energy(ii) = e_max* nr_bus; % update energy
%                     end
%                 end
%             end
%         end
%         if energy(ii) < SoC_min*nr_bus % if energy is lower then min then new bus needed
%             energy_error = 0; % variable that new bus is neded
%             break
%         end
%         energy(ii+1) = energy(ii);
%     end
```

```
213 %     break
214 % end
215 %
216 %
217 %
218 % %% plots
219 % figure
220 % hold on
221 % grid on
222 % title(['kWh left at each time point for ',num2str(nr_bus),' buses'])
223 % xlabel('Time [h]')
224 % ylabel('Energy [kWh]')
225 % plot((1:numel(energy))/60,energy,'LineWidth',1)
226 % axis([floor(min(ht_start)/60) ceil(max(ht_end)/60) 0 ...
227 %     ceil(max(energy)/10^floor(log10(max(energy))))*10^floor(log10(max(energy)))])
228 % % round to order of maginitude on the y axis
229 % xticks(floor(min(ht_start)/60):1:ceil((max(ht_end)/60)))
230 %
231 % figure
232 % hold on
233 % grid on
234 % title(['Chargers used for ',num2str(nr_bus),' buses'])
235 % xlabel('Time [h]')
236 % ylabel('Chargers used [-]')
237 % if any(Chargers_used) > 0 % if chargers are used
238 %     plot((1:numel(Chargers_used))/60,Chargers_used,'LineWidth',1) % to hours
239 %     hold on
240 % end
241 % if any(Chargers_used2) > 0 % if chargers are used
242 %     plot((1:numel(Chargers_used2))/60,Chargers_used2,'LineWidth',1) % to hours
243 %     axis([min(ht_start)/60 max(ht_end)/60 0 max(Chargers_used2)])
244 %     xticks(floor(min(ht_start)/60):1:ceil((max(ht_end)/60)))% to hours
245 %     yticks(0:1:max(Chargers_used2))
246 % end
```