APPLICATION-ORIENTED SCHEDULING PROBLEMS IN PUBLIC BUS TRANSPORTATION

A DISSERTATION SUBMITTED TO THE
DOCTORAL SCHOOL OF COMPUTER SCIENCE
OF THE
UNIVERSITY OF SZEGED
by
Balázs Dávid

Supervisor:
Dr. Miklós Krész

Szeged, 2018

Acknowledgments

First and foremost, I would like to thank my supervisor Miklós for his support and guidance throughout all these years. Although I might have doubted some of his ideas about the direction of this research, I must confess: he has been right the whole time.

I would also like to thank all my colleagues, co-authors and friends who contributed to the results presented in this dissertation.

Studying the application-oriented concepts that provide the backbone of this dissertation would not have been possible without the support of an industrial partner. I would like to thank Tisza Volán Zrt., the public bus transportation company of Szeged, Hungary, for providing the real-life data for testing our methods, the professional feedback on the quality of our results, and especially for the several innovative ideas about the shortcomings of the current state of transportation—many of these are reflected in the different topics of our theses.

Last, but not least, I would also like to thank my family. My girlfriend Edit for her love, support and most importantly her patience. My parents, who have always been there for me, especially my mother Tünde, whom I can always count on. And my brothers (all four of them), who are the best siblings one can wish for.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Public transportation networks are huge, complex systems, and companies that operate these face a large number of difficult optimization problems every day. According to a study on the spatial aspects of transportation by Rodrigue et al. [93], customer demands in public transportation are changing, and as a result, companies have to re-evaluate transport services they provide, and rethink the structure of their transportation networks. Meanwhile, the cost of building and operating public transportation systems increases, making it harder for them to provide a relevant alternative to urban mobility, while also remaining profitable businesses.

Besides the area of network planning and design, a large portion of the expenses comes from the operating costs of the company: these mainly include standing and running costs connected to the vehicles, and salaries of the drivers they employ. The efficient solution of problems arising in these areas can decrease the overall expenses of a company significantly, allowing them to stay competitive. To optimize their planning processes, most companies use computer aided decision support systems (see [77] for an example). These systems usually optimize problems connected to vehicles and drivers separately, and they also have to be able to create daily schedules and long-term plans as well. Naturally, they can also consider other problem types as well, like providing real-time solutions for unforeseen events (disruptions) happening to a daily schedule during its execution.

In this dissertation, we examine several optimization problems connected to vehicles in the fleet of a public bus transportation company. These cover a wide range of different problem types, including the creation of daily schedules, long-term planning, real-time management of unforeseen events, and generation of test instances. The topics are not only studied from a theoretical point of view, but their applicability in real life is also considered.

First, we discuss the optimization problems of transportation in Chapter 2, and introduce the basic concepts and notations of vehicle scheduling in Section 2.2. We also present the most important mathematical models from the literature that we will be referring to in future chapters. Our thesis topics are then discussed in the following chapters.

In Chapter 3, we introduce the concept of application-oriented vehicle scheduling, which aims to provide good quality solutions that also have an acceptable structure from a practical point of view. First, we give a series of variable fixing heuristics for creating vehicle schedules in a short time, then introduce an iterative algorithm that constructs schedules satisfying the basic rules of driver shifts as well. The list of our publications connected to this topic include [32, 34, 9].

Chapter 4 presents the integrated vehicle scheduling and assignment problem, which creates vehicle schedules that also consider vehicle-specific activities like refueling or parking. We give a set partitioning model for the problem, and a column generation framework for its solution. Our research connected to this chapter can be seen in [16].

Chapter 5 deals with addressing unforeseen events happening to pre-planned schedules of a transportation company. The problem is examined from two perspectives: first, we introduce a model and two heuristic solution methods for the multi-depot vehicle rescheduling problem. Then, we propose the concept of dynamic vehicle rescheduling, which evaluates the efficiency of solutions for a series of disruptions by considering their effects at the end of the day instead of treating them as separate problems. Our publications connected to these topics include [35, 37].

In Chapter 6, we introduce the schedule assignment problem for the long-term planning of bus transportation, where vehicles are assigned to daily schedules over a longer horizon. This assignment has to consider vehicle-specific tasks, such as parking at the end of every day, or regular preventive maintenance activities. We studied this problem in the following publications: [33, 38].

Chapter 7 contains three topics connected to rarely studied parts of an optimization system. First, we give the concept for a decision support framework for vehicle rescheduling that functions regardless of its implemented solutions methods. Then, we present a new modeling approach for problems in transportation scheduling: we give a timed automata-based model for the schedule assignment problem, which can be used for validating queries about certain scenarios, or for visualizing steps of the solution process. The last topic is an algorithm that generates random instances for vehicle scheduling problems that consider multiple depots, vehicle types, and activities connected to these types. These are studied in our following three publications: [12, 39, 34].

The dissertation is concluded with Chapter 8, where we also present our future research plans regarding transportation scheduling. The theses of this dissertation are collected into five groups defined by Chapters 3-7. Table 1.1 shows the connections between these groups and the key publications of the author.

Table 1.1: Relation between the thesis groups and the corresponding publications

	[32]	[9]	[34]	[16]	[35]	[37]	[33]	[38]	[12]	[39]
I.	●	●	●							
II.				●						
III.					●	●				
IV.							●	●		
V.	●								●	●

Chapter 2

Optimization problems in public transportation

Optimization problems arising in public transportation can be of many different types: from line planning to scheduling the tasks of vehicles and drivers, they form a large, interconnected system. Efficient solutions methods for these problems are extremely important, as they influence the effectiveness of a company, which also has a large impact on its expenses. Transportation networks can be of several types, all designed for different vehicles. As the dissertation is covering optimization problems connected to public bus transportation, this chapter will also review models and methods connected to a bus transportation system. Whenever the word *vehicle* is used throughout the dissertation, we will be referring to *buses* in the fleet of a public transportation company.

Depending on their nature, these arising problems can be categorized into three main phases:

- *strategic planning*, where the most important task is the design of the transportation network and the creation of its bus lines,
- *tactical planning*, with the aim of developing the underlying timetable of the system,
- and *operational planning*, that creates the schedules of vehicles and drivers, and defines driver rosters.

Naturally, different approaches also exist to these main topics. For example, Borndörfer [24] combines the first two phases into a single *service design* step, where they consider fare planning as well. They also present a fourth phase called *operations control*, which deals with the dispatching of vehicles and crew. While such differences might exist for other categorizations as well, the step of operational planning is always present, and includes the above mentioned problems for both vehicles and drivers. As we intend to study vehicle scheduling problems only, the others steps will not be reviewed in this chapter. A comprehensive overview of the problems connected to transportation systems is given by Desaulniers and Hickman in [41], or in a more recent and exhaustive review by Ibarra-Rojas et al. [63].

2.1 Operational planning of public transportation

In this section, we briefly overview operational planning, The subproblems of operational planning are not independent of each other: their relations can be seen in Figure 2.1:

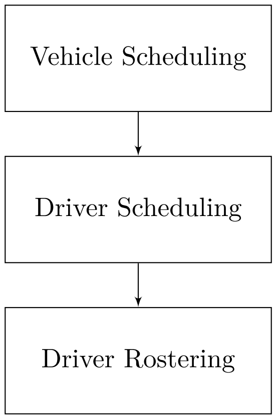


Figure 2.1: Subproblems of operational planning

Vehicle Scheduling gives a feasible daily schedule for a set of daily timetabled trips, considering the fleet of a transportation company. The resulting schedule has to service every input trip exactly once, and trips executed by the same vehicles cannot overlap in time. The goal of the problem is to minimize the number of vehicles in service, or the operating costs of these vehicles. It is important to note, however, that these vehicle schedules only determine the types of vehicle that should service certain groups of trips, but does not assign specific vehicles to these groups.

Driver Scheduling creates the daily shifts for the employees, satisfying all regulations connected to the daily working time of an employee. The most important such rules are the minimum and maximum working time limit on the length of the shifts, and the assignment of breaks in certain intervals where drivers can rest. This phase usually tries to minimize the number of drivers, or the total working time during the daily schedule.

These daily shifts are then used by *Driver Rostering* to create rosters over a longer planning period. The aim of this phase is to assign daily shifts to specific over a horizon of several weeks or months. Different driver rules considering a longer period also apply to this problem, like regulations on when drivers have to take a day off, or the minimum length of time that has to pass between two consecutive shifts of a driver. This phase usually tries to minimize the average under- and over-time of employees over the planning horizon.

An overview of the most important vehicle scheduling models is given by Bunte and Kliewer [25], while driver scheduling and rostering is reviewed by Ernst et al.[46].

It can be seen, that—while the above steps can be considered as independent, stand-alone tasks—these subproblems are connected, and all of their requirements should be considered when optimizing a transportation system. This can traditionally be done in two separate ways: with the sequential solution of the phases, or by integrating the different arising problems.

While integrating the entire process seems a natural approach for the solution of the subproblems, it often results in an intractable problem: as the subproblems are all NP-hard, their combined system of constraints is especially difficult to define and solve. However, the integration of smaller parts of this system has been extensively studied in the literature, combining different pairs of the subproblems introduced in this chapter. The most well researched such field is *Vehicle and Driver Scheduling*, which aims to create daily schedules by taking both vehicle and driver constraints into consideration.

A good example for this approach is presented by Steinzen et al. [96], but a detailed overview can also be seen of this field in [63], which presents both the most popular exact and heuristic methods for the problem. Other integration strategies also exist in the literature, although they are not as well studied as vehicle and crew scheduling. Ibarra-Rojas [63] gives a good overview of these approaches as well.

We presented a sequential solution framework in [7], and showed its effectiveness with a case-study using real-life data from the transportation company of the city of Szeged, Hungary. The major difference of our system from the general sequence of operational planning seen in Figure 2.1 is that we added the additional phase of *Vehicle Assignment* between vehicle and driver scheduling.

As it was mentioned before, vehicle scheduling only provides a theoretical solution, as the real-life vehicle in service are not specified in the resulting schedule. The aim of the vehicle assignment subproblem is to create vehicle schedules that also include the vehicle-specific tasks (such as parking and refueling) of buses executing the trips, which makes the practical application of these results possible.

This dissertation will present our work connected to both the vehicle scheduling and assignment phases of this system. For some of our early results studying the subproblems of driver scheduling and rostering, please refer to [8].

As this dissertation studies different problems connected to vehicle scheduling, the presented results can also be fit into a sequential solution framework. Such a system takes the subproblems introduced in Figure 2.1, and solves them in the given order, with the output of a problem being the input of the next step. This process provides great flexibility, as partial results can easily be obtained for the subproblems, and different solution approaches for certain steps can easily be integrated due to the modularity of the system. However, this might come at the price of efficiency: as subproblems do not consider constraints of other steps, their results will most likely require certain transformation steps to meet requirements of other subproblems.

In the following sections, we introduce the first subproblem of this sequential framework, and the main research topic of this dissertation, the vehicle scheduling problem.

2.2 The vehicle scheduling problem

In this section, we formally introduce the vehicle scheduling problem. While Section 2.2.1 gives the basic concepts of the problem, Sections 2.2.2 and 2.2.3 present its main variations, and introduce its important state-of-the-art models. The outline of this section and the notations follow our review paper [10].

2.2.1 The basic concept of vehicle scheduling

The input of the vehicle scheduling problem (VSP) is the *fleet of vehicles* of a transportation company, and the set of *timetabled trips* for a single day. Let these be represented by sets $V = \{v_1, v_2, \ldots, v_m\}$ and $T = \{t_1, t_2, \ldots, t_n\}$ respectively. Vehicles belonging to a set V are completely homogeneous, meaning that they are similar in every aspect and feature. If a fleet contains vehicles of several different types, set V is partitioned into disjoint subsets V_i $(i = 1, \ldots, k)$. Furthermore, let $c_{i,j}$ give the cost of vehicle i executing any task j. The goal of the problem is to assign the vehicles of set V to every timetabled trip in T such that the arising costs are minimal. Not every vehicle has to be assigned to a trip, and multiple trips can be assigned to a single vehicle. However, this must fulfill certain constraints: a vehicle has to be able to service all its assigned trips, and trips assigned to the same vehicle have to be compatible with each other.

Every trip $t \in T$ has a $dt(t)$ *departure time* and $at(t)$ *arrival time*, and also has a $sl(t)$ *starting location* and $el(t)$ *ending location*. Trips t and t' are *compatible*, if they can be serviced by the same vehicle without any conflicts. This basically requires $at(t) \leq dt(t')$, and $dt(t') - at(t) \leq time(el(t), sl(t'))$, where $time(el(t), sl(t'))$ gives the time needed for a vehicle to travel between $el(t)$ and $sl(t')$.

Figure 2.2 presents and illustrative example for the problem. The colored boxes in the figure represent trips, while dashed lines connect compatible pairs of trips. The right side of the figure shows a possible solution, where the trips are executed by three vehicles.

Vehicles start the scheduling period in *depots*, and return there at the end of the period. A vehicle can enter these depots any time during this period, when it has no assigned trips for a longer time. Depots represent garages, parking places, and similar locations, where vehicles can stay until the beginning of the next scheduling period.

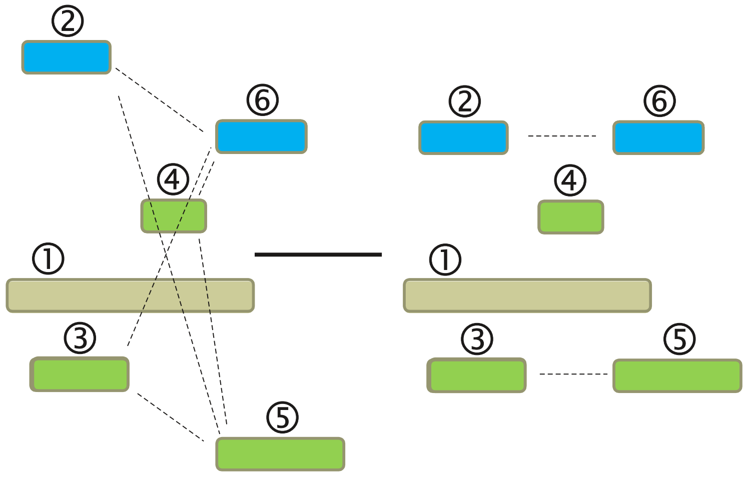


Figure 2.2: Simple example for the VSP

If the VSP has only one depot, it is called a *Single Depot Vehicle Scheduling Problem* (SDVSP). In the case of two or more depots, the problem is referred to as a *Multi-Depot Vehicle Scheduling Problem* (MDVSP). The MDVSP can also have an additional constraint for the execution of the trips called *depot-compatibility*: a subset of the depots is given for every trip t, and only vehicles belonging to this set are able to service t.

Vehicles can also travel between geographical locations without servicing a timetabled trip. Such activities are called *deadhead trips* (or deadheads in short), and usually take place between the ending location of a trip and the starting location of another one. Special cases of a deadhead trip include the *pull-out* and *pull-in* trips, which represent traveling activities from a depot to the starting location of a trip, or from the ending location of a trip to the depot.

A *vehicle block* is a series of pairwise compatible traveling activities. A vehicle block usually represents the set of daily tasks of a single vehicle, which is referred to as the *vehicle duty*. A feasible vehicle block always begins with a pull-out trip, and ends with a pull-in trip. The set of these blocks is called the *vehicle schedule*. Another solution of the above example extended with a single depot can be seen in Figure 2.3, where the dashed line represent the and appropriate travel activities (deadhead or pull-out/pull-in trips).

The basic task of the VSP is to give a vehicle schedule where every trip is executed exactly once by a vehicle satisfying its depot-compatibility constraint. The problem can either simply minimize the number of vehicles in service, or the costs of traveling can also be taken into account. If the objective function is the combination of the previous two terms, then the cost of a vehicle v belonging to depot d is $dc(d) + tc(d) \cdot dist(v)$, where $dc(d)$ is a one-time *daily cost* of a vehicle belonging to depot d, $tc(d)$ is the *travel cost* of the same vehicle to cover a unit distance (usually 1 km), and $dist(v)$ is the distance traveled by v.

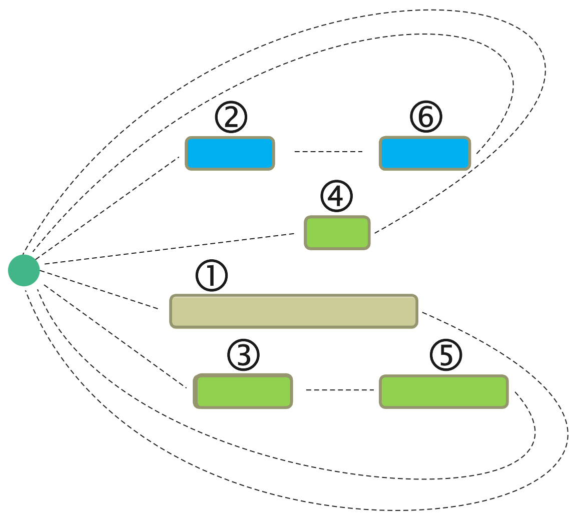


Figure 2.3: Example for a vehicle schedule

There is a large number of different mathematical models for the solution of the VSP in the literature. In the following sections, we will present some of the most important ones both for the SDVSP and MDVSP. A comprehensive review of this field is given by Bunte and Kliewer [25].

2.2.2 Single depot vehicle scheduling

The first solution method for the SDVSP was published by Saha [94]. He defined a partial ordering for the trips of T, and introduced an α ordering relationship. This relationship ensured that trip t' could only be serviced after t if an only if $el(t) = sl(t')$, and $at(t) \leq dt(t')$. It can be seen from the constraint that this model does not allow deadheads between two consecutive trips, which makes α weaker than the compatibility defined in the previous section.

There have been several bipartite graph-based models published for the SDVSP (see [25]). In the following, we will present the model and solution algorithm of Bertossi et al. [18], who considered the SDVSP as a matching problem. Let $G = (N_1, N_2, E)$ be a complete bipartite graph, where nodes $i \in N_1$ and $j \in N_2$ represent the arrivals of trip i and departures of trip j respectively. Let E be the set of edges, $|N_1| = |N_2| = |T| = n$ and $|E| = n^2$. Assume that E can be partitioned into subsets E_1 and E_2, where:

$$E_1 = \{(i, j) \mid t_i \text{ and } t_j \text{ are compatible trips},$$

$$E_2 = E \setminus E_1.$$

Edges $(i, j) \in E_1$ represent the possible deadhead trips between the arrival time of t_i and departure time of t_j, while all $(i, j) \in E_2$ edges correspond to two consecutive deadhead trips: one of these is a pull-in from the ending location of t_i to the depot while the other is the pull-out from the depot to the starting location of t_j. Using these concepts, it can be seen that the M perfect matching of graph G results in a possible vehicle schedule, where the number of required vehicles is $|M \cap E_2|$

Figure 2.4 presents the above defined bipartite graph for the simple problem introduced in Figure 2.2. The solid lines represent edges of set E_1, while dashed lines give edges of E_2.

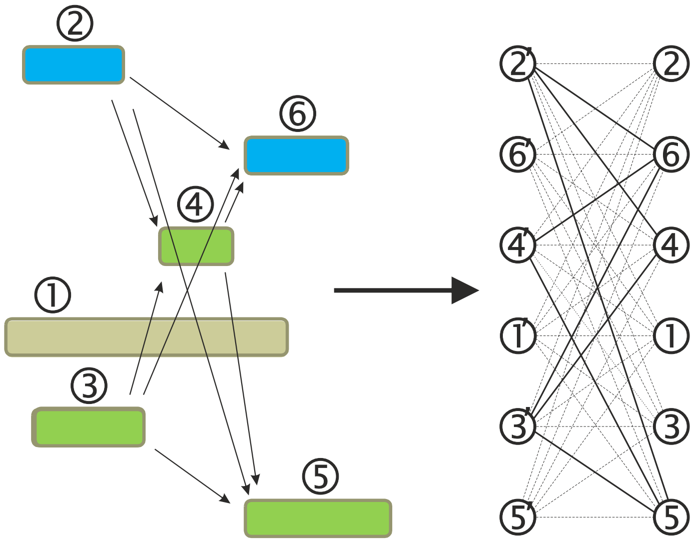


Figure 2.4: Bipartite graph for the SDVSP

In the case where every (i, j) edge is assigned a $c_{i,j}$ cost, the SDVSP can be solved by finding a minimum cost perfect matching in G. Depending on the costs of the edges, differently structured results can be achieved for the problem:

- If $c_{i,j} = 1$ for all $(i, j) \in E_2$, and $c_{i,j} = 0$ otherwise, then the objective value will give the minimum number of vehicles needed for the solution.
- If values of $c_{i,j}$ are chosen as the costs needed to perform the corresponding deadhead trips, then the solution will give the minimal travel costs.

Naturally, any combination of the above costs can be applied for the problem.

A limit k can also be introduced for the number of available vehicles. This results in a capacitated matching problem, the aim of which is to find a minimum cost matching in G that also satisfies $|M \cap E_2| \leq k$. The problem can be formally defined the following way: let $\boldsymbol{x}$ be a binary vector, where $\boldsymbol{x}_{i,j} = 1$ if edge (i, j) is in matching M, and $\boldsymbol{x}_{i,j} = 0$ otherwise. Consider a cost vector $\boldsymbol{c}$, and the set X of feasible solutions. Given an objective

$$min_{(i,j)}\{cx \mid x \in X, \quad x_{i,j} \in \{0, 1\}\}, \tag{2.1}$$

the problem is defined by the following constrains:

$$\sum_j x_{i,j} = 1, \quad i = 1, 2, \ldots, n \tag{2.2}$$

$$\sum_i x_{i,j} = 1, \quad j = 1, 2, \ldots, n \tag{2.3}$$

$$\sum_{(i,j)\in E_2} x_{i,j} \leq k. \tag{2.4}$$

The above problem can be transformed into a minimum-cost network flow. For this, simple modifications have to be carried out to the graph.

Additional edges have to be introduced for every trip t_i: the (i_1, i_2) edge has to be inserted between the departure node $i_1 \in N_1$ and arrival node $i_2 \in N_2$ belonging to t_i. Both upper and lower bounds of these edges have to be set to 1, ensuring that every trip is executed exactly once. As the additional costs belonging to these edges will only contribute a constant value to the objective function, this will not influence the result of the problem.

Similarly to the trips, departure d_1 and arrival d_2 nodes representing the depot also have to be added. These are connected by a circulation edge (d_2, d_1) of cost 0. If the number of available vehicles is limited for the problem, then the upper bound of this edge has to be set to k. Edges (d_1, i_1) and (i_2, d_2) also have to be introduced for every trip t_i, representing pull-out to and pull-in from the trip. These will replace the edges of E_2 in the network.

Because of these transformation steps, the result of the above network will be equivalent to that of the matching problem. As shown by Ahuja et al. [3], such a problem is solvable in polynomial time.

2.2.3 Multi-depot vehicle scheduling

The MDVSP was first proposed by Bodin et al. [19], and its NP-hardness was proven by Bertossi et al. [18]. The main advantage of the MDVSP over the SDVSP is that its constraints are much closer to the requirements of real-life problems. As it was mentioned in Section 2.2.1, depots traditionally represent garages or parking places, where vehicles start and end their daily blocks. This would suggest a homogeneous fleet with different starting locations, but the concept of the depots can be extended to include vehicle types as well. In real-life, restricting the types of vehicles that can service a trips is quite a common requirement: trips in some time intervals might always require larger buses because of a daily peak in passenger transportation, or trips belonging to some lines might only be serviced by smaller vehicles, because larger ones cannot navigate the narrow streets of that line.

We will be using the concept of depots to reference both the starting locations and the types of the vehicles: whenever we talk about vehicles belonging to the same depot, we will be referring to a set of vehicles that share the same type, and start the schedule at the same geographical location.

When the concept of a heterogeneous vehicle fleet has to be emphasized, a multi-vehicle-type scheduling problem (MVTSP) can also be considered. This problem is frequently treated as an alternative version of the MDVSP, as we introduced before. However, several papers have studied problems with multiple vehicle types in the past years. While Laurent and Hao [68] study the problem as a variant of the MDVSP, and give a powerful iterated local search method for its solution, Ceder [27] emphasizes the concept of multiple vehicle types, and uses a so-called deficit function (DF) approach to visualize the number of particular vehicles at each location in the system. A column generation-based solution framework was proposed for the MVTSP by Guedes and Borenstein [53]. As mentioned before, however, we will not be dealing with the dedicated MVTSP, rather incorporate the concept of vehicle types into our definition of depots.

In the following sections, we will present important modeling and solutions techniques for the MDVSP. Some of these reduce the problem to a multi-commodity network flow, and solve it as an integer programming problem (IP). Other approaches represent the MDVSP as a set partitioning, or set covering problem, studied for example by Ribeiro and Sumois [92] and Hadjar et al. [56]. Our notations and model formulations in the upcoming sections will be similar to that of Löbel [75].

Connection-based network

The main advantage of the connection-based multi-commodity network flow problem is that it clearly represents every possible connection between the trips of the MDVSP. In order to present the model, we have to introduce additional notations to the ones given in Section 2.2.1. Let D represent the set of depots, and $D_t \subseteq D$ give the depots compatible with trip $t \in T$. Furthermore, let $T_d \subset T$ denote the set of trips that can be services by vehicles from depot d. Let $sl(d)$ and $el(d)$ be the starting and ending node of every depot $d \in D$ respectively. Using the notations above, the set of nodes of our network can be defined by

$$N = \{dt(t) \cup at(t) \cup sl(d) \cup el(d) | t \in T, d \in D\}.$$

Let

$$E^d = \{(dt(t), at(t)) | t \in T_d\}, \quad \forall d \in D$$

give the set of trip edges that can be serviced by depot d, and let

$$B^d = \{(at(t), dt(t')) | t, t' \in T_d \quad \textit{are compatible}\}$$

be the possible deadhead edges of depot d.

Moreover, let

$$P^d = \{(sl(d), dt(t)), (at(t), el(d)) | t \in T_d\}, \quad \forall d \in D$$

give all the pull-in and pull-out edges of depot d. The circulation edge of depot d is given by

$$K^d = \{(el(d), sl(d))\}, \quad \forall d \in D$$

Based on the above sets, we can define the set of all edges belonging to depot d

$$A_d = E_d \cup B_d \cup P_d \cup K_d, \quad \forall d \in D,$$

and the set of all edges of the network

$$E = \cup_{d \in D} A_d.$$

An example for the above connection based network with two depots (represented by solid and dashed edges) can be seen in Figure 2.5.

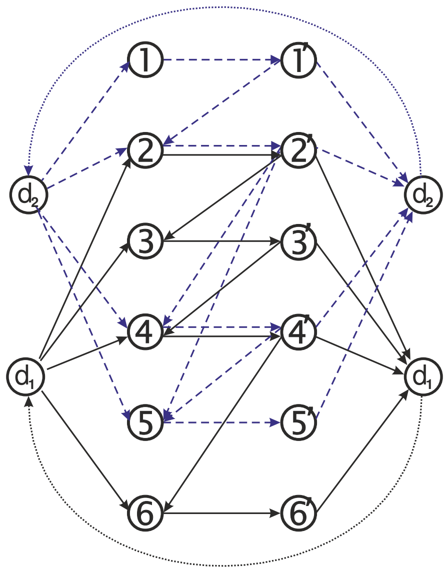


Figure 2.5: Structure of the connection-based network

Based on the sets introduced above, a solution can be determined for the MDVSP by using the network $G = (N, E)$. We define an integer vector $\boldsymbol{x}$. The dimension of $\boldsymbol{x}$ will be equal to the number of edges in the network. A vector component associated with edge $e \in E$ is denoted by x_e^d, if e belongs to depot d ($e \in A_d$). The value of x_e^d will be 1, if the edge is part of the solution schedule, 0 otherwise. The only exceptions to this are the depot circulation edges, as they can be included in a schedule multiple times.

The mathematical model of the problem can be formalized in the following way:

$$\text{minimize} \sum_{e \in E} c_e x_e,$$

s.t.

$$\sum_{d\in D_t, e=(dt(t),at(t))} x_e^d = 1, \quad \forall t \in T \tag{2.5}$$

$$\sum_{e\in\delta^+(n)} x_e^d - \sum_{e\in\delta^-(n)} x_e^d = 0, \quad \forall d \in D, \forall n \in N \tag{2.6}$$

$$x^d_{at(d),dt(d)} \le k_d, \quad \forall d \in D \tag{2.7}$$

$$x_e^d \in 0,1,(el(d),sl(d)) \text{ integer}, \quad \forall d \in D \tag{2.8}$$

where $\delta^+(n)$ is the set of outgoing, and $\delta^-(n)$ is the set of incoming edges of node n. The cost of edge e is given by c_e. Constraint (2.5) guarantees that each trip is serviced exactly once, while (2.6) satisfies flow conservation for all edges. The limit on the number of vehicles in each depot in enforced by Constraint (2.7).

The main drawback of the connection based model comes from its size. The number of compatible trips is high, even for a problem representing the transportation of a middle-sized city, and this results in a large number of possible deadhead trips. While the final solution will contain only a small percent of them, they cannot be omitted from the model, as that could lead to losing the optimal solution. This makes the model ineffective on real-life data, especially middle-sized or larger cities, where several thousand trips have to be scheduled together usually.

Time-space network

Decreasing the number of edges in our network can only be done by the modification of the underlying structure, and the time-space network does this effectively. Originally developed for aircraft fleet assignment [58], the time-space network was introduced to vehicle scheduling by Kliewer et al. [65]. It eliminates the drawback that comes from the size of the connection-based network, making it possible to solve larger-sized real-time MDVSP instances efficiently. As it was noted earlier, the number of edges connecting compatible trips in the connection-based model is high, but only a few of these are actually used in a feasible solution. However, if any of these connections were deleted from the model, the optimal solution might be lost in the process.

The model arranges problem data in two dimensions: time and space. The dimension of space corresponds to the set of geographical locations, and time-lines are introduced at each location that represent a sequence of events. The arrival and departure times of the trips are denoted on their corresponding time-lines. These give the N set of nodes for the model, which can be defined the exact same way as in the previous section. If there are nodes on a time-line that correspond to the same point in time, they can be merged together, decreasing the number of nodes. Edge sets E_d and P_d for every $d \in D$ are also defined similarly to Section 2.2.3.

The main innovation of the model is the way connections between the trips are modeled. While the connection-based network represents each compatible pair of trips with a dedicated edge, the time-lines of the time-space network can be used to aggregate a large number of these connections. This is done by the introduction of *waiting edges*. Let W_d be the set of these for every depot $d \in D$. Waiting edges follow the time-lines, and connect every subsequent pair of nodes. These edges will basically allow vehicles to wait at a location for the perfect connection, instead of leaving immediately with the first possible one.

The number of deadhead edges can be reduced because of the waiting edges introduced above. This is done by a two-phase aggregation process:

- *Finding first matches*: for every arrival node $at(t), t \in T$ on a time-line k, the first compatible departure node $dt(t'), t' \in T$ has to be determined at every other station $l \neq k$. Only the $(at(t), dt(t'))$ deadhead edges are introduced when connecting the nodes of time-lines k and l.
- *Finding latest first matches*: after the first aggregation phase, incoming deadhead edges have to be determined for every departure node $dt(t), t \in T$ on a time-line k. Out of all the incoming deadhead edges from nodes $at(t')$ on a station l, only the one with the latest time is retained between k and l for each $dt(t)$, the others are discarded.

The deadhead edges remaining in the network after the above process provide the new set B_d of deadhead edges. The set K_d of circulation edges also remains the same. Once all the above modifications have been introduced, we can give the set of edges belonging to commodity $d \in D$ of he time-space network by

$$A_d = E_d \cup B_d \cup P_d \cup K_d \cup W_d, \quad \forall d \in D.$$

and the set of all edges of the network is once again

$$E = \cup_{d \in D} A_d.$$

Figure 2.6 (taken from [15]) provides an example for such a network.

Using the above edges, the IP model of the time-space network can be given:

$$\text{minimize} \sum_{e \in E} c_e x_e,$$

s.t.

$$\sum_{d \in D_t, e=(dt(t),at(t))} x_e^d = 1, \quad \forall t \in T \tag{2.9}$$

$$\sum_{e \in \delta^+(n)} x_e^d - \sum_{e \in \delta^-(n)} x_e^d = 0, \quad \forall d \in D, \forall n \in N \tag{2.10}$$

$$x_{at(d),dt(d)}^d \leq k_d, \quad \forall d \in D \tag{2.11}$$

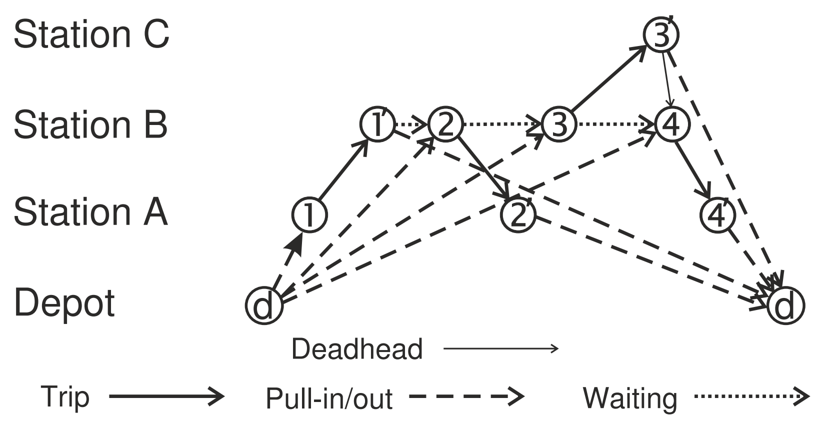


Figure 2.6: Structure of the time-space network (from [15])

$$x_e^d \geq 0, x_e^d \text{ integer}, \quad \forall d \in D \tag{2.12}$$

The above model is basically the same as the one presented in Section 2.2.3. The only difference is Constraint (2.12) that replaces Constraint (2.8).

A set partitioning model

This section introduces a set partitioning formalization of the problem based on Ribeiro and Soumis [92]. The MDVSP given for graph G in Section 2.2.3 can be reformulated using the circulation edges. Let H_d for all $d \in D$ be the set of p paths in G that start from $sl(d)$ and also return there. Let

$$H = \bigcup_{d \in D} H_d$$

be the set of all such paths in G. We introduce a binary variable y_p^d for all $p \in H_d$, such that

$$y_p^d = \begin{cases} 1, & \text{if } p \in H_d \text{ path is part of the solution} \\ 0, & \text{otherwise.} \end{cases}$$

Furthermore, let

$$a_{e,p}^d = \begin{cases} 1, & \text{if } p \in H_d \text{ path contains edge } e \\ 0, & \text{otherwise} \end{cases}$$

Let c_p denote the cost of path $p \in H_d$. Then the model can be formalized the following way:

$$\text{minimize} \sum_{d \in D} \sum_{p \in H_d} c_p y_p^d, \tag{2.13}$$

s.t.

$$\sum_{d \in D} \sum_{p \in H_d, e=(dt(t), at(t)) \in E_d} a^d_{e,p} y^d_p = 1, \quad \forall t \in T \tag{2.14}$$

$$\sum_{p \in H_d} y^d_p \leq k_d, \quad \forall d \in D \tag{2.15}$$

$$y^d_p \in \{0, 1\}, \forall d \in D, \quad \forall p \in H_d \tag{2.16}$$

Constraint (2.14) ensures that each trip edge is part of exactly one path in the solution, and Constraint (2.15) gives an upper bound on the number of vehicles in each depot.

Chapter 3

Heuristics for application-oriented vehicle scheduling

In Chapter 2, we discussed the most important models and algorithms for the solution of the MDVSP. These methods usually study the problem in a classical theoretical way, and then either solve it directly by mathematical programming methods, or by the use of different combinatorial heuristics. The main issue with this approach is that these methods usually do not consider the problems and their models as part of a transportation system, and are not interested in the structure of their solution, only their quality. Because of this, such approaches are not valid from an operational management point of view: in a transportation system, solutions for individual problems are only part of a more complex process, and they are often used to aid experts in making decisions.

We presented the subproblems of operational planning in Chapter 2, and showed that solving a VSP is the first phase of this area. In this Chapter, we present two application-oriented heuristic approaches connected to the VSP. We call these methods application-oriented, because they consider both problem structure (for easy application in real-life) and running time (to be suitable for decision support) besides the quality of the solutions.

First, we review the literature of MDVSP heuristic methods in Section 3.1, then present our proposed algorithms for the solution of the problem.

The first method in Section 3.2 uses the concept of variable fixing to solve the VSP: it tries to reduce its problem size by finding trips that are likely to be in the same sequence in the final solution, and combines such groups into single, long trips. The resulting smaller problem is solved using the classical IP modeling approach, and solutions are obtained with a greatly reduced running time. This method is studied in our following publications: [32, 34].

The second method in Section 3.3 is an iterative algorithm that produces vehicle schedules with a structure that also satisfies basic driver scheduling constraints. First, a classical VSP is solved, and the result is modified with a cut-and-join heuristic. This process iterates until all trips of the input are part of a feasible a vehicle block. We introduced this method in [9].

3.1 An overview of selected MDVSP heuristics

In this section, we review different heuristic approaches developed for the MDVSP.

An early review by Daduna and Paixão [31] describe a process called 'interactive alterations' of a raw vehicle schedule created as a working basis.

Gintner et al. [51] proposed a two-phase heuristic for solving large instances of the MDVSP by decreasing its model size. Their main idea was to solve simplified SDVSP problems, and use their results to fix variables representing sequences of trips in the mathematical model. This smaller problem is then solved to optimality. Details of this approach will be introduced in Section 3.2.

Suhl et al. applied a rounding algorithm for the problem in [98], which they call relaxation-based search space heuristic. Their main idea is to solve an LP relaxation of the problem, and round variables that have 'quasi-integer' value. Multiple rounding iterations can be performed after each other, as long as they yield a feasible LP solution.

Pepin et al. [84] compare five different approaches: truncated branch-and-cut—which they achieve using the CPLEX solver on the time-space model of the problem—, a Lagrangian heuristic, truncated column generation, a large neighborhood search, and a tabu search algorithm.

Laurent and Hao [68] give an iterated local search that constructs its initial solution using an auction algorithm, then selects neighbors using 'block-moves'.

Otsuki and Aihara [82] propose a local search algorithm that utilizes a variable depth search framework, speeding up its solution with pruning and deepening techniques.

A two-stage heuristics is presented by Guedes et al. [55]. They reduce the problem size by solving either a series of SDVSP problem, or a relaxed MDVSP, and truncated column generation is used for the resulting model.

3.2 Heuristic size reduction with variable fixing

In this section, we present several variable fixing heuristic for the MDVSP that take both operational costs and the application-oriented structure of the vehicle schedules into consideration. Development of these methods was also done with an interactive decision support system in mind, where a short running time and well structured schedules are both important factors for a solution.

Variable fixing is a well known technique for speeding up the solution of MIP problems, and has two major variations. The first approach is more commonly known as a rounding heuristic. It uses the iterative solution of the relaxed mathematical model, with a fixing phase at the end of each solution step: the values of certain 'quasi integral' (but still fractional) variables are rounded to the nearest integer. The other approach fixes the values of certain variables based on some structural property of the problem.

A two-phase heuristic proposed by Gintner et al. [51] uses variable fixing for the solution of large MDVSP instances by decreasing the model size. This is done by fixing variables representing trip sequences in the mathematical model, and then solving the resulting new problem to optimality. Because of the two-step solution process, they call this approach fix-and-optimize. The variables to be fixed are selected by analyzing the solutions of several simplified models of the original problem. This is done by finding series of trips that appear in all the simplified solutions. If such trips exist, it is presumed that they are likely to appear in the global optimum in the same way. Such a sequence of trips is referred as a 'stable chains', and is used as a single trip in the model of the MDVSP.

Simplified subproblems are obtained by decomposing the original MDVSP into an SDVSP for every depot. An SDVSP for a single depot d is constructed and solved in the following way:

- The total vehicle capacity of the SDVSP is equal to the sum of all depot-capacities of the MDVSP.
- Only those trips are considered that can be executed from depot d.

After all SDVSP sub-problems are solved, their solutions are used to create the 'stable chains'. If the same sequence of trips can be found in all solutions, then it is considered as such a chain. Using these 'stable chains' as single trips, a new, smaller MDVSP model is built that has the following properties:

- The number and capacity of the depots are the same as in the original problem.
- The set of trips of the new problem consists of the trips that are not included in any of the stable chains, and a newly created trip for each 'stable chain'. The cost of such a new trip is the sum of the costs of all the trips that its chain represents. The departure time and starting location of the first trip of the chain, and the arrival time and ending location of the last trip of the chain are used as as the starting and ending data of this new trip. These new trips can be executed from any depot.

After this new MDVSP is solved, the trips in the 'stable chains' have to be substituted back instead of the new trips to acquire the final solution.

We chose to develop several heuristics based on the above idea of variable fixing, because it models the application oriented aspect of the problem: fixing trips 'that should belong together in the final solution' in the same chain. During the creation of these chains, we also have control over several real-world constraints, like limiting the amount of time between two consecutive trips of a chain by not adding a possible trip to a chain if that would leave too little, or too much gap in between.

In [34], we proposed the solution of a simplified model of the problem that we call a 'quasi-multiple depot' model. Though this model uses only a single depot, two trips are connected only if they would be connected in the multiple depot case as well. This means that the trips have to be compatible, and they must also share a common depot from which they can be served. The cost of the arc between these two trips in our model is calculated using the cheapest possible cost of all their common depots. The capacity of the depot is the sum of the capacities of all depots in the original problem. Pull-out and pull-in arcs of the depot have the weight of the minimal deadhead trip from and to any of the depot locations of the original problem. Once this 'quasi-multiple depot' model is constructed, it is solved by an MILP solver.

We experimented with three different approaches for finding stable chains in the solution of the above problem:

- building chains with regards to depot costs,
- fixing trips with the same depot-compatibility, and
- assigning trips of the same bus-line to a chain.

A preliminary version of the first approach was also studied in [32], where we showed its efficiency by comparing its results to other MDVSP heuristics.

The above methods are be presented in the following sections, and the difference of their resulting 'stable chains' is also illustrated on a small example. Their results are then compared to each other on real-life and random test instances.

Common notations for all three approaches include:

- C is the set of fixed chains.
- F is the set of fixed trips that will not be considered in future chains.
- L is the set of trips that represents the chain that is currently being built.
- the function $nextTrip(t)$ will return the trip that follows t in the solution of the 'quasi-MDVSP' problem.

3.2.1 A greedy approach using depot costs

The first variable fixing approach we developed aims to create 'stable chains' based on the cost of servicing their included trips. Depots are sorted in ascending order based on the following cost:

$$\frac{1}{\epsilon} \cdot cost(daily) + cost(km)$$

where $cost(daily)$ is the daily cost of a single vehicle from the depot, $cost(km)$ is the cost of that vehicle to travel 1 km, and $\epsilon > 0$ is a parameter.

For every depot in this order, the algorithm examines all the vehicle schedules in the solution of the 'quasi-MDVSP' problem. If subsequent trips are found that can be executed from the current depot, they are considered to be in the same 'stable chain'. These trips are flagged, and cannot be the part of other 'stable chains'. The description of this algorithm can be seen in Algorithm 1.

Algorithm 1 Variable fixing using depot costs.

```
 1: Determine the order $D$ of depots
 2: $C \leftarrow \emptyset, F \leftarrow \emptyset, L \leftarrow \emptyset$
 3: for each $d \in D$ do
 4:     for all trips $j \notin F$ in the solution do
 5:         while $j$ can be executed from $d$ do
 6:             $L \leftarrow j$
 7:             $F \leftarrow j$
 8:             $j = nextTrip(j)$
 9:         end while
10:         if $|L| > 1$ then
11:             $C \leftarrow L$
12:         end if
13:         $L \leftarrow \emptyset$
14:     end for
15: end for
16: return $C$
```

This algorithm is only the basic outline of finding the chains, further constraints can also be introduced:

- The number of trips in a chain can be limited.
- A maximum duration in time can be set for a chain.
- The minimum/maximum gap in time between two consecutive trips of the chain can be given.

Experience shows that limiting the maximum duration of these 'stable chains' results in a solution with better cost, but has an increased running time. The running time of the heuristic was very fast, even when additional constraints are introduced, but the quality of the solutions was far from what we have expected. Because of this, further changes have been experimented with to improve the solution at the expanse of a minimal increase in the running time.

3.2.2 A greedy approach using depot compatibility

The approach in Section 3.2.1 showed that greedily choosing the chains with the lowest cost does not yield good quality solutions. Because of this, we experimented with fixing trips that have some property in common instead of using a cost function.

The aim of our second approach is to construct 'stable chains' based on similar depot-compatibilities of the trips. We first examine those trips from the solution of the 'quasi-MDVSP' that can be executed from all d depots, then all that are compatible with $d-1$ depots, and so on. Two subsequent trips are assigned to the same chain if they have exactly the same depot-compatibility in the solution. This algorithm is described in Algorithm 2.

Algorithm 2 Variable fixing based on depot-compatibilities.

```
1: $C \leftarrow \emptyset, F \leftarrow \emptyset, L \leftarrow \emptyset$
2: for $d = numOfDepots$ downto 1 do
3:     for all $j \notin F$ trips compatible with exactly $d$ depots do
4:         $L \leftarrow j$
5:         $F \leftarrow j$
6:         $k = nextTrip(j)$
7:         while $depots(j) = depots(k)$ do
8:             $L \leftarrow k$
9:             $F \leftarrow k$
10:            $j = k$
11:            $k = nextTrip(j)$
12:        end while
13:        if $|L| > 1$ then
14:            $C \leftarrow L$
15:        end if
16:        $L \leftarrow \emptyset$
17:    end for
18: end for
19: return $C$
```

The value $numOfDepots$ represents the number of depots of the problem, and $depots(t)$ gives the set of depots that are able to serve trip t. The main difference of this algorithm from the previous one is the flexibility of its chains. While a chain that was fixed based on a cost function will only be compatible with depots in the intersection of the depot sets of all its trips (which usually means only 1 or 2 depots), most of the chains fixed with this approach will be compatible with a large number of depots, which leaves more options for the solver to work with. This can be seen on the quality of the test results as well.

The additional constraints that we proposed for Algorithm 1 can also be introduced here.

3.2.3 Exploiting the structure of real-world problems

Our third approach applies a method for constructing its 'stable chains' that is similar to the real-world schedule building practice of transportation companies. A driver usually uses the same vehicle during a shift, and consequent trips of the same bus-line are assigned to the same shift. Some line changes might occur during the day, but their number remains low. In contrast, the solution of any classical MDVSP mathematical model results in vehicle schedules that have a high number of line changes. Applying such a schedule in practice is not efficient: though it cannot be modeled directly in operational costs, the frequent line changes put a big pressure on the drivers.

We developed an algorithm that builds 'stable chains' using the above observation: only those trips are fixed in a chain that belong to the same bus-line. A limit is also given for the maximum idle time between such trips. If two subsequent trips belong to the same bus-line, but are far from each other in time, then they are not fixed in the same chain. The description of this method can be seen in Algorithm 3.

Algorithm 3 Variable fixing based on same bus-lines.

```
1: $C \leftarrow \emptyset, F \leftarrow \emptyset, L \leftarrow \emptyset$
2: **for** all $j \notin F$ trips **do**
3:     $L \leftarrow j$
4:     $F \leftarrow j$
5:     $k = nextTrip(j)$
6:     **while** $line(j) = line(k)$ and $depots(j) \cap depots(k) \neq \emptyset$ and $dt(k) - at(j) < limit$ **do**
7:         $L \leftarrow k$
8:         $V \leftarrow k$
9:         $j = k$
10:        $k = nextTrip(j)$
11:    **end while**
12:    **if** $|L| > 1$ **then**
13:        $C \leftarrow L$
14:    **end if**
15:    $L \leftarrow \emptyset$
16: **end for**
17: **return** $C$
```

The departure and arrival time of a trip t are represented by $dt(t)$ and $at(t)$ respectively, the set of depots that can service t is defined by $depots(t)$, and $line(t)$ gives the bus-line of t. The value of $limit$ is the maximum time limit between two trips of a chain.

The resulting chains will contain trips that are close to each other in time, and the compatible depots of such chains will also be similar to the depots of the trips that construct them; in a real-world case, trips of the same line usually have the same depot-compatibilities.

3.2.4 An illustrative example

In this section, we show the difference between the three variable fixing approaches presented above. As a simple example, let us consider a vehicle scheduling problem with 3 depots, 8 trips and 4 geographical locations (A,B,C,D).

Table 3.1 gives every detail of the trips, including their starting and ending geographical locations (From, To), departure and arrival times, and the depots that they are compatible with. Furthermore, suppose that trips between the same pairs of geographical locations belong to the same bus-line.

Table 3.1: Details of trips illustrating variable fixing.

Trip	From	To	Departure	Arrival	Depots
1	C	B	10	12	1,2,3
2	B	A	12	14	1,2,3
3	B	C	12	14	2,3
4	A	B	14	16	1,2,3
5	B	A	16	18	1,2
6	A	B	18	20	1,2
7	C	D	22	24	2,3
8	D	C	24	26	2

This gives us the following three lines:

- Line A-B: trips 2,4,5,6;
- Line B-C: trips 1,3;
- Line C-D: trips 7,8.

Let the deadhead distance between any pair of geographical locations, and the pull-in and pull-out distance for all depots be 2 minutes. The depot costs of the problem are the following:

- Depot 1: 100 daily cost and 10/minute distance cost;
- Depot 2: 200 daily cost and 20/minute distance cost;
- Depot 3: 300 daily cost and 30/minute distance cost.

The structure of the above problem can be seen on Figure 3.1. The horizontal lines represent the geographical locations, while the arrows between them correspond to the trips. The solution of the 'quasi-multiple depot' problem for all three approaches results in the following two vehicle blocks:

- Blocks 1 executes trips 1,2,4,5, and 6.
- Blocks 2 executes trips 3,7, and 8.

All approaches construct their 'stable chains' based on these blocks. Applying the *method based on depot costs*, the cost function will give the depot order 1,2,3 for an arbitrary $\epsilon > 0$. Using this order, the following chains are constructed:

- Chain 1: trips 1,2,4,5,6;
- Chain 2: trips 3,7,8.

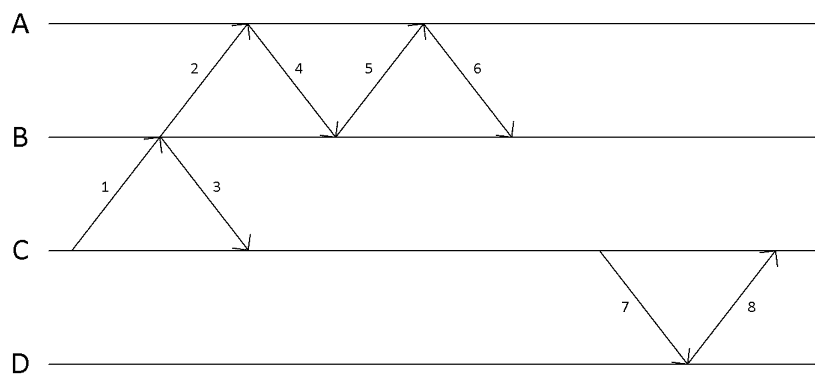


Figure 3.1: Structure of the problem illustrating variable fixing

If chains are built with regards to *depot-compatibility*, the first trips to be examined are the ones that are compatible with all 3 depots, then the trips that are compatible with 2 depots, and finally the trips that are compatible with 1 depot only. This results in the following chains:

- Chain 1: trips 1,2,4;
- Chain 2: trips 5,6;
- Chain 3: trips 3,7;
- Chain 4: trip 8.

If *bus-lines* are considered during the creation of the chains, the following trips are fixed together by the approach:

- Chain 1: trips 2,4,5,6;
- Chain 2: trip 1;
- Chain 3: trip 3;
- Chain 4: trips 7,8.

It can be seen from the above examples that the first approach creates two chains, which limits the results model to a single feasible solution, with these chains acting as the two blocks of its schedule. Contrary to this, the other two approaches have four chains, which gives more solution options for the resulting model. While the size of these resulting problems is the same, the chains given by the third approach are more flexible, as almost all solutions that can be achieved by the second approach can also be produced by the third one as well.

3.2.5 Test results

The different variable fixing approaches discussed earlier were tested on real-life data provided by the bus transportation company of the city of Szeged, Hungary, as well as on random input generated by the method that will be described in Section 7.4.2. This section presents the results of these instances.

Real-life instances

The real-life instances provided by the transportation company use 11 different day-types, and solutions are presented for all of these. Along with the day-types of the instances and their number of trips, the most important properties of their optimal solutions is also presented in Table 3.2: the number of vehicles and the running time in seconds. The optimal solutions were achieved by solving the time-space networks of the problems with the SYMPHONY solver. The problems include several different workday-types (called Weekday), Saturdays, a Sunday, and a holiday.

Table 3.2: Optimal VSP solutions of the real-life instances.

Instance	Day-type	Trips	Running time(s)	Vehicles
szeged1	Weekday	2724	872	96
szeged2	Sunday	1768	431	44
szeged3	Weekday	2724	1053	97
szeged4	Saturday	1981	276	55
szeged5	Saturday	1984	250	55
szeged6	Weekday	2723	1381	96
szeged7	Weekday	2690	1297	95
szeged8	Weekday	2724	860	97
szeged9	Weekday	2720	869	97
szeged10	Weekday	2723	1179	96
szeged11	Holiday	1646	180	43

As it can be seen from the table, the running times of the weekday instances can reach 20-30 minutes, and solving all the 11 day-types of the company to optimality would take about 8500 seconds. The 2-2.5 hours of running time for calculating the vehicle schedules is not a problem when creating plans for a longer horizon, but might be too long when considering a decision support system, where quick solutions are needed to test different input configurations, or the effects of structural changes to the resulting schedules. We will examine the following aspects of the results given by the heuristic approaches:

- The optimality gap between the results of the heuristics and the respective optimal solutions given by the MDVSP for the same problem.
- The ratio of the running time of the heuristics compared to the running time of the IP solutions.

The results of the variable fixing heuristic of Gintner et al. can be seen in Table 3.3. Every solution shows a decrease in running time compared to the optimal solutions: the average running time of the instances is about 40% of the original, which would mean a running time of 3000-3500 seconds (almost 1 hour) for all the vehicle schedules. The gap from the optimum varies between 0.25%-0.40%, with an average of 0.33%.

Table 3.3: Variable fixing results using Gintner et al.

Instance	Gap	Running time ratio	Vehicles
szeged1	0.27%	57.45%	96
szeged2	0.41%	31.55%	44
szeged3	0.33%	47.67%	96
szeged4	0.35%	36.59%	55
szeged5	0.37%	42.80%	56
szeged6	0.27%	28.10%	96
szeged7	0.35%	51.50%	96
szeged8	0.33%	52.56%	96
szeged9	0.33%	50.75%	96
szeged10	0.25%	34.69%	96
szeged11	0.34%	34.44%	44

If chains are built based on the proposed cost function, the average running time decreases below 10% of the time needed for the optimal solution; this means that most results can be obtained in several minutes (in case of our test cases, it is at most 4-5). The combined running time of all 11 day-types is between 10-15 minutes, which is really fast. However, the gap from the optimal solution has risen significantly: it was greater than 2.5% in some cases, with an average of 1.83%. As opposed to the method of Gintner et al., the greedy approach fixes significantly more trips ($\sim$ 66% in comparison to $\sim$ 33%) into 'stable chains'. This greatly reduces the size of the problem, and is the main reason behind the fast running time. However, the method is also less precise because of the fact that more trips are fixed in chains. If the construction of these chains is limited by some of the above mentioned alternative constraints (e.g. limit on the size/length of the chains, or the types of chosen trips), will lead to a solution with a better cost. On the other hand, less fixed trips also mean a greater problem size, which results in an increase in running time. The results of this method are presented in Table 3.4.

Building the chains based on depot-compatibility results in solutions with a more ordered structure than the previous two methods. Although more trips remain single, which comes with a slight increase in running time (in average 11.63% of the IP solution, which is about 15 minutes for all day-types), solutions are still achieved quite fast. In addition to this, the gap is also significantly smaller: it is at most 1.26%, with an 0.86% average. Including additional constraints also result in a better solution at the expense of an increase in running time, however, the quality improvement

Table 3.4: Variable fixing results based on depot costs.

Instance	Gap	Running time ratio	Vehicles
szeged1	2.63%	9.29%	100
szeged2	1.20%	3.71%	46
szeged3	2.15%	5.22%	99
szeged4	1.01%	10.14%	57
szeged5	1.12%	10.40%	57
szeged6	2.30%	6.08%	99
szeged7	2.20%	6.17%	97
szeged8	2.03%	10.70%	98
szeged9	2.35%	11.85%	99
szeged10	2.32%	8.40%	98
szeged11	1.04%	14.44%	46

is not significant enough to justify this. The results of this approach can be seen in Table 3.5.

Table 3.5: Variable fixing results based on depot-compatibility.

Instance	Gap	Running time ratio	Vehicles
szeged1	1.14%	12.84%	97
szeged2	0.34%	6.50%	44
szeged3	1.19%	8.74%	98
szeged4	0.43%	15.58%	56
szeged5	0.38%	16.00%	56
szeged6	1.14%	6.95%	97
szeged7	1.06%	6.63%	96
szeged8	1.11%	13.95%	97
szeged9	1.14%	10.24%	98
szeged10	1.26%	19.42%	97
szeged11	0.29%	11.11%	43

Using trips of the same bus-line for building chains will result in an overall lower number of fixed trips, which leads to a decreased running time of 21.78% of the original. Solving all day-types with this approach requires about 25 minutes. Gaps from the optimum are more favorable than the ones of our previous two approaches, with an average of 0.60%. The results of this method are found in Table 3.6.

The analysis of the above results reveals an interesting property: the approaches that build their chains using the structure of the problem (either the depot-compatibilities or the bus-lines) perform much better on the weekend/holiday day-types than the weekday ones. While the heuristic of Gintner et al. gives solutions with a better gap on average, considering even such a simple real-life property as the bus-lines of the trips will result in solutions with a significantly better gap and

Table 3.6: Variable fixing results based on bus-lines.

Instance	Gap	Running time ratio	Vehicles
szeged1	0.58%	16.28%	97
szeged2	0.03%	21.35%	44
szeged3	0.83%	19.66%	97
szeged4	0.23%	22.46%	55
szeged5	0.22%	28.00%	55
szeged6	0.83%	9.56%	96
szeged7	1.06%	12.80%	96
szeged8	1.10%	17.91%	96
szeged9	0.77%	15.19%	97
szeged10	0.59%	10.86%	97
szeged11	0.18%	65.56%	44

running time, if the input is structured that way.

We provided several different variations for the variable fixing heuristic, which can all be easily included in a decision support system. Depending on the requirements (quality versus a quick solution) of the problem that has to be solved, the most adequate approach can be chosen for it.

Solutions on random data input

As we mentioned earlier, the approaches were also tested on random data instances generated by the method in Section 7.4.2. We used several instance sets in different problem sizes of 50, 250, 500 and 1000 trips, each set containing 10 randomly generated input in the given size. Out of the 4 methods presented above, both the heuristic of Gintner et al. and our approach of using bus-lines rarely fixed any trips, and even when they did, they only found one or two small chains. This means that both methods ended up solving the original (or almost exactly the same) MDVSP, thus the quality of their results cannot be assessed properly.

However, we further analyzed the connection between the quality of the approaches and the structure of the problem, and came to the following conclusions: the heuristic of Gintner et al. needs a large number of trips in the input that can be executed from any of the depots. Besides this, these trips also have to be close enough to one another in time so that every SDVSP sub-problem schedules them in the same sequence. If the trips that are compatible with every depot are scattered on the time-line of the problem, then it is likely that none, or only a small number of them will be fixed in chains. This scenario is likely to happen in the case of the proposed random instances, which explains the failure of the heuristic in finding chains.

The method based on bus-lines has the same problem on this randomly generated input. Real-life instances always have different bus-lines, which are represented by given pairs of p and q geographical locations that have trips occurring back and forth between them with a given frequency. Randomly generated instances will not have this kind of order in their timetable, thus this heuristic is very likely to fail too.

The results of the other two heuristics can be seen in Table 3.7. The column marked with (cost) represent the heuristic that uses a cost function, while the column marked with (depot) give results for the heuristic based on depot-compatibility. Each row of the table gives the average results of the 10 problems in that instance set. The heuristic using a cost function provides approximately the same results as on the real-life instances, while the heuristic based on depot-compatibility fixed fewer trips than usual. As this method also uses a structural property of the input, it also has a more difficult time with finding chains. Running times were close to that of the solution of the original IP model, and they are not included in the table because of this. The poor running time comes from the fact that the methods could fix only a small number of trips into 'stable chains', and ended up solving an MDVSP similar in size to the original one.

Table 3.7: Variable fixing solutions on random instances.

Instance	Gap (cost)	Gap (depot)
random_50	0%	0%
random_100	0%	0%
random_500	1.57%	$9 * 10^{-6}$%
random_1000	1.54%	0.02%

Test experience on these random instances shows that the data generated by our method is still different from real-life instances in many structural aspects. Heuristics that are based on structural properties appearing in real-life problems cannot be applied effectively to most of the generated input. This means that while the instance generating methods found in the literature might work well when looking for the optimal solutions of a theoretical mathematical model, they still fail to create problems that closely resemble real-life instances in structure.

3.3 'Driver-friendly' vehicle scheduling

As it was mentioned earlier, the solution of the classical vehicle scheduling problem can only be considered as a theoretical schedule from a practical point of view. The blocks of the solutions are usually too dense for a single person to execute them, and because of this, drivers have to change their assigned buses several times a day. This procedure is a demanding task for the drivers, and also requires the introduction of certain driver events (e.g. administration). Because of this, we presented an approach in [9] that is closer to the practice used by public transportation companies: drivers are assigned a single bus to use during their shift.

The method is an iterative heuristic that is composed of multiple phases. The first step is the solution of the classical VSP for the trips of the input timetable, which can be done by any of the before mentioned methods. The second phase uses the blocks of this schedule as an input, and modifies them to satisfy some of the most basic driver constraints: the assignment of breaks during the day, and the maximal driver shift length. The blocks of the schedule are sorted into different classes according to their length, and blocks belonging to certain classes are cut into two parts. Matching parts are then joined together to form feasible shifts with regards to their length. Finally, a transformation step is executed, where the intervals of the driver breaks are determined for every block. The insertion of these breaks might require the deletion of assigned trips from the blocks; these deleted trips become the input of a new VSP, and the process is iterated until all blocks satisfy the above mentioned driver rules, and there are no deleted trips.

To evaluate the efficiency of this approach, we also introduce a method that provides a lower bound on the working time of the drivers and driver activities for a given timetable of trips.

3.3.1 The 'driver-friendly' algorithm

There are two main differences between vehicle blocks and driver shifts: while a block has no constraint on its length or structure, a shift has to fulfill certain regulations, such as not exceeding a maximum working time, or providing fixed time intervals for drivers to take a break. The aim of our algorithm is to construct vehicle blocks that comply with the above mentioned rules, which are two of the most important driver shift regulations. This enables drivers to execute the whole vehicle block during their shift, meaning that they will stay on the same vehicle during the day.

The process iterates over several steps: each iteration creates a vehicle schedule first and then transforms its blocks to satisfy the given driver rules. Maximum working time is managed by a cut and join method: the blocks are classified, then different cuts are executed on them based on their class, and finally the parts are joined together to form feasible blocks once again. Driver breaks are inserted into the blocks through a transformation step that creates intervals where they can be assigned. The outline of this algorithm is shown in Figure 3.2, while the details of every step are presented in the following subsections.

Classification, cut, and join

The main idea behind the algorithm is the classification of vehicle blocks based on the EU regulation that maximizes the working time of a driver on a given day. For this, we examine the length of the blocks, and determine the theoretical number of drivers needed to execute them. The definition of these classes is based on the two shift-types that are usually used by transportation companies: the continuous full-time shift that contains short breaks only, and the split shift that consists of two separate duties that are divided by a long break. According to the above aspects, we introduce the following three classes:

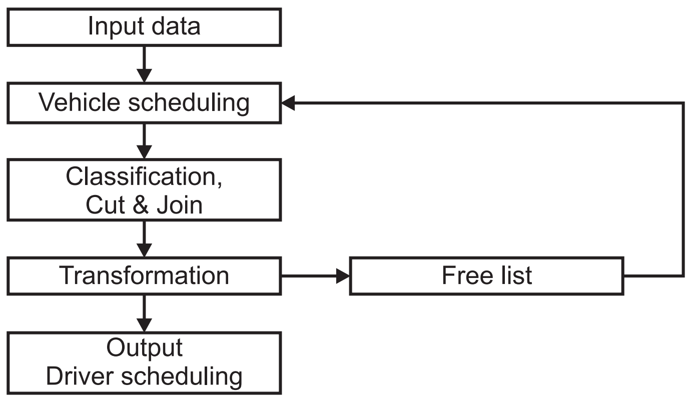


Figure 3.2: Outline of the 'driver-friendly' algorithm

$$C_1 = \{b \in B | l(b) \leq M\},$$

$$C_2 = \{b \in B | M < l(b) \leq 1.5M\},$$

$$C_3 = \{b \in B | 1.5M < l(b)\},$$

where b is a block, B is the set of blocks, M is the maximum working time of a driver shift, and $l(b)$ gives the length of block b.

Class C_1 contains blocks that can be executed by a single driver. Blocks in class C_2 require two drivers (one full-time and one part-time), so these have to be divided into two parts. Dividing points are alternated between 1/3 and 2/3 of the length of the blocks. The resulting larger half will correspond to a full-time driver shift, while the smaller half is only considered as a part-time shift. The alternation of the dividing points is needed so that the part-time shifts could later be joined together to form so-called split shifts. These part-time shifts will be referred to as *work-pieces* in the future. Class C_3 contains blocks that also require two drivers, but as opposed to schedules in class C_2, these drivers both have to work full-time.

The above division of the blocks is not a trivial task, as it may require the introduction of special driver activities to the shifts. Examples for these are getting off and on the vehicles, which usually count as a separate driver tasks. The algorithm also has to consider this when dividing the blocks into work-pieces with different length.

As it was mentioned before, there are also regulations regarding the minimum resting time and breaks of a shift. The algorithm will consider the following rules:

1. If the shift is composed of two work-pieces, there must be enough time between them for a long break.

2. The total length of the resting times and the long break must reach the minimum daily resting time.

Further rules can also be introduced if it is required by the country or the transportation company. Based on the above regulations, a graph can be built using the work-pieces as nodes. Two work-pieces are connected by and edge if they can be part of the same driver shift. Performing a maximal matching on this graph results in work-piece pairs that can be joined together as a split shift, and work-pieces that are not part of this matching can be treated as single, full-time shifts, or broken down into trips, and considered for rescheduling in the next iteration.

Transformation

While the previous step ensures the proper length of the shifts and also places long breaks into split shifts, it does not consider other daily resting activities of the drivers. The aim of this step is to modify the shifts in such a way that short breaks can also be inserted into them. The algorithm will apply the following regulations in this step:

1. Short breaks have a minimum and maximum length.
2. Short breaks can only be executed in given time-windows.
3. Depending on the length of the shift, there can be multiple time-windows where it is mandatory to execute a short break.
4. Short breaks can only be carried out at given geographical locations.

If there are free time intervals in a shift that satisfy all the above rules, then short breaks can be inserted into them. Otherwise, trips have to be removed from the shift to create suitable intervals for these breaks.

The algorithm examines the trips in the time-windows corresponding to the breaks, and determines the best sequence of trips that can be removed to produce the required time interval. Removed trips are either inserted into other shifts, or moved to a so-called 'free-list'. An example for these transformation step can be seen in Figure 3.3, where two trips are removed from a shift to make room for a short break; the first trip is inserted into another shift, while the second trip is moved to the 'free-list'.

These trips are chosen based on the following properties:

- Trips that can be inserted into other shifts without any conflicts are prioritized.
- Otherwise, if trips have to be moved to the 'free-list', then those are chosen that have a minimal overlap in time with trips already on the list.

After these intervals have been created, short breaks are inserted into them with the appropriate deadhead trips to and from the locations where these breaks can be carried out. If all shifts contain the required amount of short breaks after the transformation step, and there are no trips on the

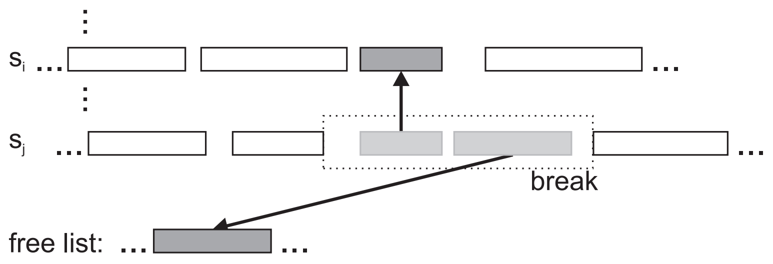


Figure 3.3: Example for a transformation step of the 'driver-friendly' algorithm

'free-list', then the current state of the schedule will provide the solution for the problem. Otherwise, the algorithm moves on to the next step.

Rearranging the 'free-list'

The 'free-list' is basically a set of trips that have to be rescheduled after the transformation step was executed for all shifts. This is done by starting a new iteration with the solution of the VSP, but this time, the input only consists of the trips that are on the 'free-list', the shifts modified in the previous steps remain unchanged.

It can be seen that the number of trips on this list is strictly decreasing with each iteration: some of the trips are always fixed as part of a driver shift, and only conflicting trips will be carried over to the next iteration, which were deleted from these shifts to create an interval for a short break, or part of a work-piece that was broken into trips. Because of this, the algorithm will terminate in finite steps (our test results show that it only needs one extra iteration step).

As mentioned before, the algorithm will terminate if there are no trips on the 'free-list', and the resulting shifts will be the solution of our problem; they form the 'driver-friendly' vehicle schedule together.

3.3.2 Lower bounds for evaluation

To evaluate the quality of our 'driver-friendly' algorithm, we will examine its results for real-life instances and compare them to the actual solutions of the transportation company, measuring the total working time of all the drivers who are needed to execute the resulting shifts. To make this comparison more meaningful, we also developed a method that gives a theoretical lower bound on this working time.

In this subsection, we present our method for generating this lower bound, which is done in two phases: the first phase determines the minimal working time connected to the number of drivers, while the second phase gives a bound on the working time of all activities that are included in a driver shift.

Lower bound for the number of vehicles and drivers

As it was shown by the different classes of blocks in the previous section, the execution of a given daily vehicle schedule requires at least one driver for every block. Because of this, the minimum number of vehicles needed for a given day also gives a lower bound on the minimum number of drivers. This value can be determined by examining the daily timetable. This is done by using an approach similar to [11].

Let T be the set of trips, every $t \in T$ represented by its $dt(t)$ departure and $at(t)$ arrival time, $sl(t)$ starting and $el(t)$ ending location, and $l(t)$ driving time. The day will be divided into smaller (several-minute long) subintervals, and each of these has to be examined independently. First, we determine the minimum number of vehicles needed to execute trips in every subinterval, and then a theoretical lower bound on the working time is calculated based on this and additional driver rules regarding driving time. The outline of this process is described in Algorithm 4.

Let l_{min} denote the length of the shortest trip in T, and let dt_{min} be the earliest departure, and at_{max} be the latest arrival time. Consider the following subintervals: $I_1 = [dt_{min}, dt_{min} + l_I), I_2 = [dt_{min} + 2l_I, dt_{min} + 2l_I), \ldots, I_n = [dt_{min} + (n-1)l_I, dt_{min} + nl_I)$, where l_I is a fixed width such that $l_I \leq l_{min}$. The value of n is chosen so that the last subinterval contains at_{max}.

For each subinterval $I_k (k = 1, \ldots, n)$, consider every trip t for which either $dt(t)$ or $at(t)$ is inside I_k. There are four possible types of trips based on their relation:

T1: only $dt(t)$ falls inside I_k;

T2: only $at(t)$ falls inside I_k;

T3: both $dt(t)$ and $at(t)$ fall outside I_k, and the driving period of t does not intersect I_k ($dt(t) \geq d_{min} + kl_I$ or $at(t) < d_{min} + (k-1)l_I$);

T4: both $dt(t)$ and $at(t)$ fall outside I_k, and the driving period of the trip contains the entire subinterval I_k ($dt(t) \leq d_{min} + (k-1)l_I$ and $at(t) \geq d_{min} + kl_I$).

It is important to note that there is no option where both $dt(t)$ and $at(t)$ fall inside I_k. This is prevented by the value $l_I (\leq l_{min})$ chosen as the width of the subintervals.

As presented in Step 6 of Algorithm 4, a bipartite graph G_k is created for each subinterval I_k. The nodes of this graph are the $dt(t)$ departure and $at(t)$ arrival times that fall inside I_k. The edges of the graph connect arrival and departure nodes: edge $(at(t'), dt(t''))$ exists, if the corresponding trips t' and t'' are compatible (not considering depots, only the deadhead time needed for a vehicle to travel from the ending location of t' to the starting location of t'').

A maximum matching is calculated for every $G_k (k = 1, \ldots, n)$ (see Step 7 of Algorithm 4), and the minimum number of vehicles needed to execute the trips of a subinterval I_k is given by the sum of three terms:

- the number of edges in the maximum matching of G_k,

Algorithm 4 MinimumBusNumbers/Intervals

```
Input:
    - set $G$ of geographical locations (the starting and ending locations of each timetabled trip $t \in T$, $sl(t) \in G$ and $el(t) \in G$),
    - set $T$ of timetabled trips of the given day, $dt(t)$, $at(t)$, $sl(t)$, $el(t)$, and $l(t)$ for every $t \in T$,
    - for each pair $f, h \in G$, the driving time of the deadhead trip between $f$ and $h$, $l(f,h)$.
Initialization phase
    1. $dt_{min}$ = earliest departure time in the timetable ($\min_{t \in T} dt(t)$)
    2. $l_{min}$ driving time of the shortest trip of the day ($min_{t \in T} at(t) - dt(t)$)
    3. A width $l_I$ of the subintervals, such that $l_I \leq l_{min}$
    4. The day is divided into subintervals $I_1, \ldots, I_n$ with length $l_I$.
    5. For each subinterval $I_k = [dt_{min} + (n-1)l_I, dt_{min} + nl_I)$, determine the $c_k$ number of trips that cover the entire interval (all $t \in T$, for which $dt(t) \leq dt_{min} + (k-1)l_I$, and $at(t) \geq dt_{min} + kl_I$)
    6. For each subinterval $I_k$, create a bipartite graph $G_k$. The nodes of the graph are those $dt(t)$ departure and $at(t)$ arrival times that are inside $I_k$. Nodes $at(t')$ and an $dt(t'')$ are connected, if trips $t'$ and $t''$ are compatible.
Determining the minimum number of buses for each subinterval
    7. Solve the maximum matching problem of each $G_k$ bipartite graph. Let $m_k$ be the given solution (the number of edges in the maximum matching), and $v_k$ be the number of nodes that are not covered by the edges of the maximum matching of $G_k$)
    8. $b_k = c_k + m_k + v_k$, $k = 1, \ldots, l$, the minimum number of buses used in $I_k$ (Output)
```

- the number of nodes not present in this maximum matching,
- and the number of trips of type **T4** presented above.

the number

Let the above sum be b_k. An example for this process is presented in Figure 3.4, where the 8 trips in relation with a subinterval I_k result in $b_k = 5$ vehicles: one vehicle given by a trip of type **T4**, three vehicles provided by three pairs of matched trips, and one vehicle given by a trip that is not in the maximum matching.

A lower bound on the number of drivers can be calculated using all b_k values, and considering the rule of maximum daily driving time. For this, let $l_{maxwork}$ denote the maximum length of a driver shift (meaning that the difference of the ending and starting time of any driver shift cannot

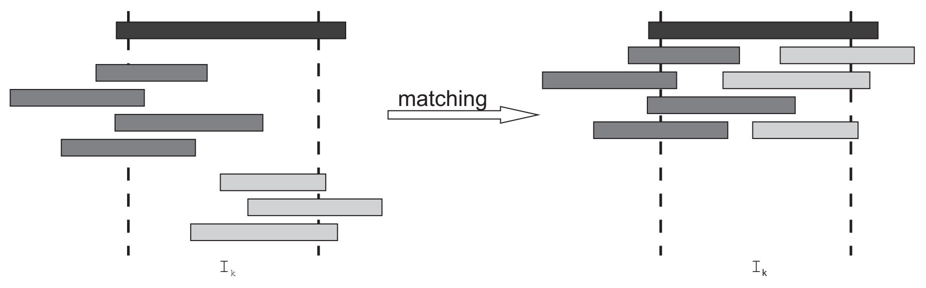


Figure 3.4: Example for a maximum matching of trips in a subinterval I_k

be greater than $l_{maxwork}$). We have to determine the $b_e + b_m$ sums for all pairs of subintervals I_e and I_m for which the difference between the last minute of I_e and the first minute of I_m is greater than $l_{maxwork}$. Because of this property, there can be no overlap between the driver shifts of the vehicles in I_e and I_m. The maximum of these $b_e + b_m$ sums will provide the minimum number of drivers for the given day.

While the minimum number of drivers is important, it can only be used to give a lower bound on the working time for activities that are mandatory for every driver who executes a shift. Examples for such activities are signing on for duty, or administration before and after the shift. For a lower bound on the total working time of different activities inside a driver shift, we present a method in Section 3.3.2.

Lower bound for total daily working time

As it was mentioned in the previous section, the number of drivers can only be used to calculate the working time of the mandatory activities that have to be performed by every driver in service. However, activities inside the shifts provide the major part of the working time: the driving time of the timetabled and deadhead trips, and the different additional activities connected to executing these trips, or waiting in the vehicle between two trips.

To give a lower bound on the working time of such activities, we build a connection-based network similar to the one presented in Section 2.2.3. However, this network uses only a single depot, regardless of the characteristics of the input set T of trips. The network has a separate node for every $t \in T$ trip, as well as a depot starting d_0 and depot ending d_1 node, resulting in the node set

$$N = \{(t|\forall t \in T) \cup d_0 \cup d_1\}.$$

Similarly to the connection based network, there are depot starting and depot ending edges for every trip, given by

$$E_d = \{(d_0, t), (t, d_1)|\forall t \in T\}.$$

The cost c_e of edges $e \in E_d$ will correspond to the working time needed to perform the pull-in or pull-out activities represented by the edge: this comes from the travel time between the geographical locations (if there are multiple depots in the original problem, we consider the minimum of the possible travel times), and any extra time connected to starting or finishing trip t.

Other edges are given by the compatibility between pairs of (t', t'') trips: t' and t'' are connected by an edge only if they compatible regarding both travel distances and depots (meaning that they would be connected in the multi-depot case of the problem as well). This results in the edge set

$$E_t = \{(t', t'') | t' \text{ and } t'' \text{ are compatible}, \forall t', t'' \in T\}.$$

The cost c_e of edges $e \in E_t$ will correspond to the working time needed to travel between trips t' and t'', as well as any additional activities needed after the end of t', before the beginning of t'', and because of waiting between the two trips. The total working time of the timetabled trips in T is constant for the problem, so it will not be included in this model, but added to the solution value afterwards.

We call the resulting $G = (N, E_d \cup E_t)$ network a 'quasi-multi depot' connection-based network model, as the connections are portrayed taking depot-compatibilities into account, but the network has only a single commodity. The capacity of this single depot is the sum of all depot capacities of the original problem. A connection based mathematical model is built for this 'quasi-multi depot' network, and its optimal solution gives a lower bound on the total working time of activities connected to driver shifts.

Using this result, the lower bound on the total working time for a daily schedule is determined by the sum of three terms: the estimated working time based on the number of drivers, the working time given by the 'quasi-multi depot' model, and the working time of the timetabled trips in the input.

3.3.3 Test results

Similarly to Section 3.2.5, we will present the efficiency of the 'driver-friendly' algorithm on real-life input. We use the same instances that were introduced in Table 3.2. To show that the methods presented in Section 3.3 and Section 3.2 can be part of the same sequential solution process in a decision support system, the 'driver-friendly' method will use the variable fixing based on depot costs (Section 3.2.1) to solve the VSP. The results of these instances are presented in Table 3.8.

The above table gives four important details of the problem: the number of buses, number of drivers, running time of the entire process, and the gap from the estimated lower bound. Three different values are given for the number of buses: the result given by the 'driver-friendly' algorithm (column *DF*), the value of the lower bound produced by Algorithm 4 (column *Bound*), and the number of buses used by the company for the same input. The same three values are also presented for the number of drivers. The solution time of the algorithm is given in seconds, and the gap in

Table 3.8: Results of the 'driver-friendly' algorithm

	Number of buses			Number of drivers			DF	Gap (%)	
Instance	DF	Bound	Company	DF	Bound	Company	Time (s)	DF	Company
szeged1	104	95	107	162	132	165	123	8.07	11.77
szeged2	63	42	52	111	78	106	39	9.21	11.62
szeged3	104	95	109	166	132	165	101	8.86	11.67
szeged4	80	55	66	127	88	120	51	9.33	11.01
szeged5	79	54	67	127	89	119	48	9.45	10.36
szeged6	106	95	109	163	132	165	114	8.03	11.62
szeged7	103	94	107	162	130	163	98	9.01	11.83
szeged8	109	95	111	168	132	165	143	8.15	11.65
szeged9	105	95	110	167	132	165	96	8.52	11.96
szeged10	109	95	110	167	131	165	256	8.19	11.49
szeged11	57	41	52	102	72	110	27	9.37	13.64

working time from the theoretical lower bound is shown for both the 'driver-friendly' algorithm, and the original solution of the company. As the result of the 'driver-friendly' algorithm does not provide feasible driver shifts, it had to be modified by a simple process: the shits were examined sequentially, and the missing driver activities were included where they were necessary. This process considered the activities that were required by the driver rules of the transportation company (naturally, the same activities were also present in the original shifts of the company).

As it can be seen from Table 3.8, the solutions of the company were slightly 10% above the given theoretical lower bound. As we do not know the tightness of this bound, their results might even be closer to the optimum than this. In comparison, our proposed algorithm gave a $\sim 3\%$ improvement on average for the instances, which we believe to be quite significant, especially when we also take into consideration that these results can be achieved in a couple of minutes.

The 'driver-friendly' algorithm was developed with the aim of creating vehicle schedules that are not only good from a theoretical point of view, but also have a good structure regarding the driver rules of a transportation company. We managed to introduce a flexible solution process for this problem that gives good quality solutions with a short running time.

3.4 Summary and remarks

In this Chapter, we introduced the concept of application-oriented vehicle scheduling, which aims to create vehicle schedules that are not only good from a theoretical point of view, but can also be applied in a real-life decision support system because of their structure and quick solution time.

In Section 3.2, we presented several different heuristic approaches for solving the MDVSP. These methods were developed to provide good quality solutions with a short running time, which is achieved by decreasing the problem size through the idea of variable fixing. The heuristics try to reduce the size of the mathematical model by finding series of trips (called 'stable chains') that are likely to belong to the same sequence in the final solution, and fix them together as single trips. We showed three different approaches to find such sequences: the first one is based on a cost function, while the second considers the flexibility of the chosen trips through their depot-compatibility. The third approach utilizes a real-life characteristic of the input: it creates chains using only trips that belong to the same bus-line. All three methods result in good quality solutions with short running times. Extensive testing on real-life and randomly generated instances revealed the following property: while general methods (eg. using a cost function) perform similarly for any instance type, exploiting the structure of real-life instances gives even better solutions in practice at the price of performing badly for any instance set without this structure. While generally good methods might be more appealing from a theoretical point of view, the approach based on bus-lines results in solutions with a far better quality and structure when considering a practical application.

In Section 3.3, we introduced an algorithm that creates vehicle schedules with a structure similar to driver shifts. This is done with the help of an iterative process, where a VSP is solved for the input trips of the problem, and then the resulting blocks are modified and transformed through different steps. First, blocks are modified so that they satisfy the rules regarding maximum driver shift length. Then, they are further transformed so that driver breaks can also be inserted into them. If any trips are still not fixed after this step, a new iteration starts with the same steps. To present the quality of the resulting schedules, we also developed a method that gives a lower bound on the total working time of a schedule based on a timetable of trips. Solutions on real-life instances show that, based on the gap from this lower bound, a significant improvement can be achieved compared to the original schedules of the transportation company. The process is flexible enough so that it can be used in the case of different driver regulations, and it can also apply any kind of VSP solution method. Moreover, we achieved good quality solutions in several minutes for these instances.

Chapter 4

Integrated vehicle scheduling and assignment

In this chapter, we introduce the integrated vehicle scheduling and assignment problem, which aims to give feasible vehicle schedules that also include tasks specific to the requirements of the executing vehicle. Vehicle schedules of a transportation company are not only virtual sets of tasks that have to be executed in the given sequence, but they also have a real vehicle (or at least a vehicle brand/type) assigned to the schedule as well. Knowing the vehicle responsible for the execution of the trips also means that there are special needs that have to be taken into account while the vehicle is in service: for example, it can run out of fuel (and has to be refueled), or spends too much time in service (and has to be sent to short maintenance during the day). Constraints such as these are not widely studied. We will refer to these as vehicle-specific activities.

Approaches that consider vehicle-specific characteristics in the VSP started appearing recently with the rise of electric and alternative-fuel vehicles, as these are both cheap and environmentally friendly to operate. However, their major drawback is that they can only run a limited distance, so refueling events are a crucial part of such schedules [71]. While refueling is an important constraint to include in a VSP model, other vehicle-specific activities should also be considered, such as parking, maintenance [57, 23], etc. Constraints like these get significantly less attention than refueling.

We present a set partitioning model for the integrated vehicle scheduling and assignment problem with vehicle-specific activities. If such activities are considered for a vehicle schedule, they also determine certain extra tasks that the vehicle has to execute besides its timetable trips. This results in a vehicle assignment combined with scheduling. Our goal is to give a general framework that can integrate most vehicle-specific activities, and also to provide a flexible model where many of these application-oriented constraints can be included easily. While there may exist other models and methods dealing with these problems separately, such activities have not been considered together in the same problem before to our knowledge. We give a column generation-based solution method for this model, and show its efficiency on randomly generated test instances. To showcase the model, refueling is considered for these instances as the vehicle-specific activity, and the concept of multiple fuel-types is also studied, which is also rarely studied property. We introduced the integrated vehicle scheduling and assignment problem in [16].

4.1 Vehicle scheduling with vehicle-specific activities

Solutions based on the basic concepts of the SDVSP and MDVSP cannot be applied directly in practice, as they only deal with covering all timetabled trips. In real-life, however, vehicles have to execute several types of activities during the day. These come from different vehicle-specific needs (parking, refueling, maintenance, etc.), and usually have to be executed after a vehicle has covered a certain distance (refueling, maintenance), or spent a certain amount of time without performing any activities (parking). In all of these cases, the total length of some consecutive activities is limited, and vehicle-specific events have to be scheduled after such work-pieces.

In general, extensions of the VSP that have either a time or distance constraint on the length of the vehicle blocks belong to the group of Vehicle Scheduling Problems with Time/Route Constraints [48, 25]. These alone cannot satisfy the constraints for vehicle-specific activities, as they limit the total length of the flow representing a block, while activities only need a limit only certain parts of this flow. However, most problems considering the above vehicle-specific activities are special cases of this group.

In this chapter, we introduce a set partitioning-based model for the integrated vehicle scheduling problem with vehicle-specific activities. The aim of this model is to provide a general framework that is capable of producing vehicle schedules that can be used in practice. Most papers dealing with the VSP do not consider vehicle-specific activities, although they are really important real-life constraints for vehicles. The most studied such activity that has an increasing popularity in recent years is the scheduling of alternative fuel (natural gas, hybrid) or electric vehicles.

Vehicle scheduling for alternative fuel vehicles (AF-VSP) is hard because of the limited distance they can cover. Vehicles have to be refueled during their daily blocks, and there are usually very few special refueling stations, with a limited number of refueling pumps. Because of this, location problems for refueling stations are also important [66]. Li presented a flow network based model for both alternative fuel and electric vehicles in [70]. Here, a single depot VSP is considered with a single fuel type, and a single refueling station at the depot. Several column generation-based solution techniques are presented on instances with up to 947 trips. Adler considers the problem with multiple depots in [1, 2]. He uses a set partitioning model for alternative fuel vehicle scheduling, and also uses column generation to solve instances with up to 50 trips. Larger instances are solved with the use of heuristic methods.

Electric vehicle scheduling (E-VSP) has also been getting a lot of attention lately. Both the previously mentioned paper by Li [70] and the dissertation of Adler [1] present models and solution methods for the E-VSP as well. Their solution approaches are similar to the ones they use for the AF-VSP. They mainly deal with battery swapping, and are also based on column generation. A time-space network-based model that allows charging of the vehicles is given in Reuer et al. [91]. In [102], van Kooten et al. present multiple models for the E-VSP considering battery charge. The solution is once again obtained using a column generation approach.

Another important vehicle-specific activity is the assignment of small maintenance tasks to vehicles during their daily blocks. According to [57], they can be of three types: daily, preventive, and emergency maintenance. Daily inspections can be built into a vehicle schedule at the beginning or the end of the blocks, while emergency maintenances are only issued as part of a disruption management process when something unexpected happens to a vehicle. However, preventive maintenance has to be executed by vehicles after a set distance or time interval. Such maintenance activities are considered for rolling stock rotations by Borndörfer et al. in [23], while Haghani et al. also give a model for inserting preventive maintenance for existing bus schedules in [57].

As it can be seen from the above, none of the presented papers deal with multiple vehicle-specific activities at once. Moreover, the vehicle-specific activities that they study usually consider a single vehicle characteristic (eg. vehicles will all have the same fuel-type, or need the same type of maintenance). The next sections introduce the multiple depot integrated vehicle scheduling and assignment problem with vehicles specific tasks, and present a mathematical model for it. This model acts as a general framework, where multiple activities with time or distance constraints can be included depending on the requirements of the problem. The model can give feasible solutions with regards to these, also taking capacity constraints into account. While all the above introduced papers solve the vehicle scheduling problems with a single specific vehicle activity, our goal was to provide a general model that can handle multiple different activities simultaneously.

4.2 Problem definition

We define the integrated vehicle scheduling and assignment problem with vehicle-specific activities (VSAP-VS) the following way: let the input of the problem be set T of timetabled trips, set D of depots and set V of vehicles. The concepts of these are are similar to the VSP defined in Section 2.2: every trip t has a $dt(t)$ departure and $at(t)$ arrival time, a $sl(t)$ starting and $el(t)$ ending location, along with the distance that they cover.

Similarly to the VSP, trips have a set of vehicles that can execute them, and this is also done through the concept of *depots*. Vehicles can belong to depots, which are determined by the features of the trips; certain trips might be served only by vehicles satisfying given constraints (eg. is the vehicle wheelchair accessible, does the vehicle have air conditioning, or a given passenger capacity, etc). Such constraints are important, and are centered around the services you want to provide to passengers taking the given trip (eg. people with wheelchair should have no problem getting on any bus along the line of this trip), or regulation connected to the trip (eg. every bus covering a trip that is longer than a given distance must have air conditioning),

The concept of these depots usually comes from the combination of two features: the type of the vehicle (eg. with/without air conditioning, solo, elongated, etc.), and its starting location at the beginning of the day. The VSAP-VS will consider depots as groups of vehicles sharing the same vehicle-type and having the same starting/ending geographical location at the beginning/end of the day. If the problem has at least two depots, every trip is also assigned a depot-compatibility vector that corresponds to the depots that can execute it.

As opposed to the VSP, the VSAP-VS also considers *vehicle characteristics*. These also refer to different attributes of the vehicles, but only ones that do not have an influence on servicing trips. For example, the fuel type of the vehicle can be such a characteristic; vehicles belonging to the same depot can run on different fuels, but can still service the same trips.

Let n different vehicle-specific activities be represented by set R. These activities are connected to vehicle characteristics rather than depots (eg. refueling vehicles with natural gas, preventive maintenance of hybrid vehicles, etc.), and certain activities can only be carried out by given vehicles. Let set $R_j \subseteq R$ denote the possible tasks belonging to activity j. Similarly to timetabled trips, task $r \in R_j$ also has a starting and ending time, and departure and arrival locations (although these two are usually the same, as activities like parking or refueling are stationary). In order to properly model the capacities of the certain activities, suppose that they can only be carried out in fixed, discrete time intervals instead of continuous availability. Each activity has its own different rules and regulations, but these are usually connected to a time-span or distance limit. For example, vehicles cannot travel more than a given distance without carrying out a refueling task, or they have to start a parking task if they would remain idle for more than a set amount of time. Compatibility between trips and activities also has to be defined. Considering a pair of tasks $a, a' \in T \cup R$, (a, a') are compatible if

- the same vehicle $v \in V$ is able to service both of them (the depot of v is compatible with any trips in $\{a, a'\}$),
- they both satisfy the vehicle characteristics of v (any vehicle-specific task in $\{a, a'\}$ can be carried out by v, eg. we cannot assign battery recharging to a vehicle running on gas), and
- vehicle v can fully service a' after finishing a with respect to the running time and distance between the arrival location of a and the departure location of a' (also considering the possible deadhead trip between a and a', if needed).

The aim of the problem is to give a feasible vehicle schedule that includes both timetabled trips and vehicle-specific activities, and tasks in any of the vehicle blocks are pairwise compatible. The cost of such a vehicle schedule is the linear combination of three different terms: a one-time daily cost for each vehicle covering a block, a distance proportional cost for covering timetabled trips and deadheads, and costs of the vehicle-specific activities included in the blocks.

4.3 Mathematical model

In this section, we present our model for creating vehicles schedules that also take vehicle-specific activities into account.

4.3.1 A set partitioning mathematical model

For each depot $d \in D$, let B_d be the set of feasible b blocks that start from $dt(d)$ and also return there. A block is a sequence of compatible trips and activities that can be executed by the same vehicle, and satisfies all activity rules connected to this vehicle. Let

$$B = \bigcup_{d \in D} B_d$$

be the set of all such blocks. For each $b \in B_d$, let y_b^d be the following binary variable

$$y_b^d = \begin{cases} 1, & \text{if } b \in B_d \text{ block is part of the solution} \\ 0, & \text{otherwise} \end{cases}$$

Furthermore, let $a_{e,b}^d$ be the following

$$a_{e,b}^d = \begin{cases} 1, & \text{if } b \in B_d \text{ block contains activity edge } e \\ 0, & \text{otherwise} \end{cases}$$

Let T denote the set of trips for the problem, and R give the set of tasks belonging to all vehicle-specific activities. Tasks that are connected to a single activity of a given vehicle characteristic are given by $R_i \subseteq R (1 \leq i \leq n)$, where n is the number of all such activities. The capacity of a depot $d \in D$ is denoted by k_d, and the maximum number of vehicles that can simultaneously carry out a vehicle-specific task $r \in R$ is given by k_r. Let c_b be the cost associated with block $b \in B_d$. Then we can formalize our model the following way:

$$\text{minimize} \sum_{d \in D} \sum_{b \in B_d} c_b y_b^d, \tag{4.1}$$

s.t.

$$\sum_{d \in D} \sum_{b \in B_d, e=(dt(t),at(t)) \in E_d} a^d_{e,b} y^d_b = 1, \forall t \in T \tag{4.2}$$

$$\sum_{d \in D} \sum_{b \in B_d, e=(dt(r),at(r)) \in E_d} a^d_{e,b} y^d_b \leq k_r, \forall r \in R \tag{4.3}$$

$$\sum_{b \in B_d} y^d_b \leq k_d, \forall d \in D \tag{4.4}$$

$$y^d_b \in \{0,1\}, \forall d \in D, \forall b \in B_d \tag{4.5}$$

$$a^d_{e,b} \in \{0,1\}, \forall d \in D, \forall b \in B_d, \forall e \in E_d \tag{4.6}$$

Constraint (4.2) ensures that every trip is covered exactly once. Blocks simultaneously containing certain vehicle-specific activities are given by constraint (4.3), every task $r \in R$ having a maximum capacity k_r. Note, that this constraint only ensures the vehicle limits on each task, as we suppose that every block is feasible, also meaning that they satisfy the distance and time constraints or any other rules connected to the activities. Managing this feasibility will be addressed in the following subsections. Constraint (4.4) limits the capacities of each depot $d \in D$.

4.3.2 A column generation approach

Due to the extremely high number of possible blocks (resulting in a large amount of decision variables), the model cannot be solved directly by a MIP solver. Moreover, generating all blocks is also problematic, as the number of combinations is too large. Instead of giving all the possible blocks in the model, only the most important ones have to be generated to achieve a good quality solution.

Column generation [42, 76] is a classical method that is usually applied to such problems, relaxing the integer constraints of the variables. This relaxed problem is also called as the master problem. The usual steps taken during the solution process are the following:

1. Create an initial solution. The resulting schedule will provide the starting set of columns.
2. Solve the relaxed problem (master problem) on the actual set of columns, store the lower bound, duals.
3. Solve a pricing problem in order to look for new columns that have a negative reduced cost.
4. Add the new columns to the master problem, and erase any old columns that are obsolete, and have a large cost.
5. Check termination criteria. If none apply, go to step 2.
6. Create the final schedule based on the current columns of the problem.

Creating the final vehicle schedule in step 6. can be done by solving the resulting problem as an IP using a solver. A solver can also be used in step 2. for the solution of the master problem LP. The process can terminate in step 5. if a given iteration count is reached, or the solution has not improved significantly over the past iteration steps. The most important part of the algorithm is the solution of the pricing problem in step 3.

4.3.3 Initial solution

In this subsection, we present our heuristic for creating an initial solution for the column generation process. Its pseudo code can be seen in Algorithm 5.

Algorithm 5 Initial vehicle schedules with activities.

```
Funct buildSchedule($T, V$)
 1: Let the set of blocks be $B := \{\}$
 2: Order T by ascending trip departure times
 3: for ($t \in T$) do
 4:     Let $f := b \in B$ with the cheapest insertion cost for $t$
 5:     if $f = \emptyset$ then
 6:         $f := v \in V$ that can serve $t$ with the smallest cost
 7:         Assign $t$ to $f$
 8:         $B := B \cup \{f\}$
 9:     else
10:         Assign $t$ to $f$
11:     end if
12:     for all activities $R_i \subseteq R$ do
13:         checkAtivity($R_i, f$)
14:     end for
15: end for
16: return $B$
Funct checkActivity($A, b$)
 1: $P$ := all available tasks in A for $b$
 2: $d$ := resource (time/distance) needed for $b$ to serve any trip $t$
 3: $v$ := is an activity rule violated servicing any trip $t$?
 4: if $d \geq remainingResource(b)$ or $v$ = true then
 5:     $p$ := cheapest compatible task from $P$
 6:     Assign $p$ to $b$
 7:     $remainingResource(b) := MAX$
 8: else if v = true then
 9:     $p$ := cheapest compatible task from $P$
10:     Assign $p$ to $b$
11: end if
```

The input of the algorithm is the set T of trips and set V of vehicles. The process iterates over the input trips in ascending order of their departure times, and assigns the current trip to an existing block with the cheapest cost. If there is no block where the trip can be inserted, then a new block is created for the trip. The vehicle chosen for this block is the one with the smallest cost that can execute the trip. After each trip assignment, the current block is checked whether its vehicle has to undergo a task belonging to any of its vehicle-specific activities.

The function $checkActivity(A, b)$ checks block b if it could execute any of the remaining trips without violating the rules of activity A. If any of the rules would be violated, b is sent to carry out the given activity, and is assigned the cheapest compatible task. If the activity was connected to a resource (eg. distance, time), then this is also replenished for the vehicle servicing the block.

Algorithm 6 Function for checking refueling activities.

```
Funct checkFuel($b$)
1: $R$ := all available refueling times for $b$
2: $d := 2 \cdot maxDeadheadTime + maxTripTime$
3: if $d \geq remDist(b)$ then
4:     $r$ := cheapest compatible refueling possibility for $b$
5:     Assign $r$ to $b$
6:     $remTime(b) := MAX$
7: end if
```

As an example, we present a function for managing refueling activities in Algorithm 6. Vehicles are sent for refueling tasks if their remaining distance (denoted as *remDist*) would not allow them to head out for any trip, service it, and then head back to any location. For this, we count the distance of two deadheads as *maxDeadheadTime* (with the maximal possible distance), and the distance of the trip as *maxTripTime* (taking the maximal remaining trip distance into account). If refueling is needed, the vehicle is assigned to the next available possibility with the cheapest cost. After the refueling task is completed, the remaining distance that the vehicle can cover is again set to its maximum value.

When the initial solution is created, its blocks are used as the starting columns of the relaxed master problem.

4.3.4 Pricing problem

After the solution of the master problem, information about its duals is used to create new columns that can improve its current objective. Each such column corresponds to a legal vehicle block. These blocks are created with the use of a generation network.

This network is the basis of the pricing problem, and is used to build vehicle blocks with a negative cost that also satisfy all the above mentioned vehicle-specific activities. This basically means that after a vehicle has consumed enough of a given resource, the appropriate events for its vehicle-specific needs also have to be scheduled. This can be done by solving a resource-constrained shortest path problem.

We use a time-space generation network similar to the one presented by [96], and a separate network is created and solved for every pair of depot and activity type. Each network is given by the possible tasks that can be assigned to the vehicles: timetabled trips, deadhead trips and other vehicle-specific events. The nodes of the network correspond to the arrival and departure time of these tasks on their given time-lines. The edges of the network can either correspond to their respective tasks, or be waiting edges on the time-lines.

Formally, let $G = (N, E)$ represent a general duty generation network. Such a network is created for each combination of depot d and activity a, resulting in networks $H_a^d = (N_a^d, E_a^d)$. The nodes in N_a^d are the following: *depot_source, depot_sink, trip_start, trip_end, activity_start, activity_end.* Edges in E_a^d are connecting these nodes; *trip_start* and *trip_end* nodes belonging to the same trip are connected by *trip edges*, *activity_start* and *activity_end* nodes belonging to the same activity are connected by *activity edges*. Any end node is connected to the start nodes of other tasks; *deadhead edges* connect the ones in different locations, while *waiting edges* run between nodes in the same location. The only exception to this is activities: the end node of an activity task is not connected to the start node of the same activity type, as there is no point of executing two similar vehicle-specific activities after each other. *Block_start edges* connect the *depot_source* node to every *trip_start* node, and every *trip_end* node is connected to the *depot_sink* node with *block_end edges*.

An example of such a network can be seen in Figure 4.1. In this figure, refueling is considered as the vehicle-specific activity of the network, and it contains 20-minute refueling tasks.

To provide feasibility with respect to vehicle-specific events, so-called resources are also associated with each vehicle on the network. A single resource is allocated for each vehicle-specific activity, which calculate the time/distance (depending on the resource) traveled by the vehicles with the help of a resource extension function. These will filter out blocks whose total consumed resources violate any of the vehicle-specific needs. Tasks belonging to the activity replenish the appropriate resource capacity of the executing vehicle.

As suggested in [96], negative reduced cost vehicle blocks are generated on these networks using a dynamic programming approach presented in [44]. Blocks with the lowest negative reduced cost are added to the master problem from every generation network.

4.3.5 Creating the final schedule

After the column generation steps have concluded, the resulting master problem only gives us a solution for the LP-relaxed scheduling problem. To obtain a final integer solution, a second phase is usually executed. This can be done in several ways.

One approach is using a Lagrangian relaxation [59], as in the case of integrated vehicle and crew scheduling problems, where the linking constraints between vehicle and crew are an ideal candidate to be relaxed.

Another method is embedding the column generation process in a branch-and-bound framework

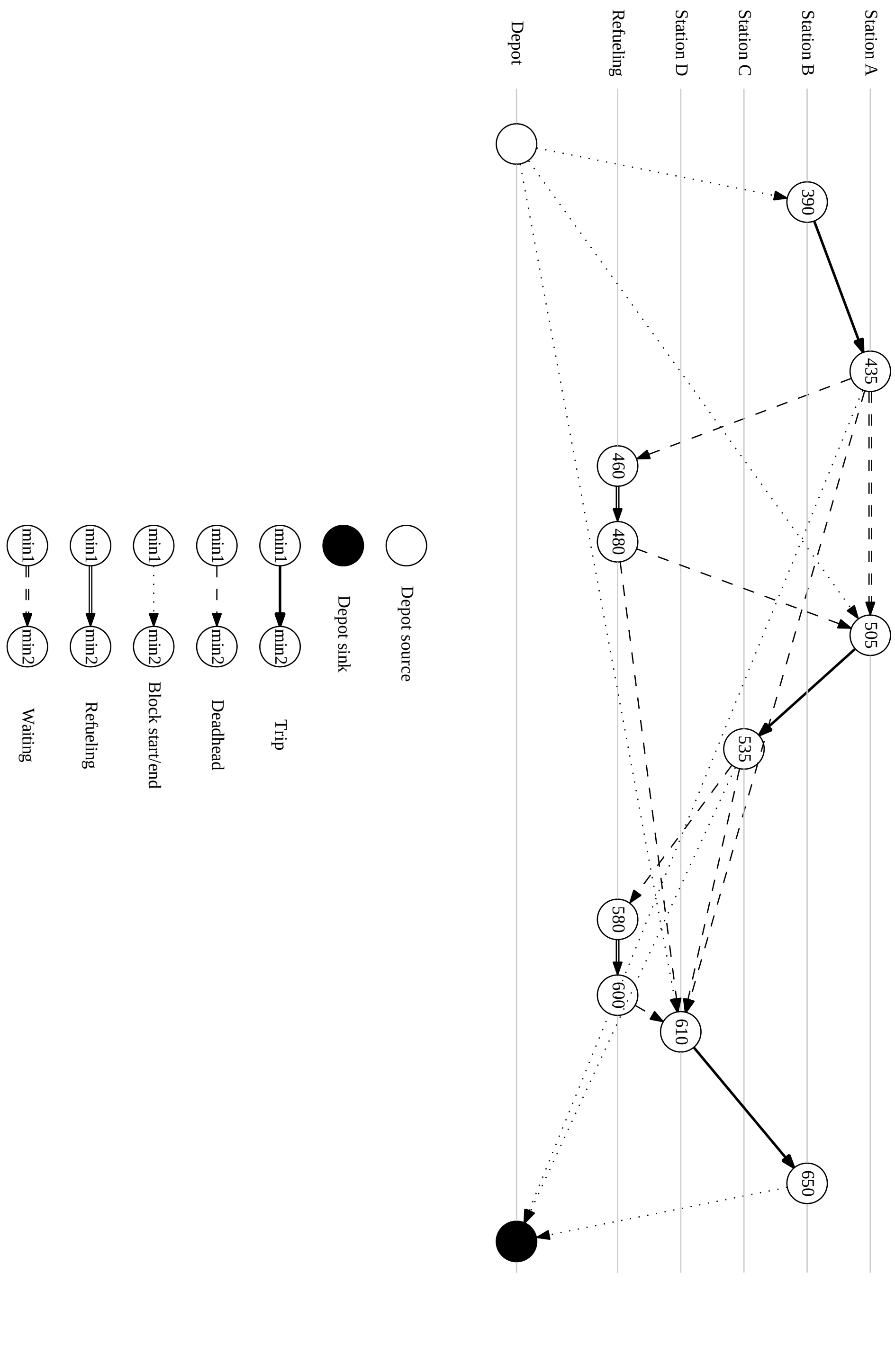


Figure 4.1: Example of a generation network for the VSAP-VS

that searches for the optimal integer solution [20, 78]. This method is also referred to as branch-and-price.

Integer solutions can also be obtained by the use of truncated column generation, which has been used for the MDVSP [85]. After the column generation has concluded, this approach tries to obtain an integer solution of the master problem by performing a rounding/variable fixing heuristic, fixing the values of fractional variables.

Our approach is similar to truncated column generation, but instead of applying a heuristic approach to round the variables of our problem, we simply re-introduce the integrality constraint to our master problem, and solve it using an IP solver. A similar approach is also used in [52]. To control the time of the solution process, we also set a time-limit for the solver. This approach yields good quality solutions for single depot and smaller-sized multi-depot instances, but larger, or hard-to-solve problems will usually yield poor results.

4.4 Test results

We tested our approach on random input generated by the method presented in Chapter 7.4.2. Both single and multiple-depot test cases were created in different sizes: 50, 250, 500, 1000, 1500 and 2000 trips. While the model we presented can be used as a general framework for tackling different vehicle-specific activities, we chose refueling to showcase our test instances, as this is the most widely studied vehicle activity. However, while papers in the literature usually deal with only a single fuel-type for a problem, our test instances have two different fuels. Moreover, we also allow vehicles belonging to the same depot to have different types of fuel.

A total of 24 test case were solved both for the single- and multiple depot problems: we generated 4 instances for every problem size. The ILOG CPLEX solver was used during the solution process, with a limit set on its running time: the limit on the column generation phase was 7.5 hours, while the IP phase ran for 3 hours in the single depot case, and 5 hours in the multi-depot case. This adds up to a total maximum running time of 10.5 hours (37800 seconds) for single depot problems, and 12.5 hours (45000 seconds) for multi-depot problems. These limits have to be included if we consider the practical application of such a solution method, as planners must have an estimate on time when they will have a feasible solution.

The following information is presented about the results: the number of vehicle blocks given by the initial heuristic (initial blocks), the final number of blocks given by the resulting IP (IP blocks), the final optimality gap (%) given by CPLEX, and the solution time of the instance.

Table 4.1 shows results for the single depot test instances. It can be seen that good quality solutions are achieved for all 24 problems, the optimality gap is mostly under 1% (with only a single instance being above 3%, and six instances sitting between 1%-3%). The maximum runtime was only reached by the 1500 and 2000 trip instances. One notable feature of all single depot results is that

Table 4.1: Single depot test instances for the VSAP-VS.

Instance	Trips	Initial blocks	IP blocks	MIP gap (%)	Time (s)
input_01	50	16	16	0.00%	95
input_02		16	10	0.00%	80
input_03		16	11	0.00%	86
input_04		16	16	0.00%	85
input_05	250	58	71	0.01%	258
input_06		56	56	0.00%	335
input_07		58	58	0.01%	329
input_08		56	69	0.00%	257
input_09	500	99	131	0.08%	4938
input_10		107	104	0.43%	5473
input_11		103	107	0.49%	5527
input_12		96	137	0.18%	5014
input_13	1000	184	290	1.51%	11228
input_14		183	231	2.16%	19808
input_15		187	219	2.03%	19361
input_16		185	288	1.54%	11073
input_17	1500	260	611	0.01%	21028
input_18		268	529	3.56%	21629
input_19		264	612	0.01%	18488
input_20		262	595	2.05%	21612
input_21	2000	374	731	2.70%	21643
input_22		364	663	0.76%	21615
input_23		358	383	0.00%	18105
input_24		358	593	0.01%	21007

the IP solution of the problem uses significantly more buses than the initial heuristic. The reason for this phenomenon can be found by examining the different cost factors of the input. Most of the vehicles generated for the input instances ended up having a relatively low daily cost, and a more significant distance proportional cost factor. Because of this, the IP solution aimed to minimize the distance traveled by its vehicles (which means trying to schedule as few deadhead trips as possible). The initial heuristic is essentially a greedy assignment of trips to vehicles, which does not take traveled distance into account, but only introduces new vehicles to the schedules if it really has to.

Overall, the solutions we achieved for a single depot and two fuel-types are promising, even for larger instances. The maximum problem size presented by the test results for similar problems in other papers in the literature rarely exceed 1000 trips, while we show good quality solutions for several instances with 1500 and 2000 trips as well, while considering vehicles with two different refueling constraints.

Table 4.2 gives the test results for the multi-depot test cases. Here we used vehicles belonging

Table 4.2: Multiple depot test instances for the VSAP-VS.

Instance	Trips	Initial blocks	IP blocks	MIP gap (%)	Time (s)
input_1	50	26	9	0.00%	10
input_2		25	8	0.00%	16
input_3		10	9	0.01%	26
input_4		9	9	0.00%	13
input_5	250	51	32	8.06%	19028
input_6		46	34	6.42%	19301
input_7		47	35	7.30%	19253
input_8		45	35	6.49%	18998
input_9	500	84	64	11.43%	22801
input_10		100	63	10.84%	23184
input_11		81	57	12.73%	24836
input_12		86	58	12.48%	24452
input_13	1000	181	135	15.29%	45067
input_14		164	122	23.59%	45045
input_15		177	124	21.65%	45087
input_16		182	124	23.48%	45064
input_17	1500	281	216	25.11%	45081
input_18		270	217	26.64%	45033
input_19		291	221	27.02%	45024
input_20		284	222	27.59%	45112
input_21	2000	366	320	26.35%	45113
input_22		376	329	26.10%	45160
input_23		378	336	26.19%	45119
input_24		335	313	27.60%	45061

to two different depots, and the problem also considered two fuel-types (both depots had a mix of vehicles with both fuel-types). This results in a more complicated problem (multiple depots, two fuel-types, instances with a large number of trips) that is usually considered in the literature.

The results for small, 50-trip instances are promising, as we also found near optimal solutions in a short time. However, these were the only cases, where both the column generation phase and the IP phase concluded well before its time limit. Even for the 250 trip instances, the IP solution process was aborted prematurely as they ran out of time. For the 250 and 500 trip instances, the column generation process concluded with no more columns to generate, but it also ran out of time for the larger input.

The effect of reaching the time limit can clearly be seen on the results. While the average gap given by the solver was 9.47% for the 250- and 500-trip instances, where the column generation finished without any problems, the three remaining instance sets (with 1000, 1500 and 2000 trips), where both the column generation and IP solution phases were aborted because reaching the time limit, had an average gap of 24.72%. While these results might not seem particularly promising, the quality of the achieved solution only depends on the time allocated for the two phases. Based on the single depot, and small multi-depot results, the model will provide good quality solutions, but the time limit has to be increased significantly. However, this would invalidate the very reason that we introduced the time limit in the first place: to make the solution process usable in practice, where results have to be achieved in a foreseeable time.

4.5 Summary and remarks

In this chapter, we presented a general framework for the integrated vehicle scheduling and assignment that also considers tasks for vehicle-specific activities. This framework gives a daily vehicle schedule that also includes the special needs of the vehicles executing it; activities such as refueling, parking, etc. are considered in the resulting vehicle blocks. We gave a set partitioning-based mathematical model for the problem, where most vehicle-specific activities can easily be integrated based on the desired constraints. This model is then solved using a column generation approach. We presented the efficiency of the proposed model on randomly generated test instances, using refueling to showcase vehicle-specific activities.

Instances with one and two depots were both created, and all of them had two fuel-types with different constraints (also allowing that the same depot can contain vehicles with different fuel-types). To guarantee that a feasible result is achieved within a foreseeable period, a time limit was set for both phases of the solution process. Test runs for the single depot cases resulted in good quality solutions, as did the smaller-sized multi-depot ones. Results for the larger multi-depot instances had a bigger optimality gap, but this is due to one or both of the phases terminating because of the time limit. Based on the results of the other test cases, these gaps would also decrease significantly given more time for the solution.

While the results are promising, there is always room for improvement of the process. Implementing a proper branch-and-price framework could result in better solution. Because of the limited running time, a truncated column generation approach with a rounding heuristic as the second phase should also be examined. Another possibility to decrease solution time could be the parallelization of the column generation process. Implementing these approaches and comparing their results should be a next step of this research.

The model was created as a general framework that can handle multiple vehicle activities. Although we presented problems with two different fuel-types as our test cases, we did not generate instances with two (or more) completely different activity types. Our future research will also include experimenting with different vehicle constraints.

Chapter 5

Managing disruptions with vehicle rescheduling

The methods discussed so far in Chapter 3 and Chapter 4 can be used to construct vehicle schedules for a single day. However, public transportation companies create their schedules in advance for a planning period, which is usually a several-week-long horizon. Such a planning period is a series of interconnected parts; the smallest such parts are generally the single daily schedules. We will refer to these as the planning units of the period. Each planning unit has a pre-planned schedule that has to be executed by the vehicles of the company.

While such a schedule is useful from a planning perspective, the series of tasks carried out in practice by the end of a planning unit are generally different from the pre-planned ones. The main reasons for this are unforeseen events that happen during the execution of a planning unit. These events are called disruptions by the literature, and the field that deals with them is referred to as disruption management. Clausen et al. [28] define a disruption as "an event or a series of events that renders the planned schedules for aircraft, crew, etc. infeasible". Disruptions can occur for various reasons, such as lateness, vehicle breakdown, or the introduction of completely new tasks that have to be serviced. Disruptions have to be addressed as soon as possible to restore the order of transportation. If such and event happens, the company has to create a new feasible schedule, where all available tasks are carried out in a feasible manner once again. The vehicle rescheduling problem (VRSP) was defined by Li et al. [73] for bus transit, and it deals with the scenario of restoring the order of transportation after a single disruption.

Although the philosophy behind the VRSP (addressing an unknown disruption) suggests a dynamic problem, solving the scenario after the disruption has already happened is a static one: all parts of the input data (the pre-planned schedule and the disruption) are known in advance, and the problem can be solved to optimality given enough time. From an operations management point of view, focusing on solving a single disruption is not enough. In real-life scenarios, there are several different disruptions happening over a planning unit, and the effectiveness of disruption management can be measured by the quality of the final solution after the planning unit has ended. However, as the disruptions are not known in advance, we are dealing with an online input for the problem.

In this Chapter, we first review the dynamic problems of transportation in Section 5.1, then present two different aspects of vehicle rescheduling. In Section 5.2, we give a multi-depot network model for the bus rescheduling problem, and also present two fast heuristic methods for solving the problem in practice. These were first introduced in [35], then later extended in [37]. Section 5.3 introduces the concept of dynamic vehicle rescheduling (DVRSP), which aims to evaluate the efficiency of disruption management methods over a planning unit. We proposed this problem in [37].

5.1 Dynamic problems of vehicles

There are multiple research fields dealing with the dynamic problems connected to vehicles and transportation . The majority of this research considers a variation of the vehicle routing problem (VRP). The dynamic vehicle routing problem (DVRP) [88, 89] deals with pre-planned vehicle routes and new incoming requests during their execution. This can result in the alteration or the complete redefinition of the routing plan for the vehicles. Pillac et al. give a review of this field in [87].

The dynamic vehicle scheduling problem (DVSP), a dynamic variation for the VSP was introduced by Huisman et al. [61, 62]. In these papers, they don't solve an entire VSP in advance, but generate it online solving a sequence of optimization problems. Their approach is different from the one that was discussed for vehicle rescheduling previously, because they solve a VSP for an online input of tasks instead of addressing disruptions occurring for a pre-planned schedule.

Disruption management in transportation can also be considered as a dynamic problem. Optimization methods for handling airline disruptions [69, 99] aim to solve disruptions by canceling flights or rerouting aircraft. Models for these problems are specific to this field due to the relatively small number of airplanes and their limited number of connections. Moreover, the methods used for airline disruptions are computationally intensive, and have a long running time for the significantly larger vehicle transportation problems. An overview of this field is given by Clausen et al. [29, 28].

Disruption management in railway transportation is covered in [64]. The structure of these problems is also significantly different from the VRSP, the main difference being the fixed railway network, and the capacity limit of the tracks.

For a long time, the VRSP for bus transportation was only considered in papers by Li et al. In [73], they propose a quasi-assignment model and an auction algorithm for the problem, while they introduce a network flow model for the VRSP in [74], which they solve using a Lagrangian method. Their methods are also considered as part of a decision support system [72]. However, they only deal with the single depot case of the problem (where all vehicles are uniform and share same starting location), which does not apply to all real-life problems. Since the publication of our research, two other papers have been published dealing with the VRSP. Uçar et al. [101] present a robust model for the multi-depot vehicle rescheduling problem that considers multiple disruption types. They solve this model using a simultaneous column- and row-generation algorithm. Guedes et al. [54] introduce the multi-depot vehicle type rescheduling problem, and give a mathematical model for it that can also handle simultaneous disruptions. They achieve fast solutions using a three-phase heuristic framework.

The above papers dealing with the VRSP aim to solve a single disruption scenario efficiently. However, in practice several disruptions occur during a planning unit. These disruptions happen independently of each other, and a disruption has no information about other future disruptions. As far as we know, there has been no research of the overall effect of disruption management methods on the final resulting schedule.

5.2 Recovery from disruptions

As it was mentioned earlier, companies create their vehicle schedules in advance for a longer planning horizon. Each planning unit of this period has a pre-planned vehicle schedule. Such a schedule consists of vehicle blocks, each such block assigned to a unique vehicle. When a disruption happens at a given time s, the pre-planned schedule of the given unit may become infeasible, and as a result, one or more trips can no longer be executed by their original vehicles. There are two major types of disruptions according to their effect:

- **Immediate effect**: This type of disruption affects the schedule of the current day, and has to be addressed as soon as possible. It can be caused by several different reasons:
  - *Shortage of vehicle*: This usually happens when a vehicle is unavailable for a shorter period of time (eg. due to lateness, or a breakdown), or there are newly introduced trips that are not part of the original daily schedule (eg. there is a special need for them).
  - *Shortage of crew*: Similarly to the above item, this happens when one or more drivers are unavailable for a shorter period of time.
  - *Changes in the route*: This is a somewhat different problem compared to the disruptions introduced above. Trips have fixed routes that their vehicles have to cover, and these can also be blocked by unforeseen event. In this case, the re-routing of trips also has to be done in addition to managing lateness and any shortage.
- **Long-term effect**: More than one day of the planning period is affected by this disruption. It is usually caused by a longer shortage of vehicle or crew, or an unforeseen roadblock that lasts for a longer period.

In this section, we will only be addressing the short-term disruptions of vehicle schedules: these can happen either because of problems with the vehicle, or newly introduced trips to the daily schedule. First, we introduce the VRSP in Section 5.2.1, then give a mathematical model for its multi-depot case in Section 5.2.2. Section 5.2.3 presents two fast heuristics methods for solving the problem.

5.2.1 The vehicle rescheduling problem

The most important input for the VRSP is the disrupted schedule DS of the planning unit. As a disruption will happen at a time s, DS should only contain vehicles and trips that are still relevant from an operational point of view: trips that were already executed before s and vehicles that are not available after s should not be considered. We also have the set DT of disrupted trips, which contains the trips that cannot be served due to the disruption. The set DT can contain timetabled trips that no longer have their assigned vehicle as a result of the disruption, and it can also contain newly introduced trips that were not part of the original daily schedule. Some vehicles might not be available over a period of time due to the disruption. For this, we also get a list S of 3-tuples (v, s_1, s_2), which correspond to vehicle v not being able to service trips between times s_1 and s_2. Let $T' \subseteq T$ be the subset of trips that start later than s. The aim is to provide a feasible vehicle schedule for the problem that deals with all trips $T' \cup DT$, and minimizes the arising costs.

The costs of the problem depend on the restrictions that are taken into consideration. We will introduce the following cost components:

- **Operating costs**: This cost is proportional to the distance covered by the given vehicle. If a new vehicle is introduced, it also has a fixed daily cost. This cost can also be scaled with a penalty parameter for new vehicles if we want to primarily use our current vehicles in service.
- **Deviation from the original schedule:** If a trip is carried out by a different vehicle in the solution of the VRSP than in the original schedule, we introduce an extra penalty.
- **Lateness of the trips:** It should be allowed for trips to start later than their original starting time. However, as this is not a desired results, each minute of lateness should be penalized.
- **Trip cancellation:** Trips can also be canceled in the final solution, but a penalty should also be introduced for each such trip. This has to be a greater cost than the actual cost of the trip and its possible deadhead trips, and it also has to be higher than the cost introduced for the deviation from the original schedule. Generally, cancellations should be avoided, and this penalty has to be chosen accordingly.

Figure 5.1 models a typical situation that can arise in the daily practice of a transportation company. Each part of the figure (*a), b), c)*) presents a vehicle schedule, each row these schedules corresponding to a vehicle block. The boxes in a row represent trips that have to be executed. The original schedule in part *a)* is disrupted, and the two blue trips have to be rescheduled using the three available vehicle blocks. We give two different solutions: in part *b)*, a task is moved from the first block to the third one before the insertion of the disrupted trips, while part *c)* gives a solution with trivial insertion of the blue trips into different blocks. Note, that other solutions are also possible.

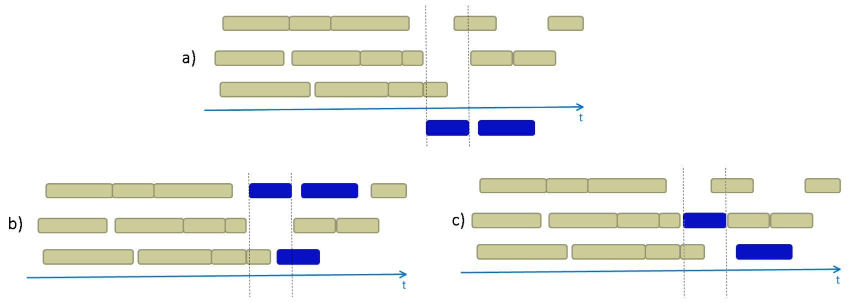


Figure 5.1: A typical example for the VRSP: the two disrupted trips in example a) have to be inserted into the given blocks, b) and c) represent possible solutions

5.2.2 Mathematical models

In this section we give two multi-commodity network flow models for the VRSP. The VRSP has a similar structure to the VSP, and because of this, we are using similar modeling approaches. The connection-based network (CBN) presented in Section 2.2.3 is an important model for the VSP, which represents all possible connections between the trips of the problem. The first mathematical model we introduce for the quasi-static VRSP in Section 5.2.2 will be based on this CBN.

The main drawback of this model is its size: problems with a large number of trips become impossible to solve directly, and this also remains true in case of the VRSP. The size of the model can be decreased by reformulating it as a time-space network (TSN), which was introduced in Section 2.2.3. By using their method presented there, we reformulate the connection based VRSP model to a TSN model in Section 5.2.2.

To show the major differences between the introduced models, we present a small VRSP instance in Section 5.2.2. Based on this example, we construct the network for both models to illustrate the structure of the problems.

The advantage of the CBN is that it represents the complete structure of our problem, where each connection is explicitly modeled. This also gives its major drawback, that a large number of possible connections result in a large problem size. Compared to the CBN, the TSN offers a significant size reduction at the cost of aggregating unique connections.

Connection based network model

Our most important input is the schedule of the company for the given planning unit, denoted by DS. Let D be the set of depots (denoting groups of vehicles with the same type and starting/ending geographical locations), V be the set of vehicles still in service at the disruption time s. Vehicles used in the solution of the problem can either be newly introduced ones starting from one of the depots, or they can also be the ones in service at the time of the disruption. To denote both options as possible sources of vehicles for the problem, we also introduce the set P, where $P = D \cup V$.

Let T' be the set of non-disrupted service trips of the given planning unit that start after s, and let set DT contain any newly introduced disrupted trips. Let the set $T = \{T' \cup DT\}$ represent all the trips of our problem that have to be executed. Every trip $t \in T$ has a departure time $dt(t)$, arrival time $at(t)$, starting location $sl(t)$ and ending location $el(t)$. The set of depots and vehicles that can execute a trip t is denoted by $d(t)$. Let $T_d \subseteq T$ be the set of trips that can be executed from depot d, and $T_v \subseteq T$ the set of trips that can be carried out by vehicle v. We also have to give a set S of tuples (v, s_1, s_2). If a vehicle v cannot service any trips due to technical problems between times s_1 and s_2, then it will have no compatible trips that have $s_1 \leq dt(t), at(t) \leq s_2$.

For every depot $d \in D$, we introduce notations $sl(d)$ and $el(d)$. A depot d is represented by $sl(d)$ when we consider it as the starting location of its vehicles, while we use $el(d)$ when it gives the ending location of its vehicles. Similarly for every vehicle $v \in V$ currently in service we define a starting location $sl(v)$ and ending location $el(v)$. For a vehicle v, $sl(v)$ corresponds to the current geographical location of v at the time of the disruption, and $el(v)$ is the geographical location where it should end the planning unit. The set of nodes of our network will be the following:

$$N = \{dt(t) \cup at(t) \cup sl(d) \cup el(d) \cup sl(v) \cup el(v) | t \in T, d \in D, v \in V\}.$$

Using the nodes above, we define the different edges of the network. Let

$$J^d = \{(dt(t), at(t)) | t \in T_d\}$$

be the set of trips that can be served by vehicles located at depot d, and let

$$J^v = \{(dt(t), at(t)) | t \in T_v\}$$

be the set of trips that can be executed by vehicle v currently in service. Let

$$K^d = \{(at(t), dt(t')) | t, t' \in T_d \textit{ are compatible}\}$$

be the possible deadhead trips of vehicles located at depot d and

$$K^v = \{(at(t), dt(t')) | t, t' \in T_v \textit{ are compatible}\}$$

be the possible deadhead trips of a vehicle v in service.

Let

$$L^d = \{(sl(d), dt(t)), (at(t), el(d)) | t \in T_d\}$$

be all the pull-in and pull-out edges of vehicles located at depot d, and let

$$L^v = \{(sl(d), dt(t)), (at(t), el(d)) | t \in T_v\}$$

be the pull-in and pull-out edges of vehicle v in service. The above sets together with circulation edges for every depot and vehicle give us the set of edges of our network:

$$E = \{J^d \cup J^v \cup K^d \cup K^v \cup L^d \cup L^v \cup \{(el(d), sl(d))\} \cup \{(el(v), sl(v))\} \text{ for every } d \in D,\, v \in V\ \}.$$

With the nodes and edges introduced above, a solution of the vehicle rescheduling problem can be determined by using the network (N, E). We give an integer vector x for every edge of the network. Every $p \in P$ defines a separate commodity for this network. Commodities share the same N set of nodes, but may contain different edges between these nodes. For every edge $e : (n_1, n_2)$ we use the notation x_e^p if there is a directed e edge present between nodes $n_1, n_2 \in N$ in commodity p. We also introduce a variable w_t for every $t \in T$, which allows the cancellation of t.

The difference between the original schedule and the resulting schedule should also be modeled. For every vehicle v, and trip-edge $e \in J^v$, we introduce a value

$$q_e = \begin{cases} 1, & \text{if } v \text{ carries out } e \text{ in the pre-planned schedule} \\ 0, & \text{otherwise} \end{cases}$$

This value will influence the cost of a trip edge e, because αq_e will be added to the cost of each such edge, where α is the penalty for deviation from the original schedule.

To allow lateness for trips, we introduce variable z_t, which gives a new departure time for every trip t. This value includes the added lateness, if any. In order to satisfy the feasible connections between late trips, a constraint has to be added that examines trip compatibilities with respect to z_t:

$$(z_t + length(t) + deadhead_{t,t'} - z_t') \sum_{p \in P} x^p_{a(t),d(t')} \leq 0, \forall (t, t') \in E, \tag{5.1}$$

where $length(t)$ gives the running time of service trip t, and $deadhead_{t,t'}$ represents the running time of the deadhead trip between $el(t)$ and $sl(t')$. Constraint (5.1) is not linear, but it can be rewritten as such with the introduction of a large constant M, as seen in [43].

The IP model of the problem can be formalized in the following way:

$$\text{minimize} \sum_{e \in E} \sum_{p \in P} c_e^p x_e^p + \sum_{t \in T} \beta(z_t - start(t)) + \sum_{t \in T} \gamma w_t, \tag{5.2}$$

s.t.

$$\sum_{p \in g(t)} x^p_{dt(t),at(t)} + w_t = 1, \forall t \in T \tag{5.3}$$

$$\sum_{e:(sl(d),dt(t)) \in L^d} x^d_e \leq k(d), \forall d \in D \tag{5.4}$$

$$\sum_{e:(sl(v),dt(t)) \in L^v} x^v_e = 1, \forall v \in V \tag{5.5}$$

$$\sum_{e \in \delta^-(n)} x^p_e - \sum_{e \in \delta^+(n)} x^p_e = 0, \forall p \in P, \forall n \in N \tag{5.6}$$

$$z_t + length(t) + deadhead_{t,t'} - z'_t \leq \sum_{p \in P} 1 - x^p_{a(t),d(t')} M, \forall (t,t') \in E \tag{5.7}$$

$$z_t \geq dt(t), \forall t \in T \tag{5.8}$$

$$z_t \leq dt(t) + L, \forall t \in T \tag{5.9}$$

$$x^p_e, w_t \in \{0,1\}, \forall e \in E, p \in P, t \in T \tag{5.10}$$

Due to constraint (5.3), every trip is either executed exactly once, or canceled. Constraint (5.4) gives maximum capacities for the depots of the problem, while vehicles in service always have assigned tasks according to constraint (5.5). Constraint (5.6) ensures flow conservation. Constraint (5.7) is the linear reformulation of (5.1). Constraints (5.8) and (5.9) limit the values of the trip starting times: a trip $t \in T$ will always depart in the $[dt(t), dt(t) + L]$ time window, where $dt(t)$ is the departure time of t and L is the maximum allowed lateness. Values β and γ are penalty parameters for lateness and trips cancellation respectively. c^p_e gives the corresponding operational cost of edge x^p_e, along with the possible added penalty of deviation from the original schedule given by αq_e. The set of edges leaving and entering node n are denoted by $\delta^+(n)$ and $\delta^-(n)$ respectively.

As it was mentioned above, the CBN cannot be solved directly for bigger input due to its large size. The model size can be decreased by reformulating it, which will allow solutions for larger problem sizes as well. One such reformulation will result in the TSN.

Time-space network model

The TSN does not model every explicit connection between trips: it aims to reduce the number of possible deadhead edges by carrying multiple connections on a single edge. The node set of this model is the same as the set N we defined above. A time-line is created for each geographical location and depot. These time-lines contain all the departure and arrival nodes belonging to their respective depot or location. This structure can be utilized to aggregate connection edges of the network. This is done by introducing waiting edges between the pairs of adjacent events on every time-line. Let W^d be the set of waiting edges for vehicles belonging to depot $d \in D$, and W^v be the set of waiting edges for a vehicle $v \in V$ currently in service. We use the edge aggregation technique from [65] for the pull-in/pull-out edges and the possible deadhead edges. This gives us a set K^d representing the possible connections between locations for vehicles belonging to depot $d \in D$ while set K^v gives the possible connections between locations for vehicle $v \in V$. Sets L^d and L^v are given in a similar way. Edge sets J^d and J^v representing timetabled trips remain the same. The circulation edges for the depots are given by the set

$$C^d = \{(el(d), sl(d)) | \forall d \in 1D\},$$

and let

$$C^v = \{(el(v), sl(v)) | \forall v \in V\}$$

represent the circulation edges of the vehicle.

The set of edges for the TSN formulation of our problem is given as:

$$E = \{J^d \cup J^v \cup K^d \cup K^v \cup L^d \cup L^v \cup C^d \cup C^d, \forall d \in D, \forall v \in V\}.$$

The IP formulation of the TSN is similar to the CBN model presented above:

$$\text{minimize} \sum_{e \in E} \sum_{p \in P} c_e^p x_e^p + \sum_{t \in T} \beta(z_t - start(t)) + \sum_{t \in T} \gamma w_t, \tag{5.11}$$

s.t.

$$\sum_{p \in g(t)} x^p_{dt(t),at(t)} + w_t = 1, \forall t \in T \tag{5.12}$$

$$\sum_{e:(sl(d),dt(t)) \in L^{d'}} x_e^d \leq k(d), \forall d \in D \tag{5.13}$$

$$\sum_{e:(sl(v),dt(t)) \in L^{v'}} x_e^v = 1, \forall v \in V \tag{5.14}$$

$$\sum_{e \in \delta^-(n)} x_e^p - \sum_{e \in \delta^+(n)} x_e^p = 0, \forall p \in P, \forall n \in N \tag{5.15}$$

$$z_t + length(t) + deadhead_{t,t'} - z'_t \leq \sum_{p \in P} 1 - x^p_{a(t),d(t')} M, \forall (t,t') \in E \tag{5.16}$$

$$z_t \geq dt(t), \forall t \in T \tag{5.17}$$

$$z_t \leq dt(t) + L, \forall t \in T \tag{5.18}$$

$$w_t \in \{0,1\}, \forall t \in T \tag{5.19}$$

$$x_e^p \text{ integer}, \forall e \in E, p \in Pt \in T \tag{5.20}$$

Trips are either executed exactly once or canceled due to constraint (5.12). The maximum capacity of the depots is given by constraint (5.13), and constraint (5.14) ensures that vehicles in service are always considered in the final solution. (5.15) is the flow conservation constraint. Constraint (5.16) controls the maximum lateness for the trips, (5.17) and (5.18) give the time window for a feasible late departure time.

An illustrative example

To show the difference between the two models presented above, we give a small illustrative example in this section. Let us consider an original daily schedule of 4 trips (t_1, t_2, t_3, and t_4), and a single depot d. Trips t_1 and t_2 have a starting location A and ending location B, while trips t_3 and t_4 start at location C and end at location D. Both t_1 and t_2 are compatible with trips t_3, t_4. The original schedules uses vehicles v and v' (from depot d) to serve the trips in the following way:

1. vehicle v executes trips t_1 and t_3
2. vehicle v' executes trips t_2 and t_4

Let us suppose that vehicle v' is no longer available as a result of a disruption. This means that the trips of our problem can either be served by vehicle v, or new vehicle(s) from depot d. Figure 5.2 illustrates the network of our first model for the above problem. The green edges represent possible connections belonging to vehicle v, while the red edges give all the connections for any new vehicle from depot d. The black edges belong to the trips, which either have to be executed or canceled.

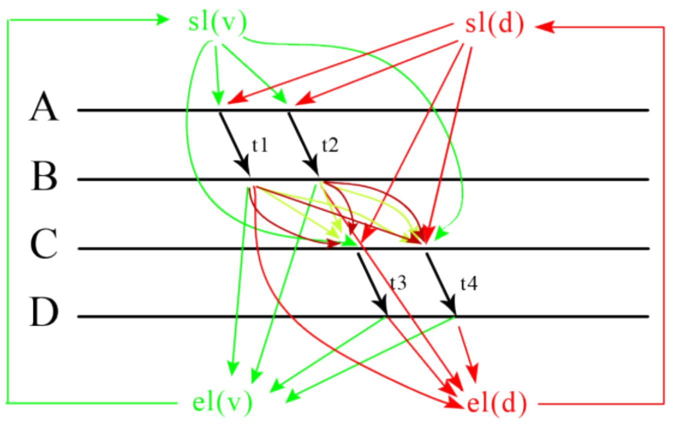


Figure 5.2: Example for the connection based model of the VRSP

It can be seen in Figure 5.2 that the number of all possible connections is large even for such a small example. This is the reason why we introduced the TSN network for the problem, which uses so-called waiting edges to aggregate these connections. Figure 5.3 shows the TSN network of the problem.

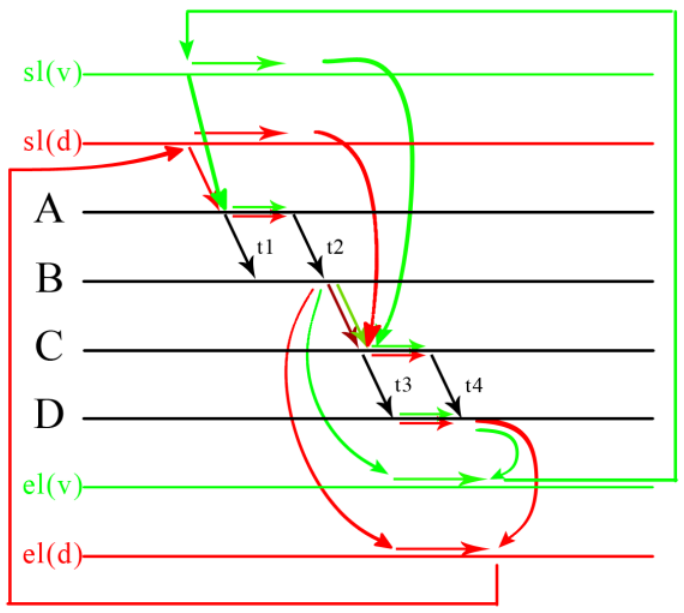


Figure 5.3: Example for the time-space model of the VRSP

5.2.3 Solution methodology

Recovering from a disruption has to be done as quickly as possible, and fast and efficient solution heuristics are needed because of this. However, the cost-wise optimal solution might not always be the best one regarding operations planning, as results given by an algorithm are usually used as suggestions by an operator of the company. These results help them in making fast decisions about the resolution of arising disruptions. Integrating fast algorithms into a decision support system that provides multiple suggestions can speed up the real-time decision process of the operators of the company: in practice, they have to come up with an alternative solution for the disruption, make the final decisions, and communicate it to the affected employee of the company, all in a matter of minutes. For this reason, the running time of a solution algorithm is just as important as the quality of the provided result when measuring its efficiency.

In this section, we present two fast solution methods that can provide multiple different good quality solutions for the VRSP in a short time. These methods can easily be integrated into a decision support system for disruption management in public transportation, which recommends possible solutions for the operators depending on their parameter settings. We developed a decision support system similar to the one that will be presented in Section 7.2, and these heuristic methods also performed well as part of such a system .

A recursive search algorithm

The first algorithm we propose for the problem is a recursive search heuristic. Recursive search seems an ideal method because of the above described expectations: the algorithm is able to find multiple solutions with a short running time. Our method can be seen in Algorithm 7.

The input of the algorithm is the following:

- $vBlocks$: The disrupted schedule of the planning unit. It includes all available vehicles and the blocks assigned to the vehicles in service.
- $dTrips$: The set of all disrupted trips that have no assigned vehicle blocks.
- mod: A non-negative integer parameter that limits the number of modified vehicle blocks, and as a result, it also limits the depth of the search tree. This parameter is reduced whenever one or more trips are removed from a vehicle block (as a result of another trip being inserted).

The input of the heuristic is the set of feasible vehicle blocks, and the set of disrupted trips. Every function call chooses the disrupted trip dt with the earliest departure time, and tries to insert it into every compatible vehicle block vb. There are three possibilities for every $dt - vb$ pair:

- One or more trips have to be removed from vb. The removed trips are flagged as temporary disrupted trips.
- Trip dt can be inserted into vb, but lateness has to be introduced for some of the trips of vb.
- Trip dt can be inserted into vb without additional modifications.

If the set of disrupted trips is empty after a modification, and there are no temporary disrupted trips, the heuristic has found a feasible solution, which is saved. Otherwise, the recursive function is called with new parameters:

- $vBlocks'$: The original $vBlocks$ is updated with the modified block.
- $dTrips'$: The temporary disrupted trips are inserted into $dTrips$, while dt is removed.
- mod': If the size of $dTrips'$ is smaller than the size of $dTrips$, $mod' = mod$. Otherwise, $mod' = mod - 1$.

The algorithm will explore the solution space determined by the trips and the blocks of the problem, examining every possible solution found during its runtime. The depth of this search tree

Algorithm 7 Recursive search for vehicle rescheduling.

```
procedure RecSearch(vBlocks, dTrips, mod)
    if mod = 0 then
        return 0
    end if
    for i = 1 to Size(dTrips) do
        Trip = dTrips[i]
        for j = 1 to Size(vBlocks) do
            nBlocks = vBlocks
            nTrips = dTrips
            Block = vBlocks[j]
            if Trip and Block are not compatible then
                continue
            end if
            if Trip overlaps with trips in Block then
                if Possible solution with lateness then
                    Block' = Insert Trip into Block with added lateness
                    nBlocks' = nBlocks with Block' inserted into nBlocks[j]
                    nTrips' = nTrips without Trip
                    if Size(nTrips') = 0 then
                        Add(Solutions, nBlocks')
                    else
                        RecSearch(nBlocks', nTrips', mod)
                    end if
                end if
                tRemoved = overlapping trips from Block
            end if
            Insert Trip into Block
            Remove Trip from nTrips
            nBlock[j] = Block
            if Size(nTrips) = 0 then
                Add(Solutions, nBlocks)
            else if Size(tRemoved) > 0 then
                Add(nTrips, tRemoved)
                RecSearch(nBlocks, nTrips, mod-1)
            else
                RecSearch(nBlocks, nTrips, mod)
            end if
        end for
    end for
    return Best solution in Solutions
end procedure
```

is limited by the parameter *mod*. Further limitations can also be introduced into the method to exclude visiting similar configurations multiple times.

These limitations (especially the parameter for the number of modified blocks) help to keep the

Algorithm 8 Local search for vehicle rescheduling.

```
1: procedure LocSearch(vBlocks, dTrips, tRange)
2:     Build infeasible block dt from dTrips
3:     Label dt temporary
4:     Add(dt to vBlocks)
5:     tabuList = list of forbidden transformations
6:     while notEmpty(dt) & !(terminatingConditions)  do
7:         tmpSchedules = empty container for vehicle schedules
8:         for i = 1 to Size(vBlocks) do
9:             for j = i+1 to Size(vBlocks) do
10:                for each neighborhood transformation $t$ do
11:                    if t(vBlocks[i], vBlocks[j]) is not forbidden then
12:                        newSchedule = apply $t$ to vBlocks
13:                        Add(newSchedule, tmpSchedules)
14:                    end if
15:                end for
16:            end for
17:        end for
18:        bSchedule = best schedule from tmpSchedules
19:        tS = transformation that can reverse bSchedule
20:        vBlocks = bSchedule
21:        Add(tabuList, tS)
22:    end while
23:    return vBlocks
24: end procedure
```

running time of the algorithm from exploding. The introduction of this parameter was also based on a practical observation: each level of the recursive search tree corresponds to a vehicle whose original block is modified. Companies want to keep the number of modified vehicle blocks minimal. Because of the way *mod* is decremented, its initial value also defines the maximum number of vehicle blocks from which the algorithm can remove trips. As it was mentioned in Section 5.2.1, deviation from the original schedule should have a high cost, so it is unlikely that good quality solutions are cut from the search tree by this parameter.

The algorithm terminates after it has traversed the above defined search tree, and the solution with the lowest cost is returned as a result. If there are any trips left in the set *dTrips* of disrupted trips, then those are considered to be canceled.

A local search algorithm

Our other algorithm for finding a feasible solution for the VRSP is a tabu search heuristic. A brief outline of the algorithm can be seen in Algorithm 8.

The input of the algorithm is the following:

- *vBlocks*: The disrupted schedule of the planning unit. It includes all available vehicles, with undisrupted blocks assigned to their original vehicles.

- *dTrips*: The set of disrupted trips. These are currently not executed by vehicles, and have to be assigned to vehicle blocks.
- *tRange*: Gives a time window in which the events are considered. The time window begins at the start time of the disruption, and ends after the ending time of the last disrupted trip.

The initial candidate solution of the algorithm is constructed from the original vehicle schedule. A new vehicle block is added to the schedule that contains all disrupted trips, sorted in ascending order of their departure time. If there are more than one disrupted trips, this new block is likely to be infeasible, which will also make our initial solution infeasible. This new block is labeled as a *temporary block*.

In each iteration, the algorithm will examine all (i, j) block pairs. It checks the trips of the blocks that are in the given *tRange* time window, and examines the following two neighborhood transformations:

- **1-move:** Moves a trip from block i to block j. This transformation is not carried out, if j is a temporary block.
- **1-change:** Exchanges a trip from block i with the corresponding trip(s) from block j. This transformation is not carried out, if any of the blocks are temporary.

All feasible neighbors given by the above transformations are assigned a cost. This cost is computed from the operational cost of the blocks and the penalties introduced in Section 5.2.1. If the transformation moves a trip from a temporary schedule to another schedule, a high negative penalty is added to decrease the cost, which will make it more likely to be chosen in an early iteration of the local search. Trips on a temporary schedule contribute the penalty for cancellation instead of their operational costs.

The local search algorithm chooses the neighbor solution with the lowest cost as its new candidate. If any trip t was removed from a block B in the process, the (t, B) pair is saved on a tabu list. For every (t, B) pair on the tabu list, trip t cannot be moved to block B by any of the transformations.

The algorithm terminates when at least one of the following terminating conditions is met:

- Limit for the running time.
- If the difference in quality of the consecutive candidates is always below a given gap for a fixed amount of iterations.

The feasible solution with the lowest cost is returned by the algorithm as a result.

5.3 Dynamic vehicle rescheduling

The standard VRSP presented in Section 5.2 only deals with solving a single possible disruption scenario, providing a result that is as close to the optimal solution as possible. However, if we consider the practical application of the problem, this approach is usually not a valid one. There are several disruptions happening to a pre-planned vehicle schedule over a planning unit, and after the first one has been resolved, the second disruption will impact the rescheduled vehicle blocks instead of the original schedule.

Addressing subsequent disruptions happening over a planing unit is similar to solving an online problem. An online algorithm receives a series of requests that have to be served in the order of occurrence, without any additional information about future ones. In our case, these requests correspond to the disruptions of the planning unit, and they arrive in the same order as they would occur. We have to address each disruption without having any further information about future disruptions, and modify the current schedule accordingly.

The quality of disruption management tools should be measured over the entire horizon of the planning unit, and not by evaluating every disruption separately. For this, we introduced the dynamic vehicle rescheduling problem (DVRSP) in [37], which we will present in this section. The DVRSP aims to give a feasible solution for a planning unit with multiple disruptions, where all the disruptions are resolved first before evaluating the effectiveness of their solutions methods. The input of the DVRSP is the original vehicle schedule of the planning unit, and an ordered list of disruptions which have to be solved in the order of occurrence. The output of the problem is a final feasible vehicle schedule. While the VRSP wants to solve a single disruption to optimality, the DVRSP aims to give a solution by the end of the planning unit that is as close to the pre-planned schedule as possible.

As the input of the problem occur in an online manner, we cannot give the optimal solution for the problem. However, we will also introduce the problem of the quasi-static DVRSP (QDVRSP): this acts as the theoretical 'offline' case of the problem, where all the disruptions of the planning unit are known in advance.

5.3.1 The dynamic vehicle rescheduling problem

The DVRSP is a special case of the VRSP, and because of this, its problem definition will be similar to the one given in Section 5.2.1. The main difference between the DVRSP and the QDVRSP is that while the DVRSP solves the series of disruptions by always considering a single disruption at a given time, the solution of the QDVRSP presents the theoretical scenario of rescheduling the entire planning unit if all the disruptions were known in advance.

The input for the QDVRSP considers the entire original schedule DS of the planning unit, with the set T of timetabled trips. Set DT will represent trips that did not belong to the planning unit originally, but have to be executed as a result of a disruption. Periods where certain vehicles are not available over due disruptions are represented by a list S of 3-tuples (v, s_1, s_2). Such a tuple denotes vehicle v not being able to service any trips between times s_1 and s_2. Similarly to the VRSP, we also aim to give a feasible solution for the problem by executing (or canceling) all trips $T \cup DT$, minimizing the arising costs. The costs of the problem are the same ones that were introduced in Section 5.2.1: the standard operational costs, and penalties for deviation from the original schedule, lateness, and trip cancellation.

The mathematical models we introduced for this problem in [37] are also similar in structure to the ones presented for the VRSP in Section 5.2.2. As the TSN model proved to be more useful in the case of the VRSP, we will only present that for the QDVRSP.

Time-space network model for the quasi-static DVRSP

As it was mentioned above, the QDVRSP is a special case of the VRSP, and the model we present in this section will be similar to the one given for the VRSP in Section 5.2.2 because of this. The input of the model is the entire original DS schedule of the company for a given planning unit. Let D be the set of depots (denoting groups of vehicles with the same type and starting/ending geographical locations), V be the set of vehicles in service for the planning unit according to DS. Vehicles in V start the planning unit at the location defined by their respective depots. Let set P denote all possible sources of vehicles for the problem, $|P| = |D \cup V|$.

Let T' be the set of original service trips of the given planning unit, let set DT contain any extra trips that have to be executed due to a disruption, and let CT give all trips that have to be canceled (and are not considered at all because of this) during the day. Let set $T = \{(T' \setminus CT) \cup DT\}$ represent all the trips of our problem. Every trip $t \in T$ has a departure time $dt(t)$, arrival time $at(t)$, starting location $sl(t)$ and ending location $el(t)$. The set of depots and vehicles that can execute a trip t is denoted by $g(t)$. Let $T_d \subseteq T$ be the set of trips that can be executed from depot d, and $T_v \subseteq T$ the set of trips that can be carried out by vehicle v. A a set S of 3-tuples (v, s_1, s_2) is also given that defines unavailability periods for the vehicles. If a vehicle v cannot service any trips due to technical problems between times s_1 and s_2, then it will have no compatible trips that have $s_1 \leq dt(t), at(t) \leq s_2$.

For every depot $d \in D$, we introduce notations $sl(d)$ and $el(d)$. A depot d is represented by $sl(d)$ when we consider it as the starting location of its vehicles, while we use $el(d)$ when it gives the ending location of its vehicles. Similarly for every vehicle $v \in V$ currently in service we define a starting location $sl(v)$ and ending location $el(v)$, which will both correspond to the location of the depot where the vehicle belongs. The set of nodes of our network will be the following:

$$N = \{dt(t) \cup at(t) \cup sl(d) \cup el(d) \cup sl(v) \cup el(v) | t \in T, d \in D, v \in V\}.$$

The edges of the network can be defined using the above nodes. Let

$$J^d = \{(dt(t), at(t)) | t \in T_d\}$$

be the set of trips that can be served by vehicles located at depot d, and let

$$J^v = \{(dt(t), at(t)) | t \in T_v\}$$

be the set of trips that can be executed by vehicle v currently in service. A time-line is created for every geographical location, depot, and vehicle. Time-lines contain all departure and arrival nodes belonging to their respective depots or locations. Let W^d be the set of all waiting edges for vehicles belonging to depot $d \in D$, and W^v be the set of waiting edges for a vehicle $v \in V$ originally in service. Similarly to Section 5.2.2, we apply the edge aggregation technique from [65] with the help of these time-lines. This results in an edge set K^d representing all possible connections between locations for vehicles belonging to depot $d \in D$ while set K^v gives the possible connection edges between locations for vehicle $v \in V$. Let

$$L^d = \{(sl(d), dt(t)), (at(t), el(d)) | t \in T_d\}$$

be all the pull-in and pull-out edges of vehicles located at depot d, and let

$$L^v = \{(sl(d), dt(t)), (at(t), el(d)) | t \in T_v\}$$

be the pull-in and pull-out edges of vehicle v in service. Let

$$C^d = \{(el(d), sl(d)) | \forall d \in D\}$$

be the circulation edges belonging to depots, and

$$C^v = \{(el(v), sl(v)) | \forall v \in V\}$$

be the set of vehicle circulation edges. This results in an edge set

$$E = \{J^d \cup J^v \cup K^d \cup K^v \cup L^d \cup L^v \cup C^d \cup C^v, \forall d \in D, \forall v \in V\}.$$

for the time-space network of the QDVRSP. While the underlying network of the QDVRSP has some differences compared with the one given for the VRSP, the IP formulation of its TSN is exactly the same as the mathematical model in Section 5.2.2:

$$\text{minimize} \sum_{e \in E} \sum_{p \in P} c_e^p x_e^p + \sum_{t \in T} \beta(z_t - start(t)) + \sum_{t \in T} \gamma w_t, \tag{5.21}$$

s.t.

$$\sum_{p \in g(t)} x^p_{dt(t),at(t)} + w_t = 1, \forall t \in T \tag{5.22}$$

$$\sum_{e:(sl(d),dt(t))\in L^{d'}} x_e^d \leq k(d), \forall d \in D \tag{5.23}$$

$$\sum_{e:(sl(v),dt(t))\in L^{v'}} x_e^v = 1, \forall v \in V \tag{5.24}$$

$$\sum_{e\in\delta^-(n)} x_e^p - \sum_{e\in\delta^+(n)} x_e^p = 0, \forall p \in P, \forall n \in N \tag{5.25}$$

$$z_t + length(t) + deadhead_{t,t'} - z_t' \leq \sum_{p\in P} 1 - x^p_{a(t),d(t')} M, \forall (t,t') \in E \tag{5.26}$$

$$z_t \geq dt(t), \forall t \in T \tag{5.27}$$

$$z_t \leq dt(t) + L, \forall t \in T \tag{5.28}$$

$$w_t \in \{0,1\}, \forall t \in T \tag{5.29}$$

$$x_e^p \text{ integer}, \forall e \in E, \forall p \in Pt \in T \tag{5.30}$$

Once again, trips of the planning unit are either executed exactly once or canceled due to constraint (5.22). The maximum capacity for vehicles waiting in depots is given by constraint (5.23), while vehicles that are used in original schedule have to execute at least one compatible trip (if they have any) due to constraint (5.24). Flow conservation is ensured by constraint (5.25), and maximum lateness is controlled by constraint (5.26), with constraints (5.27) and (5.28) defining time windows for late departure times.

The set of edges leaving and entering node n are denoted by $\delta^+(n)$ and $\delta^-(n)$ respectively. Operational costs belonging to edge e^p (and decision variable x_e^p) are given by c_e^p, with the added penalty αq_e if trip edge e was not executed by vehicle p in the original schedule. Penalties for lateness and trip cancellation are given by values β and γ respectively.

Managing disruptions over a planning unit

As it was mentioned before, the model we presented for the QDVRSP above is similar to the TSN model of the VRSP given in Section 5.2.2. However, there is a fundamental difference in how they consider disruption management. While our VRSP model follows the traditional method of dealing with a single disruption, and solving it to optimality, this is usually only useful from a theoretical point of view. Solving a single disruption scenario has to be done efficiently, but the effect of this solution to the rest of the planning unit should also be considered; a solution that is good cost-wise might make future disruptions harder to address.

We introduced the concept of DVRSP to propose a more application-oriented approach to vehicle disruption management. From an operational point of view, the efficiency of a disruption management system should be measured by the combined quality of every change over the horizon of the planning unit, and not by evaluating the solution of every disruption as a separate problem. As real-life disruptions happen in an online manner, giving an optimal final schedule for the planning unit would mean knowing every information in advance. This is of course not possible in a real-life scenario, which means that heuristic solution methods have to be considered. In the theoretical case where all the disruptions are known, the quasi-static model of the DVRSP can be used to evaluate the series of decisions done by a disruption management system at the end of a planning unit.

This philosophy will be applied in the following section when we evaluate the heuristic approaches proposed in Sections 5.2.3 and 5.2.3. We use these methods for a series of disruptions on the same pre-planned schedule, and present the quality of the solution by comparing the modified schedule at the end of the planning unit to the result given by the corresponding QDVRSP.

5.4 Test results

In this section, we examine the efficiency of the heuristic methods presented in Sections 5.2.3 and 5.2.3. The quality of their solutions is measured over a planning unit, after the solution of multiple disruption scenarios. Solutions for moderate-size instances are compared to the results of the quasi-static DVRSP presented in 5.3.1, while their efficiency on larger input is also presented.

All test instances consider a single day as the planning unit, and input data for their schedules is generated randomly using the method in 7.4.2. Pre-planned schedules are obtained for these days by solving the TSN network in Section 2.2.3 for their input. Disruptions are also generated randomly for these original schedules; for this, we use a structural aspect of these problems. One of the most important structural features of a real-life workday instance can be seen on the histogram of its traffic load.

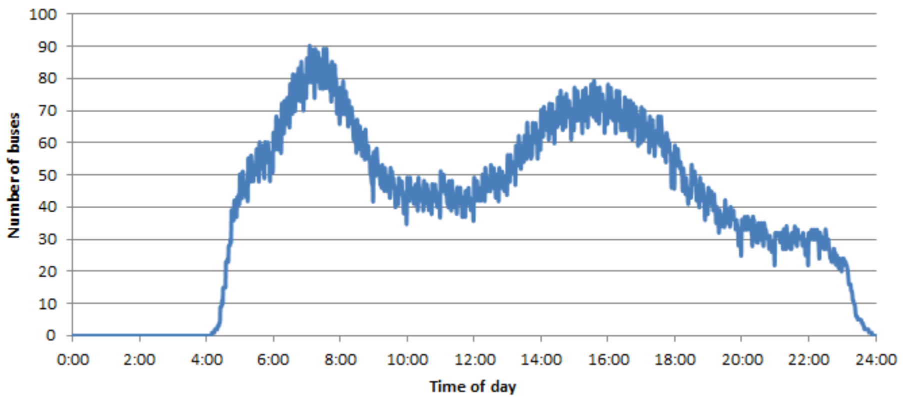


Figure 5.4: Traffic load of a typical workday in Szeged, Hungary

Figure 5.4 shows the traffic load on a typical workday in the city of Szeged, Hungary. The time of day is given by the horizontal axis, while the vertical axis shows the number of vehicles required to service all timetabled trips at a given minute. It can be seen in the figure that there are two peaks: one starting around 6:00, and the other around 15:00. These correspond to the morning and afternoon rush hours. The number of vehicles given by the figure is only a lower bound on the number of vehicles needed in that minute, as there might also be other ones traveling between locations without servicing a timetabled trip. The peak point on the curve can give us a lower bound on the total number of vehicles required for the whole day. The exact structure and the intervals of these peaks may vary depending on the country or the transportation company, but the one we present here is typical for Hungarian bus transportation.

As the structure of the random input instances is similar to that of workdays from real-time scenarios, they also contain similar peaks corresponding to the rush hour periods. Because of this, we decided to generate our random disruptions in these time periods.

Four disruptions were generated for each input instance: two of these overlap with the morning peak period (their starting time is given in a uniformly random way between 6:30-9:30), while the other two are generated for the afternoon peak period (their starting time is also given in a uniformly random way between 15:30-18:30). The length of these disruptions correspond to that of a short trip (see Section 2.2.3), which is also generated randomly. The disrupted vehicle is also chosen in a uniform random way for each disruption.

We solved the DVRSP by running the heuristics sequentially for the disruptions: in the first step, the input of a heuristic was the original daily schedule and the first disruption. Each additional step received the output of the previous step, and the next disruption as its input. The output of the final step is the solution for the problem. We tested our heuristic methods in different ways:

- either all steps of the iteration were solved with the same heuristic,
- or both heuristics were applied in each step, and the result with the better cost was chosen.

The results we received for middle-sized instances were compared to the solution of the TSN-based mathematical model of the same problem, which was solved for the whole 'offline day'. We generated problems with 3 different sizes: 100, 500 and 800 trips. All test instances had vehicles belonging to 4 different depots. For each problem, we also randomized the above disruption scenarios separately.

The cost function of our solution methods was the total distance traveled by the vehicles. We introduced the following multiplicative penalties:

- The cost of each trip that was executed by a different vehicle than in the original solution was multiplied by 2.
- The penalty of a late trip depended on its lateness. If the lateness of the trip was at most 5 minutes, then its cost was multiplied by 2. The multiplicative penalty was 3 otherwise.

- The cost of every task for a newly introduced vehicle was multiplied by 3.
- The cost of canceled trips was multiplied by 30, in order to discourage the algorithms from canceling trips.

The *mod* parameter for the recursive algorithm was set to 4, while the terminating condition for the local search was 50 iterations, during which it couldn't improve its best solution. We generated ten instances for each trips size. Our test results can be seen in Table 5.1.

The first two columns of the table give the details of the instances: trip and instance number. The remaining columns present the running times in seconds (columns with header *time*) and optimality gaps in percentage (columns with header *gap*) of the instances. Columns with header *Opt* give results for the optimal QDVRSP solution of the 'offline day', while columns with headers *Rec*, *Loc*, and *Mix* present results for the recursive algorithm, local search, or the mixed strategy respectively. For calculating the optimality gap, we only considered the extra costs caused by the disruption management process compared to cost of the original, undisrupted schedule. For example, the cost of the optimal 'offline day' would be given by *cost(Opt)-cost(Original)*, and the gap of the local search would be given by comparing *cost(Loc)-cost(Original)* to that value, where *cost(Original)* is the cost of the pre-planned schedule without any disruptions. The last four columns with headers *extra* consider the entire cost of the rescheduled solutions, and give their extra cost in percentage from the original, undisrupted ones. Column *Opt extra* gives the extra costs of the optimal 'offline day' compared to the cost of the original, undisrupted schedule, while columns *Loc extra*, *Rec extra*, *Mix extra* present the extra costs of the local, recursive, and mixed methods respectively. The last row of every instance size gives average results of all the instances. All tests were carried out on a PC with and Intel Core i5 2.80GHz CPU and 4 GB RAM. The TSN IP model was solved using the COIN-OR Symphony solver.

The heuristics were designed to provide good quality solutions for the DVRSP in a short time. It can be seen from the test results that both algorithms were able to solve the series of 4 disruptions in a couple of seconds, and results were close to the optimal solution. There were a small number of instances which had a large gap (around 10-15% more extra costs than the optimal 'offline' case). However, if we consider the total cost increase of these instances compared to their original, undisrupted schedule, it is well below 1% in every case. The running time of the IP solution is long even for smaller instances, which shows that getting the optimal solution by solving the mathematical model would not be effective in a real-life situation. However, the optimum given by the model is needed for checking the quality of the heuristic solutions.

We also experimented with instances that had different cost penalties. We tested all methods on a small number of problems where either the lateness had no penalty, switching trips between vehicles was free, or introducing vehicles had no extra cost. It seemed from these results that the quality of the solutions is similar to those presented in Table 5.1, and the methods perform similarly regardless of the costs of the problem.

Table 5.1: Test results of the heuristic methods for the DVRSP

Trips	Instance	Opt time	Loc gap (%)	Loc time	Rec gap (%)	Rec time	Mix gap (%)	Mix time	Opt extra	Loc extra	Rec extra	Mix extra
100	1	0.82	0.00	0.60	0.00	0.59	0.00	0.61	3.44	3.44	3.44	3.44
	2	0.77	0.00	0.60	0.00	0.62	0.00	0.69	1.94	1.94	1.94	1.94
	3	0.78	4.61	0.60	4.61	1.03	4.61	1.03	5.94	6.22	6.22	6.22
	4	0.83	2.03	0.59	2.03	0.63	2.03	0.65	4.56	4.66	4.66	4.66
	5	0.75	0.32	0.64	0.32	0.62	0.32	0.59	2.70	2.71	2.71	2.71
	6	0.89	0.00	0.59	0.00	0.69	0.00	0.67	3.16	3.16	3.16	3.16
	7	0.93	0.00	0.62	0.00	0.83	0.00	0.84	3.54	3.54	3.48	3.48
	8	0.88	0.43	0.60	0.43	0.61	0.43	0.61	1.84	1.84	1.84	1.84
	9	0.80	3.47	0.60	2.54	0.60	2.54	0.65	3.96	4.10	4.06	4.06
	10	0.83	3.95	0.58	3.95	0.57	3.95	0.60	1.36	1.42	1.42	1.42
	average	**0.83**	**1.77**	**0.60**	**1.44**	**0.68**	**1.44**	**0.69**	**3.24**	**3.30**	**3.29**	**3.29**
500	1	136.99	1.49	1.18	1.49	2.92	1.49	3.02	0.71	0.72	0.72	0.72
	2	135.06	2.25	5.48	2.25	8.19	2.25	8.31	0.37	0.37	0.37	0.37
	3	214.53	0.00	6.70	0.00	6.57	0.00	13.54	1.29	1.29	1.29	1.29
	4	164.73	1.89	1.26	14.45	1.57	1.89	1.80	0.82	0.84	0.94	0.84
	5	178.57	1.57	1.24	3.84	1.64	1.57	1.78	0.70	0.71	0.73	0.71
	6	165.41	6.39	1.25	6.39	2.21	6.39	2.42	0.60	0.63	0.63	0.63
	7	194.45	0.10	1.24	0.10	2.50	0.10	3.01	0.80	0.81	0.81	0.81
	8	150.20	12.04	1.20	12.04	1.48	12.04	1.48	0.27	0.30	0.30	0.30
	9	152.39	1.78	1.21	1.78	1.36	1.78	1.45	0.48	0.49	0.49	0.49
	10	134.59	9.93	1.21	9.93	1.11	9.93	1.39	0.29	0.32	0.32	0.32
	average	**162.69**	**2.43**	**2.20**	**4.32**	**2.95**	**2.43**	**3.82**	**0.63**	**0.65**	**0.66**	**0.65**

Table 5.1 – continued from previous page

Trips	Instance	Opt	Loc	Loc	Rec	Rec	Mix	Mix	Opt	Loc	Rec	Mix
		time	gap (%)	time	gap (%)	time	gap (%)	time	extra	extra	extra	extra
800	1	1093.58	0.00	1.92	0.00	2.47	0.00	3.00	0.44	0.44	0.44	0.44
	2	1082.26	0.00	2.06	0.00	5.68	0.00	5.25	0.71	0.71	0.71	0.71
	3	1166.19	0.00	1.95	0.00	2.53	0.00	2.55	0.55	0.55	0.55	0.55
	4	21621.93	0.00	2.08	0.00	2.04	0.00	2.35	0.35	0.35	0.35	0.35
	5	1199.61	0.00	2.10	0.00	2.85	0.00	3.19	0.43	0.43	0.43	0.43
	6	1528.00	10.49	1.98	10.49	2.08	10.49	2.51	0.37	0.41	0.41	0.41
	7	1127.50	2.68	2.02	11.48	3.86	2.68	3.80	0.24	0.25	0.27	0.25
	8	976.47	8.65	1.96	8.65	2.29	8.65	2.52	0.17	0.19	0.19	0.19
	9	1025.17	1.66	1.96	1.66	2.57	1.66	2.87	0.30	0.30	0.30	0.30
	10	1011.53	0.75	1.99	0.75	2.87	0.75	3.18	0.26	0.26	0.26	0.26
	average	**3183.22**	**1.63**	**2.00**	**2.31**	**2.92**	**1.76**	**3.12**	**0.38**	**0.39**	**0.39**	**0.39**

The size of the generated test examples was limited by the mathematical model. We chose problems with 800 trips as our biggest instance because they are challenging to solve, and they can also give a good estimate on the efficiency of our heuristics for the DVRSP. For problems with 1000 or more trips it became difficult to solve the model directly, and we ran out of memory during the construction of the model. Because of this, we cannot provide any information about the optimal cost of instances of this size. However, the heuristics were able to solve even these bigger instances in a short time. Running times for these instances can be seen in Table 5.2.

Table 5.2: Running time of the DVRSP heuristics for larger instances

Instance	Trips	Rec.(s)	Loc.(s)	Mixed.(s)
Rand4	1000	3.82	2.45	4.302
Rand5	1500	2.67	2.83	2.68
Real1	2762	3.47	3.63	3.62

For larger instances, we ran the heuristics for both randomly generated (Rand) and real-life (Real) problems. As we mentioned above, we could not compare the quality of our solutions to that of the optimal one, but it can be seen from the results that the running time was short even for bigger instances. This shows that the proposed heuristic methods are efficient at providing solutions in a short time, while tests on random instances presented that the quality of these solution also remains good for larger problem sizes.

5.5 Summary and remarks

In this Chapter, we studied the field of disruption management, which aims to address unforeseen events happening to the pre-planned schedules of transportation companies. This area is important from an operations management point of view, as multiple disruptions occur on a daily basis in a real-life scenario, and they have to be resolved as soon ss possible to restore the order of the transportation system.

In Section 5.2, we proposed a multi-depot network model for the vehicle rescheduling problem (VRSP), which deals with the solution of a single disruption scenario. As such a problem requires a fast, real-time result, this model can only be used as quality control for quick solution algorithms. We proposed two simple, but effective heuristic approaches for the fast solution of the VRSP. One is a recursive method that traverses the search tree of the problem, and distributes its disrupted trips to the available blocks either by simple insertion, or by deleting overlapping trips from the current block. As the size of this search tree would be extremely large for efficient use, we limit its depth using a simple practical observation: the depth of this search tree corresponds to the number of modified vehicle blocks. Since a useful solution of the VRSP will only contain a small number of modified blocks (as completely rescheduling the itinerary of a large number of vehicles would be hard to manage in real-time), this limit will mostly cut inefficient solutions. The resulting tree can be searched in a short time, and multiple good quality results can be presented as suggestions to the planners of a transportation company. The other method is a tabu search algorithm, which starts from an infeasible solution, and uses its neighborhood transformations to find a good quality feasible one. This method is incentivized to remove trips from its infeasible block first, which leads to the method trying to fix the disrupted scenario rather than wanting to 're-optimize' the already feasible block by moving or swapping trips between them. The tabu search can also provide multiple feasible solutions with a short running time, if needed. Because of their ability to produce multiple good quality solutions in a short time, these algorithms seem suitable for a decision support system that helps the operators of a transportation company in their rescheduling process by giving them possible solution suggestions for the arising problems.

In Section 5.3, we introduced the dynamic vehicle rescheduling problem (DVRSP). While papers dealing with disruption management consider the solution of a single disruption scenario, our aim was to provide an alternative evaluation for rescheduling methods. As multiple disruptions happen during a planning unit, the solutions for these problems should not be evaluated independently. We introduced the concept of DVRSP for handling multiple independent disruptions in vehicle scheduling over a planning unit, usually over a daily horizon. While the classical method of managing disruption with the VRSP focuses on solving single disruptions to optimality, the DVRSP aims for a good quality solution at the end of the planning unit after managing a series of disruptions. Because the problem itself is dynamic, and the input (the disruptions) arrives in an online manner, we also presented the concept of the quasi-static DVRSP, which gives us an 'offline' version of the problem where all the disruptions are known in advance. The quality of a solution for the DVRSP can be measured using this model.

We used both heuristic methods presented for the VRSP to solve a series of disruptions for pre-planned daily schedules, and showed that they give good quality solutions in a short time for the DVRSP. Each input instance was tested independently with both heuristics, and we also showed their combined results, where every disruption scenario was solved by both methods, and the one with the better quality was chosen as the solution of the current step. This somewhat simulates the real-life decision-making process of transportation planners, where they have to choose one of multiple possible solutions for every disruption scenario.

Chapter 6

Vehicle assignment over a planning horizon

While Chapters 3, 4, and 5 all considered problems connected to a single daily vehicle schedule, it is important to remember that the long-term plans of transportation companies are created in advance for a horizon of several days or weeks. In this chapter, we introduce the schedule assignment problem for public bus transportation, which aims to assign the daily vehicle blocks of a planning period to buses in the fleet of a transportation company. One of the most important requirements of this assignment is the compatibility between blocks and vehicles, meaning that certain blocks can only be serviced by buses belonging to given types. Other important constraints come from the fact that the problem considers a long-term plan for several days or weeks; this means that activities connected to the vehicles such as parking and periodic maintenance also have to be taken into account.

Creating a long-term schedule is one of the most important optimization problems of a transportation company. This usually considers a planning period of several days or weeks, with timetabled tasks that have to be serviced every day. An important feature of a real-life problem is that the days of this planning period are usually not completely different from each other, and they can be divided into several day-types (workdays, holidays, etc.). Days that share a day-type have the same underlying timetable of trips, and the same daily schedule will be applied to them because of this.

As a result, vehicle blocks will be the same for every day that share the same day-type. However, the vehicles executing these blocks can be different on two separate days. It was shown in the previous chapters that creating a daily vehicle schedule is a well studied field in the literature. Yet, the assignment of real vehicles to these schedules is not really considered to our knowledge. In Chapter 4, we introduced the vehicle assignment problem for a single day, and the aim of this chapter is to propose a long-term assignment between the daily vehicle blocks and the fleet of a transportation company. While our goal is to give an assignment over every day of the planning horizon, this should not be done sequentially on a day-by-day basis, as the optimal solution is most likely lost in the process. The assignment of every vehicle and task has a global effect on other days of the horizon as well, and all constraints for every day should be considered together because of this.

First, we present the problems of scheduling vehicle activities over a planning horizon, and give a literature overview of these topics in Section 6.1. Using the introduced concepts, we define the schedule assignment problem in Section 6.2, which aims to assign daily blocks to vehicles of the company over a planning period with respect to parking and maintenance constraints. We give a state-expanded multi-commodity flow network for this problem in Section 6.3, and show its results for both real-life and randomly generated instances in Section 6.4. Our preliminary results on schedule assignment can be seen in [33], while the extended model presented in this chapter is given in [38].

6.1 Considering vehicle activities over a planning period

Literature on vehicle scheduling over a planning period is really scarce, publications usually focus on creating optimal schedule for a single day. Papers dealing with a longer horizon usually study rolling stock rotations, vehicle maintenance, or try to integrate driver rostering with vehicle assignment.

6.1.1 Integrated vehicle assignment and driver rostering

Driver rostering aims to assign duties to the workers of the company over a planning horizon under different constraint. While this is a separate research field in itself (see Ernst et al. [46] for a review), there are certain papers dealing with the integration of vehicle assignment.

Peters et al. [86] gave a branch-and-price framework for the problem aided by a GRASP, and presented their results of both real and simulated data. They consider both a primary and secondary job type for the drivers, and address a fleet of heterogeneous vehicles. This problem formulation is further studied by de Matta and Peters in [40], where they present the set covering mathematical model behind the framework.

The vehicle-crew rostering (VCRP) problem is proposed by Mesquita et al. in [79], where they aim to give an assignment between trips, duties, drivers and buses. They propose a preemptive goal programming heuristic, which decomposes the VCRP into daily problems, and joins their outputs to create the final roster.

Sargut et al. [95] consider a multi-objective crew rostering problem, and proposes a model with assignment variables between vehicles and blocks. A tabu search method is proposed for the solution of the problem, and results are presented on smaller instances.

6.1.2 Rolling stock rotations

Papers about rotation planning aim to optimize long distance railway transportation, creating cyclic plans based on a standard week using timetabled trips. Borndörfer et al. [23] present a hypergraph-based MIP model for the rolling stock rotation planning problem for intercity railway. They consider railway vehicle compositions, and also include maintenance constraints and infrastructure capacities, and focus on the cyclic planning period of a standard week. They first studied this model in [21, 22]. Their results are presented on use-case scenarios of the Deutsche Bahn.

Integrating maintenance into the rolling stock circulation problem is also studied by Giacco et al.[50]. They consider the same timetable to be repeated every day (calling it a cyclic timetable), with scheduled train services also given in the input. Their proposed model inserts the maintenance activities into this pre-determined assignment, and aims to produce a cyclic roster where the number of days is minimal. In [49], they describe a framework that sequentially creates rolling stock rosters, then assigns maintenance to these using the above approach. Their results are presented on small scenarios of an Italian railway company.

Lai et al. [67] consider a MIP and a hybrid heuristic model for the rolling stock assignment with maintenance constraints. They examine a single day, which is further divided into two time slots. The assignment of a longer period is done sequentially over these time slots. They conduct optimization on a daily basis, and only consider a look-ahead of 4 days when making decisions. A rolling horizon is used with the above sequential solution approach to give results for a 90-day period.

6.1.3 Bus transportation and maintenance

To our knowledge, the maintenance scheduling problem for buses is only studied by Haghani and Shafahi [57]. They examine the insertion of different maintenance activities into existing bus schedules over several days, but the assignment of schedules to buses is given in the input. They formulate multiple mathematical models for the assignment of maintenance task into time slots of the pre-determined schedules of the buses, and give test results for two different sets; a smaller example, where maintenance is scheduled for a vehicle fleet of 10 buses over a 3-day planning period, and a larger example of 181 buses and 182 days. However, they run the scheduling simulation daily in the latter case, and solve single problems sequentially for each day.

6.1.4 Our contribution

Our motivation behind developing the schedule assignment problem was the introduction of a model which creates a rostering over a longer planning period (several weeks, not just days), where

- every daily block is assigned to the buses in the fleet of the company,

- buses are also sent to garages at the end of every day,
- regular preventive maintenance activities are carried out for every bus,
- and all of the above constraints are optimized together, minimizing the arising travel- and operational costs.

Although all of the above requirements were studied before (see Sections 6.1.1, 6.1.2, and 6.1.3), they either have not been considered together in the same problem, or solutions for a longer period were acquired by a sequential solution of smaller subproblems of days. Considering a longer planning period, both approaches have the same issue: the decisions made when fulfilling a requirement, or developing a daily solution do not only have a local effect on the given day, but affect the entire horizon as well.

Because of this, all constraints for every day should be considered in the same problem, otherwise the optimal, or good quality solutions might be lost in the solution process. Our goal is to give a model that represents the structure of the entire problem, and optimizes the whole horizon at once, considering all arising constraints together.

As the model is capable of providing solutions for a period of multiple weeks, its results can be used in a decision support system to aid long-term planning. Different configurations of vehicle fleet, maintenance and garage capacities and block types can be experimented with, and the resulting possible feasible solutions can help experts of the company in making a decision about the final schedule.

6.2 The schedule assignment problem

As it was discussed earlier in Section 2.2, the resulting schedule of the VSP only gives the vehicle blocks for a single day. This alone, however, is not enough when creating plans in advance for a longer horizon (eg. several weeks, or even months). The days of this planning period are usually divided into different 'day-types' (workday, Saturday, holiday, etc.), and a theoretical vehicle schedule is created for each of these. This means that days belonging to the same day-type will have the exact same vehicle blocks, and same blocks will always have the same vehicle requirements throughout the entire planning period. However, they will not necessarily be executed by the same vehicle on different days.

The input for the schedule assignment problem is the n-day planning period of the company, with each day i having an assigned day-type $type(j)$. The set V of vehicles available over the planning period is given as well. A set D of depots is also introduced for these vehicles, and the depot-type $d(v) \in D$ is determined for every $v \in V$. Similarly to the VSP, vehicles belonging to the same depot share the same costs and characteristics. Set G represents garages where vehicles can stay for the night between two days of the planning period.

A theoretical vehicle schedule is also provided for every day-type $type(j)$, which is the set $S(type(j))$ of vehicle blocks that have to be executed on days of the given type. We consider these schedules (and also the blocks contained by them) theoretical, because they only give the sequences of timetabled tasks that have to be executed on days of the given type. However, they do not contain information about the vehicle executing them, and consequently do not include the tasks that are specific to this vehicle on the given day. Mandatory tasks that vehicles have to execute out on a given day include a vehicle leaving its starting garage at the beginning of the day, parking at a garage at the end of the day, or carrying out maintenance. Expanding the theoretical schedules with such activities will be the task of the schedule assignment problem.

A vehicle block $j \in S(type(j))$ also has a binary depot-compatibility vector $\mathbf{v^j} = (v_1, ..., v_{|D|})$. A vehicle from depot d can service block j if and only if $v_d^j = 1$. In some cases, it may be possible for a vehicle to service multiple blocks on the same day. For this, we have to define block-compatibility: two $o, p \in S(type(j))$ blocks of the same day i are compatible with regards to depot d, if both can be serviced by vehicles of the depot (meaning both $v_d^o = 1$ and $v_d^p = 1$), and there is enough time for the vehicle between the ending time of o and the starting time of p to travel from the arrival location of o to the departure location of p with a deadhead trip.

Contrary to the solution of the VSP, the vehicle blocks in the input daily schedules do not include the starting and ending garages, as these will be given by the assignment. Papers dealing with the scheduling of buses usually apply a constraint where vehicles have to return to their starting garage at the end of each day. This may be a viable strategy for local transportation problems, where vehicles only have to travel inside a city to reach their ending destination. However, vehicles in intercity transportation do not necessarily end their blocks close to their starting garage, and returning back there might be expensive. Instead of this, a garage $g \in G$ also has to be assigned to each vehicle at the end of each day, where it will stay for the night and begin the next day of the planning period. Arising travel costs should also be considered when choosing this garage, as the vehicle has to travel here from its location at the end of the day, and then also head out to the starting location of its vehicle block on the next day.

We also consider a vehicle specific requirement during the solution of the problem, which is the assignment of mandatory maintenance activities to vehicles. Maintenance activities can usually be of two types: daily inspections are smaller tasks that can be included as tasks in the daily vehicle blocks, or larger mandatory inspections (usually called preventive or periodic inspection) that require an entire day. These large inspections usually have to be executed after a vehicle has been working for a pre-specified time, or covered a set distance while servicing blocks since its last inspection. In our case, we consider the number of days spent in service. Let integer parameter s give the maximum number of days that a vehicle can spend servicing blocks before it has to be sent on such an inspection. A vehicle can undertake an inspection activity anytime at a maintenance location $m \in M$, but vehicles that already reached their maximum runtime of s days have only two choices:

either stay at their current garage, or undertake a periodic inspection at one of the maintenance locations. Similarly to choosing the garages, arising travel costs to and from maintenance locations should also be considered.

The aim of the problem is to assign the blocks to the vehicles of the company over the planning period such that each block is executed exactly once, every vehicle stays at a garage at the end of each day, a vehicle services blocks on at most s days between two inspections, and the arising costs are minimal. A vehicle v from depot d contributes $dc(d) \cdot work_i^v + tc(d) \cdot dist(v)$ to the cost of the problem, where $dc(d)$ and $tc(d)$ are the one-time daily and unit-distance costs of a vehicle from depot d respectively, $dist(v)$ is the distance traveled by vehicle v during the planning period (either by servicing blocks or traveling to/from garages). The binary vector $\mathbf{work^v} = (work_1, ..., work_n)$ denotes whether vehicle v was in service on day i of the planning period, or not.

6.3 Assignment model with state-layers

This section introduces a state expanded multi-commodity network flow model for the schedule assignment problem. The nodes of this network will represent the different tasks that can be carried out by the vehicles (servicing a block, staying at a garage, or having a mechanical inspection), while the edges give the transitions between them.

Let us consider a planning period of n days, and let integer parameter s denote the maximum number of days that a vehicle can spend servicing blocks between two inspections. Whenever a node is said to have *inspection state* h, it can only be carried out by vehicles that have serviced exactly h blocks since their last inspection.

Let B be the node set of vehicle blocks given by the daily schedules of the planning period, hypernode $\mathcal{B}_{i,j} \subseteq B$ representing the vehicle block j on day i, where $1 \leq i \leq n$, $1 \leq j \leq k$, and $k = |S(type(i))|$ is the number or blocks on day i. This hypernode $\mathcal{B}_{i,j} = (b_{i,j}^0, b_{i,j}^1, ..., b_{i,j}^{s-1})$ consists of nodes $b_{i,j}^h$ representing all possible inspection states of the vehicle executing the given block, h giving the inspection state of the node.

Let G be the set of garage nodes for the l garages of the input. Similarly to vehicle blocks, garages are also represented by hypernodes $\mathcal{G}_{i,j} = (g_{i,j}^0, g_{i,j}^1, ..., g_{i,j}^s)$, where a node $g_{i,j}^h$ represents garage j on day i for vehicles in state h ($0 \leq i \leq n$, $1 \leq j \leq l$). The special hypernode $\mathcal{G}_{0,j}$ denotes the garage j at the beginning of the planning period. Every garage i also has a capacity $kg(i)$, which gives the number of vehicles that can simultaneously stay at that garage.

Let M be the set of maintenance nodes, representing geographical locations where the inspections of the vehicles can be carried out, node $m_{i,j} \in M$ standing for location j on day i. Maintenance nodes have a capacity $km(i)$, which gives the number of vehicles that can be serviced there in a single day. It might be possible, that the same geographical location contains both a garage and a maintenance facility, but garage nodes and maintenance nodes are handled as separate entities even in this case: such a location will contribute two nodes ($g \in G$ and $m \in M$) to the model of the problem, and both will have separate $kg(g)$ and $km(m)$ capacities.

Let D be the set of d depots representing the vehicles of the company. Each depot i is defined by two nodes: $d_{i,0}$ represents vehicles of the depot at the beginning of the planning period and $d_{i,1}$ at the end of the planning period. Vehicles belonging to the same depot are of the same type, and share the same costs and characteristics, but they will not necessarily share their starting locations at the beginning of the planning period. Depots also have a capacity $kd(i)$ that gives the number of vehicles available of that type.

The edges of the network can be given using the above nodes. These represent the possible traveling activities of vehicles throughout the planning period, either heading to block nodes to service them, to garage nodes where they stay for the night, or to maintenance nodes for a mechanical inspection. Each depot will have its own set of edges The starting state of the different vehicle types and their location at the beginning of the planning period is represented by depot starting edges

$$E^{ds} = \{(d_{i,0}, g^0_{0,j}) | 1 \leq i \leq d,\ j \text{ can be the starting garage of a vehicle from depot } i\}.$$

Vehicles of each depot ending the planning period in one of the possible garages are represented by depot ending edges

$$E^{de} = \{(g^h_{n,i}, d_{j,1}) | 1 \leq i \leq l, 1 \leq j \leq d, 0 \leq h \leq s\}.$$

Vehicles leaving their garages to execute a block at the beginning of a day are represented by block starting edges

$$E^{bs} = \{(g^h_{i-1,j}, b^h_{i,o}) | 1 \leq i \leq n, 1 \leq j \leq l, 1 \leq o \leq k, 0 \leq h \leq s-1\}.$$

Note that garage nodes in inspection state s cannot send vehicles to execute a block, as they have to carry out a mechanical inspection activity first.

Vehicles returning to garages at the end of the day from a block are represented by block ending edges

$$E^{be} = \{(b^h_{i,o}, g^{h+1}_{i,j}) | 1 \leq i \leq n, 1 \leq j \leq l, 1 \leq o \leq k, 0 \leq h \leq s-1\}.$$

When a vehicle travels through one of these block ending edges, its state (denoting the number of days spent in service) is also increased by one; this is represented by the vehicle moving to another state layer of the network (in the case of the above edges, from layer h to layer $h+1$). Also note that after servicing a vehicle block, the inspection state of the destination garage node has to be at least 1.

As mentioned before, it may be allowed in some cases for a vehicle to service multiple blocks on the same day. For every block-compatible (o, p) pair of blocks, we can introduce block connection edges

$$E^{bc} = \{(b^h_{i,o}, b^h_{i,p}) | 1 \leq i \leq n, 1 \leq o, p \leq k, 0 \leq h \leq s-1\}.$$

These edges will provide the possibility for a vehicle to service multiple compatible blocks on the same day instead of heading back to a garage at the end of its first block. Note, that the state h of the vehicle does not change between executing two blocks, as it is still in service on the given day i. The change of its state will be managed by the block ending edge that is carried out after its last block.

Vehicles leaving their garages for mechanical inspections are represented by inspection starting edges

$$E^{is} = \{(g^h_{i,j}, m_{i,o}) | 1 \leq i \leq n, 1 \leq j \leq l, 1 \leq o \leq |M|, 1 \leq h \leq s\}.$$

It can be noted again that vehicles in garages with inspection state 0 have no reason to execute a mechanical inspection, and therefore these edges are not added to the network.

Vehicles returning to garages after a mechanical inspection are represented by inspection ending edges

$$E^{ie} = \{(m_{i,o}, g^0_{i,j}) | 1 \leq i \leq n, 1 \leq j \leq l, 1 \leq o \leq k\}.$$

A vehicle always arrives at a garage node with an inspection state 0 after a mechanical inspection.

Vehicles staying at a garage for a given day are represented by garage edges

$$E^{g} = \{(g^h_{i-1,j}, g^h_{i,j}) | 1 \leq i \leq n, 1 \leq j \leq l, 1 \leq h \leq s\}.$$

The following circulation edges are also added between all depot ending and starting nodes

$$E^{f} = \{(d_{i,1}, d_{i,0}) | 1 \leq i \leq d\}.$$

An illustration of the network can be seen in Figure 6.1. The figure presents a 3-day planning horizon with a single depot, 2 garages, 2 daily vehicle blocks and 1 maintenance location. The two blocks on the first day are block-compatible (given by edges $(b^0_{0,1}, b^0_{0,2})$ and $(b^1_{0,1}, b^1_{0,2})$). The parameter s for maximal days in service before inspection is set to 2, which can be seen on the three state layers of the network ($s = 0, 1, 2$, represented by the dashed rectangles in the figure).

Using the node set $N = \{B \cup D \cup G \cup M\}$ and edge set $E = \{E^{ds} \cup E^{de} \cup E^{bs} \cup E^{be} \cup E^{bc} \cup E^{is} \cup E^{ie} \cup E^{g} \cup E^{f}\}$ the multi-commodity network (N, E) can be defined. This network will have d separate commodities, one for every depot. The commodities of this network will be denoted by $c \in D$. For each edge e of this network, we give an integer vector x_e. This vector will have one component for every commodity c, which will be denoted by x^c_e. The value x^c_e represents if a vehicle from depot c can be assigned the traveling activity connected to edge e (servicing a block, undertaking maintenance, heading to a garage, staying at a garage). Edges E^{ds}, E^{de}, E^{f} are added for the respective commodity of the depot they represent, while edges in E^{bs} and E^{be} are created

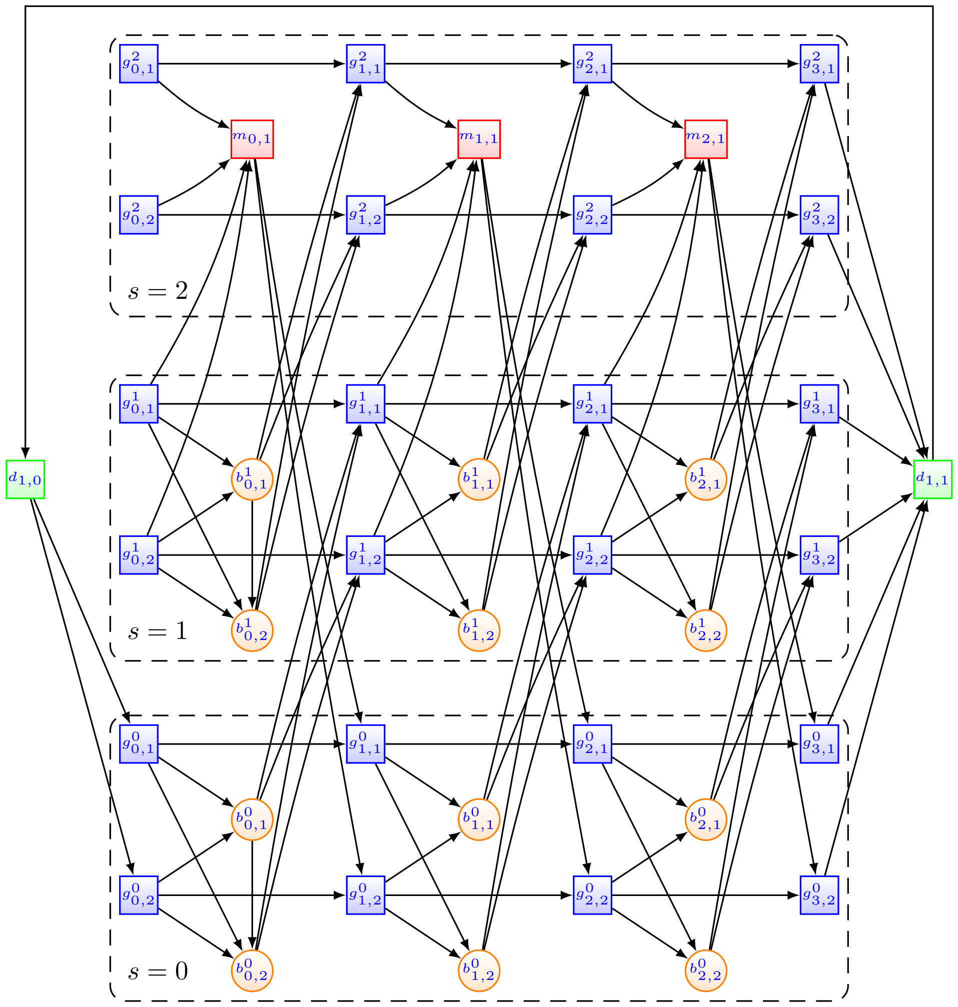


Figure 6.1: Illustration of the schedule assignment model

for every depot d that is able to execute the corresponding block. Other edges are available for all commodities. Notations $\delta^+(n)$ and $\delta^-(n)$ are used to denote the set of arcs leaving node n and entering node n respectively. Based on the above data, the mathematical model can be formalized the following way:

$$\text{minimize} \sum_{c \in D} \sum_{e \in E} tr^c_e x^c_e$$

s.t.

$$\sum_{e \in \delta^-(\mathcal{B}_{i,j})} x_e^c = 1, \forall (i,j) \text{ pair}, 1 \leq c \leq d \tag{6.1}$$

$$\sum_{e \in \delta^+(d_{c,0})} x_e^c \leq kd(c), \forall c \in D \tag{6.2}$$

$$\sum_{e \in \delta^-(\mathcal{G}_{i,j})} x_e^c \leq kg(j), \forall (i,j) \text{ pair}, 1 \leq c \leq d \tag{6.3}$$

$$\sum_{e \in E^{is}_{i,m}} x_e^c \leq km(j), \forall (i,m) \text{ pair}, 1 \leq c \leq d \tag{6.4}$$

$$\sum_{e \in \delta^-(n)} x_e^c - \sum_{e \in \delta^+(n)} x_e^c = 0, \forall c \in D, \forall n \in N \tag{6.5}$$

$$x_e^c \in \{0,1\}, \forall e \in \{E^{ds} \cup E^{de} \cup E^{bs} \cup E^{be} \cup E^{bc}\} \tag{6.6}$$

$$x_e^c \geq 0 \text{ integer}, \forall e \in E^g \cup E^{is} \cup E^{ie} \cup E^f \tag{6.7}$$

Constraint (6.1) determines that a block has to be serviced by exactly one vehicle. Constraint (6.2) gives the vehicle limits for every depot at the beginning of planning period, while (6.3) defines capacities for every garage at the end of every day. Constraint (6.4) sets the daily limits of the maintenance nodes, the possible incoming vehicles for a maintenance node $m \in M$ on day i being represented by edges $E^{is}_{i,m} \subseteq E^{is}$. Flow conservation for every node of the network is given by (6.5), while constraints (6.6) and (6.7) provide the binary and integrality constraints for all the variables.

The objective of the model is to minimize the arising vehicle and travel costs: the cost of a vehicle from commodity c to service the travel activity denoted by edge e is given by tr_e^c, the travel cost of a vehicle from depot c to cover the distance denoted by edge e. If the edge is a travel activity during which the vehicle leaves its current garage, then the cost of the edge is given by $tr_e^c + dc^c$ instead, there dc^c is the one-time daily cost of a vehicle from depot c.

6.4 Test results

The model was tested both on real-life and randomly generated input. Important characteristics of both input types presented in this section along with their solution processes, and the achieved results are also analyzed.

The mathematical model was solved using the Gurobi MIP solver, and ran on a PC with and Intel Core i7 3.30 GHz processor using 32 GB RAM. The time limit for the solver was set to 1 day (86 400 seconds), and the solver optimality gap tolerance was set to 0.00%.

6.4.1 Real-life instances

The real input was part of a 'what-if' scenario, trying to coordinate the transportation of three regions in Hungary. The companies in these regions organized their transportation semi-independently before. The transportation companies provided input for a 3-week long planning period, which consisted of vehicle blocks belonging to 7 different day-types. The important features of the input data can be seen in Table 6.1.

Table 6.1: Real input characteristics for schedule assignment

Vehicles	238
Garages	109
Maintenance locations	6
Average daily blocks	131

Vehicles of the input were separated into three different depots. Vehicles belonging to depot 3 were able to execute any of the vehicle blocks, while vehicles in depot 1 could execute blocks belonging to either depot 1 or 2. Vehicle of depot 1 could not execute blocks belonging to other depots. Blocks of the daily schedules were not block-compatible, meaning that a vehicle could only service a single block on any given day. Using the input data above, we created two main groups of test instances: one with all three vehicle types, and another with depots 2 and 3 merged into a single type. We ran tests for the entire planning period of three weeks and smaller intervals of one and two weeks also. Values between 2 and 6 were all used as the parameter s of maximum working days for every instance type. Different combinations of the above parameters result in a total number of 30 different instances for the real-life input. The model was solved using the constraints introduced at the beginning of this section.

The results of the mathematical model on the above real-life instances can be seen in Table 6.2. Each row of the table presents solution data for a single independent test run; it gives the number of depots used for the problem, the length of the planning period (in weeks), and the value of parameter s (the number of maximum days that a vehicle can spend in service before going to maintenance). It also gives the size (rows and columns) of the resulting mathematical model (denoting the number of constraints and variables), and shows the solution time of the problem (in seconds), with the optimality gap of the achieved result.

It can be seen from the table that solutions of a one-week period are easily reachable, and in most cases, optimal results can also be acquired for a longer planning period of two weeks. Almost optimal solutions are also obtainable for the large instances of the three-week long period. This is especially important when considering the practical application of the model, as the results are promising regarding both the running times and the qualities of the solutions. The easy modification of the parameter s can also help in testing different scenarios for making future decisions.

Table 6.2: Results of the schedule assignment for real-life instances

Depots	Weeks	s	columns	rows	time (s)	gap (%)
		2	397 694	8 422	1 038	0.00
		3	591 051	11 026	114	0.00
	1	4	784 408	13 630	86	0.00
		5	977 765	16 234	84	0.00
		6	1 171 122	18 838	114	0.00
		2	799 652	16 232	5 520	0.00
		3	1 188 717	21 234	4 663	0.00
2	2	4	1 577 782	26 236	2 207	0.00
		5	1 966 847	31 238	15 358	0.05
		6	2 355 912	36 240	29 345	0.00
		2	1 201 610	24 042	67 195	0.00
		3	1 786 383	31 442	48 705	0.00
	3	4	2 371 156	38 842	51 261	0.04
		5	2 955 929	46 242	77 546	0.05
		6	3 540 702	53 642	56 165	0.03
		2	544 730	11 702	585	0.00
		3	808 860	15 487	128	0.00
	1	4	1 072 990	19 272	40	0.00
		5	1 337 120	23 057	39	0.00
		6	1 601 250	26 842	57	0.00
		2	1 094 569	22 467	3 395	0.00
		3	1 625 712	29 725	19 431	0.00
3	2	4	2 156 855	36 983	25 304	0.04
		5	2 687 998	44 241	6 840	0.00
		6	3 219 141	51 499	21 640	0.00
		2	1 644 408	33 232	16 323	0.01
		3	2 442 564	43 963	22 957	0.04
	3	4	3 240 720	54 694	81 941	0.23
		5	4 038 876	65 425	79 146	0.37
		6	4 837 032	76 156	52 454	0.78

6.4.2 Random instances

Our random input data was generated in two steps. First, random VSP inputs were created using the method in Section 7.4.2. These instances had 100, 500, or 1000 trips, and used either 2 or 3 depots. A total of 60 instances were generated this way, 10 for every depot-trip combination. Solving the VSP for all these instances resulted in daily vehicle schedules, which were then used as an input for the schedule assignment problem.

Each vehicle schedule was used as the input of a planning period with a single day-type, and planning periods of one, two and three weeks were all considered for every schedule. Values between 2 and 6 were all used as the parameter s of maximum working days for every instance type. Considering all combinations of the above parameters, we achieved optimal solutions for a total of 900 test runs.

Important features of these input instances are given by Table 6.3.

Table 6.3: Random input characteristics for schedule assignment

Trips	Average vehicles	Garages	Maintenance locations	Average daily blocks
100	51	30	2	17
500	219	40	3	55
1000	433	60	4	99

The aggregated results of the instances presented above can be seen in Table 6.4. Each row of the table provides optimal results for 3 different instance sets, every set representing one of three problem sizes (100, 500 or 1000 trips). The problem size of a set is given by its column header, while additional parameters of these sets are presented in the header of the row: the number of depots, length of the planning period (in weeks) and the value of parameter s for maximum working days. Each set presents the aggregated results of 10 different test instances, giving average model size and running time of the optimal solutions.

Our preliminary test runs for the 900 instances in Table 6.4 were all executed with the same constraints as mentioned before at the beginning of this section. We managed to solve 896 instances to optimality this way within the given time. The remaining 4 instances all belonged to the set of 3 depot problems for a 3 week planning period with the parameter $s = 6$ for maximum working days. The optimality gaps for their results we achieved within the time limit were 0.02%, 0.03%, 0.003%, 0.02% respectively. As these solutions are near-optimal, and we received optimal results for all other test runs, we decided to solve these 4 instances also to optimality without the limit on running time, thus not needing to include data for optimality gaps in the table. Because of this, the table presents an average running time that is greater than the 1-day limit for the last instance set. The above results are also promising, as solutions can easily be obtained for random input of different sizes and parameter combinations. The size of the largest presented problem sets in the table (with the average of 99 daily blocks) can be equivalent to networks of some regions, thus the given model and solutions process can also be applied to real-life instances with similar characteristics.

As it can be seen both from the real-life and randomized test results, there are three main factors influencing problem size and running time. One of these is the value of the parameter s. As the state of the vehicles has to be tracked throughout the network, a separate layer is created for each such state. This basically means the duplication of all garage and block nodes, and these result in more constraints and variables to the problem (capacity constraints and flow conservation) as well as more variables.

Using a heterogeneous fleet with multiple depots also results in an increased number of constraints. Each depot has its own commodity in the network, and similarly to state layers, both flow conservation and node capacities have to be checked separately for every depot. Moreover, there are also constraints linking these different commodities; constraints ensuring that each block is serviced exactly once

Table 6.4: Aggregated results of schedule assignment for 900 random test runs

Depots	Weeks	s	100 trips			500 trips			1000 trips		
			columns	rows	time (s)	columns	rows	time (s)	columns	rows	time (s)
		2	15 787	1 135	0.79	67 641	2 423	2.37	177 985	4 044	7.35
		3	23 335	1 475	1.73	101 001	3 104	4.48	266 077	5 169	8.88
	1	4	30 883	1 814	1.35	134 361	3 785	3.99	354 169	6 293	8.98
		5	38 431	2 153	1.52	167 721	4 466	3.98	442 261	7 418	8.99
		6	45 979	2 493	2.04	201 081	5 148	3.12	530 353	8 543	10.28
		2	31 453	2 177	4.22	135 121	4 722	11.50	355 729	7 904	27.50
		3	46 519	2 826	7.84	201 801	6 044	42.04	531 853	10 093	329.77
2	2	4	61 585	3 474	15.51	268 481	7 366	144.63	707 977	12 282	462.19
		5	76 651	4 123	39.01	335 161	8 689	268.54	884 101	14 472	683.01
		6	91 717	4 772	92.19	401 841	10 011	206.11	1 060 225	16 661	372.70
		2	47 119	3 219	7.48	202 601	7 020	31.31	533 473	11 763	127.11
		3	69 703	4 177	47.62	304 281	9 026	297.94	797 629	15 018	816.75
	3	4	92 287	5 153	225.45	402 601	10 948	10 115.95	1 059 321	18 273	25 449.56
		5	114 871	6 093	6 626.98	502 601	12 911	4 752.78	1 333 186	21 621	30 154.33
		6	137 455	7 051	6 682.53	602 601	14 875	6 374.63	1 598 791	24 889	51 144.84
		2	23 431	1 208	0.76	90 713	2 521	2.97	235 609	4 259	10.51
		3	34 801	1 566	3.00	135 609	3 227	6.41	352 513	5 438	27.78
	1	4	46 171	1 923	2.94	180 505	3 932	9.76	469 417	6 617	32.84
		5	57 541	2 281	2.70	225 401	4 638	10.92	586 321	7 795	40.54
		6	68 911	2 638	3.48	270 297	5 344	6.01	703 225	8 974	13.11
		2	46 741	2 323	3.62	181 265	4 918	11.48	470 977	8 335	52.82
		3	69 451	3 008	18.60	271 017	6 289	51.55	704 725	10 632	527.44
3	2	4	92 161	3 693	34.05	360 769	7 660	363.30	938 473	12 929	2 091.90
		5	114 871	4 378	61.79	450 521	9 032	533.76	1 172 221	15 226	2 843.32
		6	137 581	5 063	116.72	540 273	10 403	1 015.38	1 405 969	17 524	2 015.59
		2	70 051	3 437	5.88	271 817	7 314	39.74	706 345	12 410	125.09
		3	104 101	4 450	50.30	406 425	9 352	499.80	1 056 937	15 826	1 644.36
	3	4	138 151	5 463	619.36	541 033	11 389	2 531.34	1 402 041	19 197	41 684.79
		5	172 201	6 476	9 622.47	675 641	13 426	7 038.57	1 751 261	22 605	51 468.83
		6	206 251	7 488	3 401.35	810 249	15 463	25 688.33	2 100 481	26 014	97 058.60[a]

[a]The running time limit was relaxed for 4 test runs of this instance set in order to achieve optimal solutions in all 900 test cases. See Subsection 6.4.2 for details.

Naturally, the length of the planning period also influences the size of the model. As the schedules of a single week are usually more or less similar in structure, the number of decision variables and constraints is expected to grow in a somewhat linear proportion to the number of weeks considered by the problem. This effect can be observed on the sizes of both the real-life and the randomly generated problems.

The combination of the above three factors (parameter s, number of depots and size of the planning period) contribute together to the problem size and solution running time. It can be seen from both the real-life and random test results that instances with a 1-week planning period are easily solvable in a short time regardless of number of blocks, depots, or s. This is also true in the case of most test instances with a 2-week planning period, slower running times only occurred for some problems with a higher (4 or greater) value of s. The only instance types that constantly resulted in slow running times are the ones with a 3-week planning period, and a value $s \geq 4$. Yet, even solutions for these instances achieved within the given time limit of 1-day were optimal or near-optimal. The model yielded good solutions for real-life data that connected the transportation of 3 different regions, and gave results for a significant planning period of 3 weeks. The largest random instance sets are also comparable with similarly sized real-life scenarios, meaning that the model can be applied generally to such problems.

6.5 Summary and remarks

In this chapter, we introduced the schedule assignment problem for bus transportation over a planning period, where the pre-planned daily vehicle blocks are assigned to buses of a transportation company. Important requirements like daily parking and preventive maintenance have to be taken into account due to the long-term nature of the task. While similar constraints were studied either separately or for shorter periods in the literature, they have not been considered together for a horizon of several weeks.

We presented a mathematical model for the problem using a state-expanded multi-commodity network, which was then solved by a MIP solver. Both real-life and random instances were used as an input, and their results were promising for different number of vehicle types and varying parameters for the time limit of the preventive maintenance. Instead of optimizing the period sequentially on a day-by-day basis, the model represents the structure of the entire planning horizon, and achieves a solution by considering all arising constraints of every day together.

However, the parking and maintenance constraints that we considered for the model are only basic requirements of such an assignment, and the model can be extended to incorporate more sophisticated needs. An obvious addition would be the inclusion of refueling activities at the end of each day for vehicles that have been executing blocks. These could be inserted between the block and garage nodes: vehicles would travel to refueling stations first after finishing their final block on a day, and then pull in to a garage from these stations.

Another important extension would be the consideration of 'vehicle history' at the beginning of the planning horizon: as the planning period is not a stand-alone unit, but the continuation of another period, different vehicles should begin this period in different states with regards to maintenance. This could be achieved by modifying the definition of vehicle start nodes in the model: these should also be present in every state-layer, not just a single one.

Chapter 7

Concepts for decision support systems

As it can be seen from the previous chapters, a transportation system is hard to design, as it incorporates many different modules that have to work together in order to achieve a good quality service. As these problems are already complex by themselves, they are usually solved in a sequential manner. Naturally, good quality solutions are important at every step of this solution process, but the aim of such a system is to provide a globally good result, which should be efficient cost-wise, but should also have a nice and flexible structure that can be easily implemented in practice.

Most companies utilize computer aided planning to achieve this, which considers the optimization algorithms of the subproblems as modules of a decision support system. The solutions given by this system are analyzed by experts, and parts of it are applied in practice. The parts of a transportation system that are connected to vehicles include: planning, evaluation and real-time control. While the planning phase is extensively studied through different problems in the literature, the other two steps are rarely considered as part of a decision support system. In this chapter, we introduce multiple concepts that are not well studied from this application-oriented aspect, but can serve as an important module of such a system. We present real-time control through a decision support framework for vehicle rescheduling, and give a timed automata-based model and a test instance generator that can be used for evaluating different modules of an optimization framework.

First, we review the general outline of the problems in an optimization system for public transit, and highlight the scarcely studied stages of such a system. This overview is presented in Section 7.1.

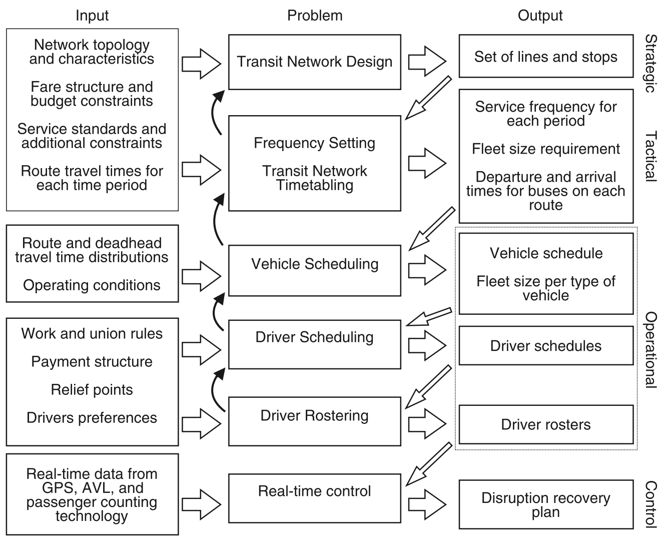
In Section 7.2, we give a framework that can be used by companies to manage disruptions happening during the execution of their schedules. This framework is designed to be flexible enough to work with any number of different solution heuristics for the vehicle rescheduling problem. The basic idea behind this system was published in [12].

Section 7.3 presents a novel modeling approach for problems in public transportation. Instead of the classical mathematical formulations, we propose a timed automata-based model to solve the schedule assignment problem of Chapter 6. This idea was introduced in [39].

The final Section 7.4 presents the random instance generation method that was used for creating input instances for the problems presented in the previous chapters. We modified the state-of-the-art method of Carpaneto et al. [26], and also gave two approaches that consider depots with multiple vehicle types. This method was published in [34].

7.1 Stages of a decision support system

In Chapter 2, we gave a brief overview the most important optimization phases of a transportation system: strategic, tactical, and operational planning. These stages are responsible for the creation of the long-term transportation plan that is executed by the company over a planning horizon. The details of these phases can be seen on Figure 7.1 (taken from Ibarra-Rojas et al. [63]), which presents the majority of the processes handled by an optimization system, with all their interactions and required input information.



Input
Problem
Output
Network topology and characteristics
Fare structure and budget constraints
Service standards and additional constraints
Route travel times for each time period
Transit Network Design
Set of lines and stops
Strategic
Frequency Setting
Transit Network Timetabling
Service frequency for each period
Fleet size requirement
Departure and arrival times for buses on each route
Tactical
Route and deadhead travel time distributions
Operating conditions
Vehicle Scheduling
Vehicle schedule
Fleet size per type of vehicle
Work and union rules
Payment structure
Relief points
Drivers preferences
Driver Scheduling
Driver schedules
Driver Rostering
Driver rosters
Operational
Real-time data from GPS, AVL, and passenger counting technology
Real-time control
Disruption recovery plan
Control

Figure 7.1: The planning process and real-time control of public transit (from [63])

The integration of the above subproblems (especially the ones connected to operational planning) into a decision support system is well studied in the literature (see [15, 77, 80] for some examples). However, additional phases can also be included in a such system, which are generally not considered.

A good illustration for this is the real-time control of a transit system. While the papers dealing with this field try to maintain the regularity of traffic, or to normalize passenger transfer times—usually by holding buses at specific stations, or modifying the frequency of their lines—, different types of real-time problems can also arise in such a system: addressing disruptions is a perfect example, yet vehicle rescheduling problems are usually only considered as a variant of the VSP, and not as a part of decision support.

Another scarcely studied aspect of these systems is the testing and evaluation of the different implemented modules. A method that is going to be introduced into such a system has to be tested extensively, on a much larger dataset than the instances of a single transportation company. As freely accessible benchmark datasets are unfortunately not available for real-life transportation problems, the easiest way to obtain test data an appropriate structure is to generate it. The structure and validity of the implemented methods might also be interesting in some cases. However, these are also not that widely studied.

In the following sections, we will propose concepts and methods for both of the above mentioned problems. First, we introduce a flexible framework for vehicle rescheduling. Then, we present two possibilities for testing and evaluation: a timed automata-based model that can be used to visualize and validate parts of the problem, and a method for generation random instances that can be used for the extensive testing of the system modules.

7.2 Framework for vehicle rescheduling

Disruption management and vehicle rescheduling were studied in Chapter 5, where we proposed both a mathematical model and heuristic solution methods for the multi-depot vehicle rescheduling problem. We also introduced the idea of dynamic vehicle rescheduling, which evaluates the efficiency of disruption management methods over the horizon of a planning unit (usually a day) instead of considering every disruption as a separate event.

In this section, we present a solution framework for the rescheduling problem in public transportation, which could be used by companies for aiding their operators responsible for managing disruptions. When a disruption happens during the execution of a daily schedule, it should be resolved as quickly as possible: the operator has to identify the exact cause of the disruption, evaluate possible scenarios for it, then share the best solution with the involved parties (mainly the drivers) so that the order of transportation could be restored. To help with such a problem solving process, a decision support framework should be able to provide multiple different suggestions to the operators, who can then make their decision based on these.

Optimization frameworks are already present for the long-term planning of the different phases of public transportation systems (for some examples, see [15, 77, 81, 100]), but these are mostly built around given solution approaches. To our knowledge, the only paper that deals with such a system for bus rescheduling is the one by Li et al. [72], where they use the operational planning processes for the waste collection of a city in Brazil as a case-study.

An important requirement of this framework is that it should be able to handle multiple solution approaches, if necessary, and should function the same way regardless of the applied solution method. As the problem has to be solved in real time, another main requirement for the solution approaches is that they have to provide suggestions quickly.

Although there are published mathematical models for the problem, these cannot be solved quickly enough to provide the desired result. This leads to the design of efficient heuristic methods, which are able to provide solutions in a short time. The system we present in this section is designed to work with any kind of solution method, and gives results for the problem based on needs of the operators, which they can control through different parameters.

In the following sections, we first give a quick overview of a disruption scenario, then provide the different criteria and parameters that should be considered when solving a disruption. Finally, we propose a flexible framework that is able to provide multiple suggestion for the problem in a short time. We published these results in [12].

7.2.1 Disruptions in transportation

As an exhaustive overview and problem definition was already given for disruption management in Chapter 5, we only present the most important concepts for a single disruption scenario.

The daily schedule of a transportation company consists of several blocks that have to be serviced, each block being a series of tasks that have to be carried out in the given order. Every block has a vehicle assigned to it, and at least one corresponding driver that executes its tasks. The most common tasks are the ones corresponding to the trips of the timetable, and the so-called deadhead trips, which are used by the drivers to travel with an empty vehicle from one location to another. Blocks can also include several different vehicle specific tasks (eg. parking, refueling), while driver shifts have driver specific tasks (eg. breaks, administration).

When a disruption happens during the execution of a daily schedule, usually one of the vehicles in service becomes unavailable for a period of time. This leads to the pre-planned schedule becoming infeasible, as one or more of the tasks are not executed by a vehicle anymore. Such trips are addressed from now on as *disrupted trips*. A vehicle schedule becomes infeasible if it contains such trips. Companies usually have a backup vehicle and driver ready, which are dedicated to such situation, but this solution might not be the best one. Our aim is to propose different suggestions that do not utilize this backup vehicle, and are more effective than this option.

Real-life criteria

In a real-life application, there are several rules and constraints that make the problem more complicated as it was described previously. There are regulations that have to be followed in every case, while the violation of others should be penalized. As we mentioned earlier, a daily schedule includes both vehicle blocks and drivers shifts, thus we classify the most important regulations into two groups. In this section, we give a list of the most important rules for both groups. Note, that these are only the most common regulations, and other ones might arise depending on the country or the transportation company.

- **Vehicle regulations**
  - *Vehicle depot and trip compatibility*: Trips can only be serviced by vehicles from a fixed set of depots, and this should be respected when creating a new schedule to manage the disruption.
  - *Vehicle type and trip characteristics*: Similarly to depot compatibility, some trips can only be executed by vehicles of a certain type. For example, trips that run between different cities, or cover a longer distance, must have a bus with special equipment (eg. air conditioning).
- **Driver regulations**
  - *Maximum driving time*: Each driver has a maximum daily driving time, which they cannot exceed.
  - *Driver breaks*: After given time periods, mandatory breaks have to be assigned to the drivers. Moreover, these breaks can only be carried out at specific geographical locations.
  - *Maximum working time*: Similarly to driving time, the maximum daily working time of drivers is also limited. This is not equal to the driving time, as driver shifts have other events as well, which do not require a vehicle (eg. administration).

Besides these regulations, permitting different structural modifications should also be considered in the schedule. For an illustrative example for these, refer to Figure 7.2.

The figure shows a disrupted daily schedule, where the disruption is represented by a vertical line. There is one disrupted trip t_0, that has no available vehicle anymore, and should be executed by one of the B_j blocks still in service. Depending on the active regulations, we can give several solutions for the problem. Here are a few examples:

- If trip t_0 is compatible with block B_1, and there is enough time to insert it to the available gap (together with any necessary deadhead trips), then we can solve the problem.
- If we want to insert t_0 to block B_2, we have to remove trip t_2. There are several different places where we can insert this newly removed trip. It might fit into the gaps in blocks B_1 or B_5 (if compatible, and there is enough time for deadheads). We might also be able to insert it to blocks B_3 or B_4. However this would mean removing trip $t_{3,2}$ or t_4 respectively, and finding a new duty for them.

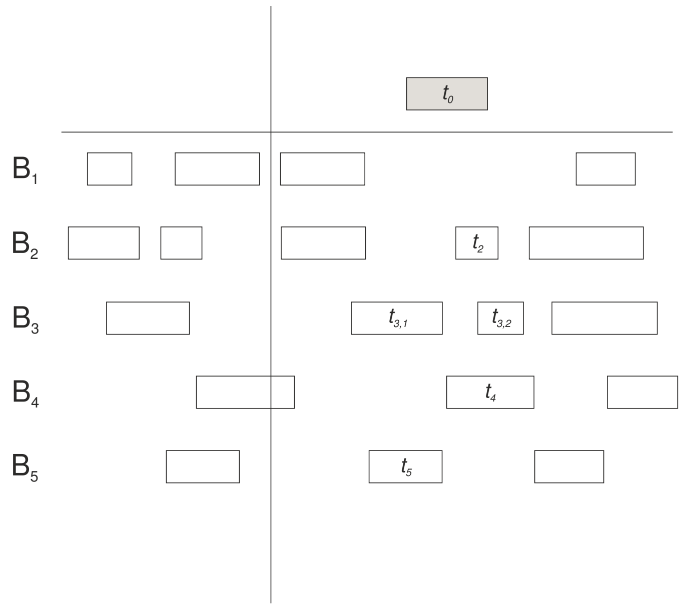


Figure 7.2: Example for possible modifications as a result of a disruption

- There are several scenarios similar to the above, where trips are removed from a block, and inserted into another one.
- Lateness—if allowed— can also be introduced: delay trip t_4 in blocks B_4. This can give us a big enough gap to insert our trip t_0 (if it is allowed by compatibility and deadheads).

As it can be seen from these examples, there are some other constraints that we have not discussed with the above requirements. These are connected to the original blocks and tasks (eg. modifying departure/arrival time, or removing a trip from its original block). A solution method for this problem has to know the penalties for violating each requirement, and it also needs to know the hardness of their constraints. This information can be given easily with the use of parameters.

Parameters

In this section, we introduce parameters of our framework based on the different rules and constraints presented in the previous sections. These parameters have to be considered by any algorithm solving the bus rescheduling problem:

- A binary parameter that allows the violation of depot-compatibilities. A penalty parameter for each violating task is also needed.
- A binary parameter that allows the violation of vehicle type correspondence. A penalty parameter also has to be introduced for each violating task.

- A binary parameter that allows the introduction of lateness. A penalty parameter also has to be introduced per 1 unit (minute) of lateness.
- A binary parameter that allows the movement of trips between feasible blocks. A penalty parameter is also needed that gives the cost of each such move.
- A binary parameter that allows the cancellation of trips. A penalty also has to be introduced for every canceled trip.
- An integer parameter that limits the maximum amount of lateness which can introduced by the algorithms to the schedule.
- An integer parameter that limits the maximum amount of lateness an algorithm can introduce to a single event.
- An integer parameter that limits the maximum amount of lateness an algorithm can introduce to one block.
- An integer parameter that gives the maximum number of feasible blocks that can be modified.
- An integer parameter that gives the maximum number of trips that can be canceled.
- A parameter that limits the maximum total length of the newly introduced deadhead trips.
- A parameter, which gives the latest point in time, by the end of which the algorithms should not modify any more feasible schedules.
- A parameter on the number of suggestions (feasible solutions) given by the framework.
- A parameter on the maximum running time.

As it can be seen in the list of proposed parameters, there are none that correspond to driver rules. Driver regulations are very strict, and most of them are defined by the European Union, or the country. Thus, they cannot be violated by any means. Depot compatibility and vehicle type correspondence might also be strict in the case of certain companies, but we decided to let the operator of the system decide about their violation.

7.2.2 The Solution Framework

In the previous sections, we presented our basic ideas behind the methodology for the problem. We described the need for a framework that does not depend on the solution algorithm it executes, and can be controlled through a list of different parameters. In Figure 7.3, we give a layout of the different parts of the system.

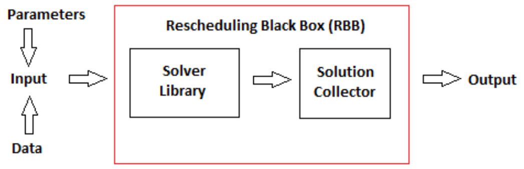


Figure 7.3: The structure of the rescheduling framework

The input for the system consists of two parts: the list of parameters described in the previous section, and the problem specific data tables. The type and structure of these tables of course can vary between different implementations of such a system, but it is important that it contains the disrupted daily schedule, the set disrupted trips, and vehicles that are unavailable due to the disruption. For the remainder of this chapter, we will refer to a 3-tuple of {*disrupted schedule*, *set of disrupted trips*, *unavailable vehicle information*} as a *configuration*.

The input determines the starting configuration of our problem, which is a schedule that contains the still feasible blocks, a set containing the disrupted trips that are not executed currently by any vehicle, and unavailability information for the vehicles in service. A configuration is supposed to be feasible, if all blocks in its schedule are feasible and do not violate any regulations or parameters. If the set of disrupted trips is not empty, then those are considered canceled.

Once the input processed, it is then transferred to the main module of the system, which we call the Rescheduling Black Box (RBB). This is the part that carries out the solution process, and consists of two parts:

- The *Solver Library (SL)* manages the different solution methods that are built into the system. The system can have any number of implemented solution methods, and the desired method can be invoked by the use of additional parameters. If there is a possibility to parallelize solution processes, multiple methods can also be executed at once.

- Any feasible solution produced by the algorithms is sent to the *Solution Collector (SC)*. This sub-module is responsible for managing feasible solutions. If more solution methods are running in parallel, all of them post their results to the SC simultaneously. The SC then filters any duplicate solutions, and also gives an ordering of the remaining ones based their cost.

Once all solution algorithms finish their execution, or the maximum running time is reached, the SC returns the desired number of feasible solutions. This value can also be set by the operator as a parameter (number of suggestions). The order in which these results are filtered can also be determined by the operator (eg. rank them based on their costs, or ask for ones with no lateness, if possible). The operator receives the output data, and can use the presented suggestions to decide on how to solve the arising disruption.

7.2.3 A real-life application

As we mentioned earlier, solution for the rescheduling problem has to be adequately quick, as the order of transportation has to be restored as soon as possible. Because of this, the implemented heuristic methods must have a running time of at most a couple of minutes to be acceptable.

Also, because of the flexibility of the SC module, it is useful to provide solution methods that find multiple feasible solutions during their runtime. This gives the operator more suggestion options to choose from. A simple approach that we implemented in our system is a *naive search*, which basically finds all the possible trivial insertions in the starting configuration (if any), and sends these to the solution collector. If there are any trivial insertions, it is highly likely that it will be the cheapest solution with regards to any of the parameters.

In Section 5.2, we proposed two heuristic approaches that have also been implemented to the system. We applied both the recursive and the tabu search heuristic simultaneously with the above introduced naive method to test our framework. As both methods explore a part of the problem space, they encounter a large number of feasible solutions. These could all be sent to the SC module even while the algorithms are running. However, we introduced a simple modification to these methods: while still registering the best feasible solution they have found so far, the algorithms also store a bound on the cost of solutions that should be sent to the SC. If they produce a better quality result than this cost, then the corresponding schedule is forwarded to the SC, but it is discarded otherwise. The SC filters every solution with regards to driver rules, and discards the infeasible ones.

Test results of the framework in real life have been promising. Every instance we experimented with had a short running time (they all finished under 1 minute even for bigger instances), and was able to produce multiple good quality suggestions based on the required parameters. It can also be seen that the proposed framework allows the integration of any number of different algorithms, as long as they take the required parameters into consideration. This provides a great flexibility for the system, as operators can even choose which algorithms they want to utilize depending on the type of the disruption.

7.3 Visualization and validation using timed automata

In this section, we propose a solution method using Timed Automata for the schedule assignment problem introduced in Chapter 6. This approach was chosen as it presents a clear system of the different actors, where the process of the solution can be tracked from the beginning to the end, and the different parts and actors of the problem are clearly separated from each other. Various types of automata have been applied successfully for the modeling of event-driven systems with discrete state space when timing is not an essential aspect of the behavior of the system. The most common extension of the simple finite automaton model with timing is the timed automaton with clocks, guards and invariants. With this model, each individual part of the system can be modeled independently, and strict timing constraints can be enforced by guards on the event transitions and invariants at the states. These individual models can be compiled into a single coherent model of the whole system by the parallel composition operator, and later used for various analysis purposes. The timed automata model has several advantages from the practical point of view. First and foremost, if done correctly, individual automata are straightforward models of the real practical constraints. This reduces the chance of making modeling mistakes, and also promotes re-usability. Moreover, the model can be used to answer various practically important questions, such as 'can the system run into a deadlock?', 'would there be a feasible solution to the problem, if a certain parameter would change its value?', 'if a certain event occurs, does it ensure some properties?', etc. Last, but not least, if optimization is a goal, the extension to Timed Automata can tackle a wide range of optimization problems, and provide the optimal sequence of events to reach a state with certain conditions.

7.3.1 Timed automata

Timed Automata [5, 4] give a general purpose model for discrete event systems with timing constraints. In this model, the automata are accompanied by several clocks, and the transitions can be guarded by timing constraints on these clocks. Transitions may also change the state of the clocks, and states may have similar guards. With these alterations, timed automata became a very expressive and powerful tool to model various systems. This model was further extended in [13] to include linear pricing on the states for optimization purposes. In this extended model, the price of a timed event sequence is increased during delay transition in certain states with a given coefficient. The event sequence with minimal price can be found with a branch-and-bound style state enumeration algorithm using Difference Bound Matrices [45]. Linearly Priced Time Automata have been successfully applied for scheduling problems of batch processes, [83, 97].

As Timed Automata are not as commonly used as e.g. Mixed Integer Linear Programming, the number of available software tools for this type of models is modest. Uppaal [17] is the result of the collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark. Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). The Uppaal-Cora project extends the capabilities of the software for finding the optimal event sequences for Linearly Priced Timed Automata[14].

7.3.2 A TA approach for the schedule assignment problem

The schedule assignment problem has been modeled in the Uppaal Cora software using timed automata. In this software, the model consists of the following parts:

Global Declarations

This is a simple text input with a C-like syntax, where the channels, variables, custom types, and functions are declared, that are in the global scope, i.e., reachable for all automata.

Templates

Templates can be considered as classes of automata, that can be instantiated with different argument values. The templates describe the states, transitions, guards, etc. for a certain type of automaton, and may also include declarations with a private scope.

System definition

The last part is the system definition, where the system is defined as a parallel composition of several instantiated automata based on the available templates.

We will present the most crucial parts of the proposed model in the following sections.

Declarations

Custom defined functions and data types can ease the formulation, and propagate the transparency of the model. However, the most significant part of the declaration is the set of defined global clocks, channels and global variables.

In the proposed system there is only one global clock defined, that represents the time passed since midnight on the current day:

```
clock time;
```

There is a single broadcast channel, that indicates the event of moving from one day to the next at midnight.

```
broadcast chan daypass;
```

For each job-type, there is an event representing its start, and another one its finish.

```
chan start_job[job_id];
chan finish_job[job_id];
```

For each garage, there is an event of letting a bus in, and out through the gate.

```
chan go_garage[garage_id];
chan leave_garage[garage_id];
```

A simple integer variables will contain the number of the jobs of a day not yet assigned to any bus, and the number of buses outside of a garage.

```
int[0,jobcount] jobsToAssign = 0;
int [0,buscount] busoutside=0;
```

The variable `currentday` always holds the value of the current day starting from 0. It is initialized to -1 to ease the modeling of the first day.

```
int[-1,daycount] currentday = -1;
```

Templates

There are 4 templates defined in the system: `Bus, Garage, Job, and Controller`. The first three of these describe different entities of the schedule assignment problem: the vehicles, the garages, and the daily blocks respectively.

The `Controller` template controls the daily behavior of the system, see Figure 7.4. Only one instance of this template is created in the System definition, and the starting state of the automaton is a so-called committed state, thus the event leading from `Start` to `InProgress` is guaranteed to be fired immediately, which calls a function initializing the global variables. During the day the automaton stays in the `InProgress` state, where a `time` $\leq$ `1440` invariant ensures that a day cannot last longer than 1440 minutes. When the system is ready to move to the next day, the loop event is fired. It has several guards to ensure that the system has not reached the last of the days, and all of the buses are in a garage. Upon this event, the automaton sets some global variables to their supposed values for the next day. When the last modeled day has ended, the automaton moves to its `Finish` state.

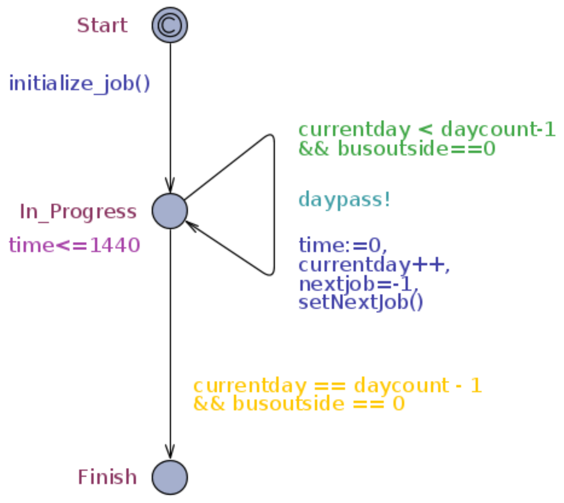


Figure 7.4: The Controller template of the timed automata model

The `Job` template is the basis for each block-type defined in the input. The same automaton is used for the same type of blocks appearing on different days. The template has 3 states: `Waiting, InProgress, and Finished_or_NotToday` (Fig. 7.5). An instance of this template is initially in the last state, and moved to the first one at the start of a new day, if the job is present on that day. Then, when the exact time of the Job comes, the automaton moves to its middle state, and finally to the last one, when it is supposed to be finished. These two events synchronize via the `job_start` and `job_finish` channels with the automaton of the bus assigned to it. The invariants on the first two nodes do not affect the soundness of the model, however, they reduce the search space of the algorithm significantly by removing branches that are unable to end up at a desired state.

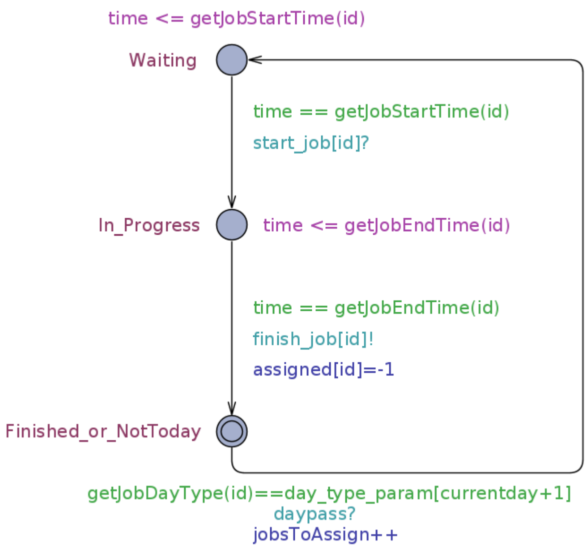

time <= getJobStartTime(id)
Waiting
time == getJobStartTime(id)
start_job[id]?
In_Progress
time <= getJobEndTime(id)
time == getJobEndTime(id)
finish_job[id]!
assigned[id]=-1
Finished_or_NotToday
getJobDayType(id)==day_type_param[currentday+1]
daypass?
jobsToAssign++

Figure 7.5: The Job template of the timed automata model

The `Garage` template is the simplest one with a single state having a loop for the `go_garage, leave_garage` channels with guards, ensuring that it is not occupied by more buses than its capacity.

Understandably, the template for the `Buses` is the most complicated. A simplified version of the template is shown in Figure 7.6, where many guards, updates and invariants are left out to support the figure's transparency. Initially each bus automaton is in the `Garage_evening` state, representing a bus that has parked in a garage for the night. When the day is over, all of the buses move to the `Garage_morning` state, from which 3 options are possible: a) simply going back to the previous state, i.e., the bus is not used during that day,, b) enqueing for inspection, or c) getting assigned to the job. When the buses leave the garage for inspection or a job, they synchronize with the garage automata through the `leave_garage` channels, and the same happens upon getting back to a garage at the end of the day with the `go_garage` channels. Similarly, the automaton synchronizes with the automaton of the assigned job via the `start_job, finish_job` channels. Note that the automaton for the bus does not have guards for these events, as the source or reason of this constraint is not the bus (e.g. its speed), but the timetable encoded in the job itself. This illustrates how the constraints of the problem are modeled together with the entities where they belong, and that every component is a closed and compact entity by itself.

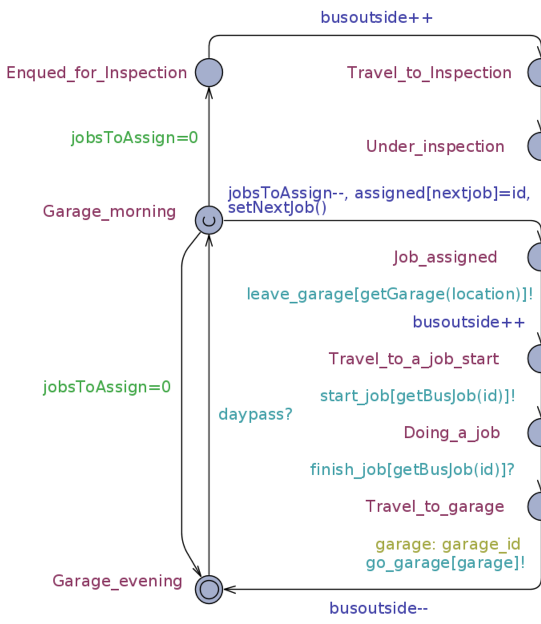


Figure 7.6: The Bus template of the timed automata model

The `Bus` template also has two local variables: one for the number of available maximal workdays before inspection and the other for the current location of the bus. The constraints for that variable are omitted from this template to serve visibility.

System definition and queries

The system definition is simple, it consists of the templates mentioned before, from which multiple ones are instantiated except for the `Controller`.

An important feature of modeling with an automaton is to be able to express various questions towards the engine, e.g.:

`E<> deadlock` - can the system get into a deadlock?

`E<> currentday==3 && Bus(5).Under_Inspection` - is there a feasible schedule (until the third day), where the fifth bus can go to an inspection on the third day?

`currentday == 4 && Bus(2).Garage_evening && time==1 -> deadlock` - if the second bus doesn't leave the garage on the fourth day, does it imply a deadlock?

Questions like the above can be answered via the engine. If the states are extended with pricing data, Uppaal-Cora can also provide the event sequence with the smallest price.

Efficient modeling solutions

The efficiency of this approach can be improved a lot by restricting the search space that is explored by the B&B optimization algorithm. Similarly to other techniques, e.g. MILP formulations, this can be done to some extent by changing the mathematical model only. The general idea is to reduce the state space of the model by excluding feasible solutions without losing the optimal one.

In case of TA models, there are some modeling pitfalls that, although they do not violate the soundness of the model, can increase the state space unnecessarily. Some of these issues are discussed here along with their resolution.

Exploring branches that lead to deadlocks One requirement for a sound TA model is that all feasible runs in the model should be feasible for the original problem as well. If there are branches in the state space exploration that do not yield any feasible solutions, the model can still be considered valid. The evaluation of these branches, however, may require a lot of computation. Unfortunately, this issue can be present in a model rather unperceived. A simple example would be the `Job` template without the invariants. The timing constraints on the transitions enforce that the automaton should stay in its `In_Progress` state only between the start and finish time of that particular job. Without including the invariant on the state, however, the algorithm also explores those runs where an automaton stays longer in this state, while other buses in the system may perform other jobs. Depending on the number of jobs for the given day, this could result in a relatively large tree, before the algorithm reaches a state on all branches, where the `busoutside` variable is not 0, but the bus in question can not leave its `Doing_a_job` state. Even if other parts of the model are excluded, the algorithm would explore 3^n runs (where n is now the number of jobs on a day), from which only a single one is feasible.

Permutation of independent transitions If there are n transitions that can be executed in any order to achieve the same state, the algorithm will visit the same state $n!$ times. Similarly to the previous issue, it is easy to include such parallelism into the model. Two different examples can be observed in the `Bus` template. First, let us assume, that the `Garage_morning` state is not set as urgent. From this state, the automaton can move to three different directions: an inspection, a job, or stay in the garage. If this state was not urgent, then the same decisions could be made in a lot of different orders; not to mention the runs, where a bus would already move on to its traveling state while the decisions for some of the other buses are not yet made. By setting the `Garage_morning` state urgent, all of these decisions are forced to take place at the same time. Another parallelism is avoided by the use of the `nextjob` variable. At the beginning of each day, this variable is set to -1, and the `setNextJob()` function is called to give back the `id` of the next active job in the list. If this was modeled with a select statement on the transition, then the same assignment of the buses could be given in any order. This way, the assignments are only made in the order of the jobs.

Large state size If possible, the size of a single state should be minimized. This can be achieved via various modeling techniques. If some states can be made not just urgent but committed, that can have a huge effect on avoiding parallel executions. Moreover, states that do not serve an important goal could be avoided for the same purpose. Such examples in our case are the multiple states for inspection in the `Bus` template. They could be merged into a single state that does not include the different phases of the inspection process. This can be done because this level of detail is not required in the case of our problem definition. However, it might be needed in the future, or under different modeling constraints. Additionally, if possible, variables should have bounded ranges.

Some modeling decisions can serve more goals from the aforementioned. As an example, in the `Bus` template, the buses are allowed to go for an inspection or stay in the garage only if all the jobs have a bus assigned to them already. This has a double effect: (i) infeasible branches, where too many buses are sent to inspection or stay in the garage (and there is not enough to perform the jobs) are not generated (ii) the job assignment decisions can not be made in parallel with the decisions for inspections or garages.

Note, that the presented templates are not fully optimized in this way, their purpose is to provide a frame for describing the system. This model can still be improved by, e.g., removing the parallelism between the inspection decisions, or to model the garages as integer arrays.

7.3.3 Proposed applications of the model

As it was mentioned at the beginning of this chapter, solving real-life problems with this model is unfortunately not possible due to their large problem size. Currently, the model only enables results for small test instances over a couple of days, with less than ten blocks during a day. However, as it was mentioned in Section 7.3.2, the model is able to answer certain queries regarding problem state, and thus validate theoretical scenarios.

Moreover, due to the underlying automata, the solution process of a scenario can be followed step-by-step. We wanted to present an alternative model where the emphasis is on the clarity and visualization of the solution process. While transport scheduling problems usually have a complex structure, this method might provide a possible way for experts of a company to interpret and analyze any feasible solution step-by-step.

7.4 Generating input instances

Testing the efficiency of algorithms developed for real-life problems is not always simple. While well maintained, state-of-the art data repositories exist for theoretical problems like bin packing [47] or the traveling salesperson problem [30, 90], this is not the case in transportation scheduling. The field of public transportation is a highly competitive industry, and companies have no incentive to share data about their internal processes.

Although we were provided with the real-life instances of the transportation company of Szeged, that only gave us a limited dataset to work with. As it is difficult to access the real-life instances of transportation companies, the easiest way to acquire more test data that has the properties of the real-life input is to generate it based on our needs. Many papers from literature that study vehicle scheduling present the efficiency of their methods on random instances generated by the algorithm of Carpaneto et al. [26]. A benchmark dataset using this generation method also was published online by Huisman [60], and several papers use this to evaluate their methods.

However, our test experience shows that the structure of the data generated by this method was different from the real-life instances of Szeged (a middle-sized city in Hungary) in many aspects. Because of this, we proposed an improved way of generating random data in [34], which we present in this section. This improvement comes both with adjustments to the original method, and with the inclusion of extra features that were not considered by Carpaneto et al. First, we describe the original method from [26], and then give our new variation for it.

The only other random instance generation method that we know of was published recently by Guedes and Borenstein [53], who aim to produced random input for the MVTSP.

7.4.1 State of the art

In this section, we present the random instance generation method proposed by Carpanet et al. in [26]. The input of the algorithm is the number n of trips, and the number m of depots. The number f of geographical locations is uniformly chosen from the interval $[\frac{n}{3}, \frac{n}{2}]$, their locations are chosen in a uniform random way on a $60 * 60$ grid. The deadhead trips between geographical locations p and q correspond to their $d(p, q)$ Euclidean distance.

The properties of every t_i trips is determined based on the above generated information. The starting and ending $sl(t)$ and $el(t)$ geographical locations are chosen uniformly from $[1, f]$. The $d(sl(t), el(t))$ length of the trip is also determined based on these locations. Trips can have two types: short trip, or long trip.

There is a 40% chance that a t trip becomes a short trip. Its $dt(t)$ departure time is chosen randomly:

- with a 15% chance uniformly from [420,480],
- with a 70% chance uniformly from [480,1020],
- and with a 15% chance uniformly from [1020,1080].

The $at(t)$ arrival time of a short trip is chosen uniformly from the interval $[dt(t) + d(sl(t), el(t)) + 5, dt(t) + d(sl(t), el(t)) + 40]$.

Long trips are generated with a 60% chance. Their $dt(t)$ departure time is chosen uniformly from $[300, 1200]$, while their $at(t)$ arrival time is chosen uniformly from $[dt(t)+180, dt(t)+300]$. Long trips have the same starting and ending location, which means that a value is assigned to $sl(t) = el(t)$ uniformly from $[1, f]$.

They also presented a possible placement of the depots for $m = 2, 3$. The number of vehicles in each depot is determined uniformly from $[3 + \frac{n}{3m}, 3 + \frac{n}{2m}]$.

7.4.2 Our approach

Our experience showed that the instances generated using the above method were very differently structured from the real-life data we were dealing with. We decided to modify this method to produce instances that are closer the ones that were provided by the transportation company. We did two major changes to the method: extended its definition of depots, and modified the structure of the generated trips.

Introducing vehicle type to the depots

While Carpaneto et al. uses the concept of depots as different starting locations for the vehicles, their vehicle fleet is homogeneous, and all trips can be serviced by any vehicle. Our real-life data uses depots as a combination of vehicle type and starting location, and because of this, we needed input that includes a heterogeneous fleet. Another requirement we wanted to model is that trips should not be executable by any vehicle, rather only the ones that have the characteristics needed for the trips. These are given by the vehicle types, which are included in our definition of depots.

We introduced an additional input type: a p_j probability for every $1 \leq j \leq m$ depot. The p_j value gives the probability that a trip can be executed from depot j. When the trips are generated, they are assigned a boolean $\mathbf{v} = (v_1, ..., v_m)$ depot-compatibility vector. For every v_j,

$$v_j = \begin{cases} \text{true} & \text{with } p_j \text{ probability} \\ \text{false} & \text{otherwise} \end{cases}$$

If all components of the $\mathbf{v}$ receive false values, then exactly one of them is set as true. This is also decided using the given probabilities. A trip can be executed only from those depots, whose corresponding components have a true value.

A more structured approach to vehicle types

We also developed a slightly different method for depot-compatibility generation, which is closer to the practice of transportation companies. Compatibility vc can be defined between certain vehicle types: vehicle v is compatible with v' (denoted by $vc(v, v')$), if v is able to service any trip that v' can. Note, that this relation is not symmetric, and does not automatically imply that v' is also compatible with v. For example, a solo bus that is wheelchair accessible could substitute any normal solo bus without any problems, but the same would not be true the other way around.

To generate vehicle depots this way, we also need the additional input probabilities p_j for every $1 \leq j \leq m$ depot, where The p_j gives the probability that a trip can be executed from depot i. A major difference is that now this input also has to satisfy $\sum_{j=1}^{m} p_j = 1$. All vc compatibilities between the m depots have to be given as well. This way, the $\mathbf{v} = (v_1, ..., v_m)$ boolean depot-compatibility vector created for every trip will only have a single true component at first, chosen according to the p_j values. After this component v_k is determined, all other $v_l \neq v_k$ components are also assigned values the following way:

$$v_l = \begin{cases} \text{true} & \text{if } v_k \text{ is compatible with } v_l \\ \text{false} & \text{otherwise} \end{cases}$$

Modifying the length of the trips

Analyzing the trips of the original generator, we found that the average length of the trips was too high compared to our real-life data. Also, trips were scattered geographically: the number of generated geographical locations was very high compared to the number of trips, and two trips rarely followed each other at the same location in a small time-frame. Because of this, the input lacked any kind of regularity in its structure.

To address this, we introduced some further changes. After experimenting, we found the $[\frac{2n}{25}, \frac{3n}{25}]$ interval that gives an acceptable number of locations. However, because of the decreased number of geographical locations, we also had to decrease the area they are generated at. We used a $30 * 30$ grid for this.

To address the problem of the too long average trip length, we slightly modified the generation of the trips as well. The ratio of the long and short trips has been exchanged, and we generated short trips with a 60% chance, and long trips with only a 40% chance.

The length of the trips has been decreased. The $at(t)$ arrival time of a short trip is chosen uniformly from the interval $[dt(t) + d(sl(t), el(t)), dt(t) + d(sl(t), el(t)) + 20]$, while the $at(t)$ arrival time of a long trip is chosen uniformly from $[dt(t) + 40, dt(t) + 60]$.

Using the modification above, the random generated instances we received resembled more closely to the real-life data we were provided by the transportation company of Szeged.

Generating details for refueling

While the above presented method will provide all the input for an MDVSP, other data might also be needed for different problem types. We discussed the VSAP-VS problem in Chapter 4, where vehicle-specific activities are also considered together with vehicle scheduling. We presented this model on random instances that also included refueling activities. For this, fuel type also has to be assigned to vehicles, and refueling station have to be generated.

The additional input for this is the number of refueling stations, number of fuel pumps at a station, the f number of fuel types, an q_j probability that a vehicle belongs to fuel type j, where $\sum_{j=1}^{m} q_j = 1$. The maximum allowed distance and refueling time is also given for every type.

For every depot $1 \leq d \leq m$, vehicles of d are all assigned a single fuel type according to the probability q_j. Refueling stations are chosen uniformly from the set of not utilized geographical locations (ones that contain neither depots nor trip locations). Fuel types are also determined for every pump at the station, according to the same q_j probabilities that were used for the vehicles.

7.5 Summary and remarks

In this section, we presented a collection of different concepts that are rarely studied, yet they can be integrated into decision support systems used by public transportation companies, or used at the development of such a system.

As we showed in Chapter 5, disruption management is an important part of the everyday operations of a transportation company. We proposed a solution framework for this problem in Section 7.2, that is capable of handling multiple solution approaches in parallel, analyzing their results, and providing multiple different suggestions to the operators of the company. The system has a high number of different parameters that the operator can modify depending on the type of disruption, or the structure of the required solution.

In Section 7.3, we proposed a new modeling technique for the schedule assignment problem using timed automata. This approach was chosen because we wanted to develop a model for a transportation scheduling problem where the solution process is easy to follow, and the structure represents every important aspect of the input scenario. While the problem size ultimately turned out to be too large for this purpose, the resulting system can be efficiently applied to visualize the structure of the problem, and validate certain queries about the input.

Section 7.4 presented a random input generation method for the MDVSP and the VSAP-VS problems. Methods that are developed for public transportation systems have to be tested extensively, but proper benchmark data is unfortunately not available freely. We developed this method as an alternative to the existing state-of-the-art approach by Carpaneto et al., as we felt that the instances generated by their algorithm were not close enough to the real-life instances we used for testing our methods. We expanded the concept of depots of their algorithm by also considering vehicle types, and gave two generation options for these. Trips are also generated in a different way, and we gave and extension for the inclusion of refueling activities.

Chapter 8

Conclusions and future work

In this dissertation, we studied optimization problems of vehicles in the fleet of a transportation company. We examined several different problem types: some were connected to the daily schedules of a company (either by creating them, or restoring their order after disruptions), while others considered a longer planning horizon. An important aspect of all the studied problems was their applicability in real-life scenarios.

The two methods we introduced in Chapter 3 were both heuristic algorithms for the creation of multi-depot vehicle schedules. Section 3.2 presented three different heuristic approaches based on variable fixing, which aimed to reduce the problem size of the MDVSP by finding series of trips that are likely to belong to the same block in the final solution. The methods provided good quality solutions with a short running time, and we also showed that using a practical property like the concept of bus-lines for size reduction is an effective approach for real-life instances. Section 3.3 introduced an iterative algorithm that creates vehicle schedules satisfying the most important constraints of driver shifts. This method can be combined with any solution approach for the MDVSP, and helps the optimization process of a sequential framework by connecting the phases of vehicle and driver scheduling.

In Chapter 4, we presented the integrated vehicle scheduling and assignment problem, which creates vehicles schedules that also include activities connected to the vehicle executing the blocks of the schedule. We gave a set partitioning model for the problem that was solved using a column generation framework, and presented solutions that consider the refueling of vehicles with multiple fuel types as the vehicle-specific activity of the problem.

Chapter 5 studied the area of disruption management in bus transportation. Section 5.2 presented a mathematical model for the multi-depot vehicle rescheduling problem for a single disruption, and gave two fast heuristics for its solution: a recursive and a tabu search method. Section 5.3 introduced the dynamic vehicle rescheduling problem, that proposes a new way of evaluating disruption management methods: instead of treating disruptions as separate problems, the efficiency of rescheduling methods should be evaluated by studying the resulting schedule at the end of the day after the solution of a series of disruption.

We introduced the schedule assignment problem in Chapter 6 for the long-term planning of public transportation. The aim of the problem is to assign vehicles to the pre-planned daily blocks of the company. This assignment also has to satisfy vehicle-specific constraints, like parking at the end of each day, and attending regular mechanical inspections. We gave a state-expanded multi-commodity flow model for the problem, which was then solved using a MIP solver.

Chapter 7 presented three different topics that are loosely connected by the concept of real-life application. Section 7.2 presented a framework for vehicle rescheduling, that can be easily integrated into a decision support system. This framework is able to provide suggestions to the operators of the company, and functions the same way regardless of the implemented solutions methods. In Section 7.3, we modeled the schedule assignment problem using timed automata. This was a new approach to transportation problems, completely different from the classical mathematical models given for them. While solutions of real-life scenarios is not yet possible with this proposed model, it is capable of verifying queries about certain scenarios, and visualizing the structure of the problem. The final Section 7.4 presented a method that generated random input similar to the ones of the city of Szeged. These inputs consider multiple depots and vehicle types, and also include the properties of refueling activities if needed.

While the above presented chapters studied many different practical aspects of problems related to vehicles, there are always possibilities for future research. An important, but not widely studied problem is the simultaneous creation of schedules for days that have the similar underlying timetables. The schedule of such days should not be developed completely independently of each other: days that have a similar underlying timetable should have similarities between their daily blocks as well. To our knowledge, this was only studied by Amberg et al. [6], who give an overview of possible solution methods an a mathematical model for the problem. We presented our preliminary work on this problem in a conference talk [36], where we created similar schedules using the modifications of methods in Sections 3.2 and 5.2. We intend to continue this research, and also formulate a different mathematical model for the problem.

Modeling transportation scheduling problems with timed automata also seems a promising technique. We intend to continue the research presented in Section 7.3, give a more efficient formulation of the presented model, and also present automata-based models for different transportation problems as well.

Finally, we believe that the generation of random input data resembling real-life instances should also be more widely studied. We would like to improve the algorithm presented in Section 7.4 to also include more structural properties of real timetables: for example, the generated trips should belong to bus-lines, and also have a more or less regular frequency.

Chapter 9

Summary

9.1 Summary in English

We presented several optimization problems that are connected to the vehicles in the fleet of a transportation company. These cover a wide range of different problem types, including the creation of daily schedules, long-term planning, real-time management of unforeseen events, and generation of test instances. The topics are not only studied from a theoretical point of view, but their applicability in real life is also considered. The theses of this dissertation are categorized into five different problem groups:

I In Chapter 3, we presented two application-oriented heuristic approaches connected to the MDVSP. We call these methods application-oriented, as their results are not only efficient from a theoretical point of view, but can also be applied in a real-life decision support system because of their structure and quick solution time.

I/1 In Section 3.2, we developed three variable fixing heuristic algorithms for the MDVSP. These methods try to reduce the problem size by finding trips that are likely to be in the same sequence in the final solution, and combine such groups into single, long trips. The resulting smaller problem is then solved using the classical IP modeling approach, and solutions are obtained with a greatly reduced running time. Through extensive testing on real-life and randomly generated input, we showed that while generally good methods might be more appealing from a theoretical point of view, using approaches based on structural properties of real-life instances result in solutions that have both a better performance and structure considering practical applications. This method is studied in our following publications: [32, 34].

I/2 In Section 3.3, we introduced a 'driver-friendly' iterative algorithm that produces vehicle schedules with a structure that also satisfies basic driver scheduling constraints. A classical VSP is solved, and the result is restructured through different steps: first, long blocks are modified through a cut-and-join approach combined with a matching problem to satisfy maximum shift length, then mandatory driver breaks are inserted into them. This process iterates until all trips of the input are part of a feasible a vehicle block. To present the quality of the resulting schedules, we also develop a method that gives a lower bound on the total working time of a schedule based on its timetable of trips. Solutions on real-life instances show that, based on the gap from this lower bound, a significant improvement can be achieved compared to the original schedules of the transportation company. Any method can be applied in the first step of the process for the solution of the VSP, and because of this, good quality solutions can be achieved for the real-life test instances in several minutes. We introduced this 'driver-friendly' method in [9].

II In Chapter 4, we introduced the integrated vehicle scheduling and assignment problem, which aims to give a feasible vehicle schedule that also includes tasks specific to the requirements of the vehicles executing its blocks. For example, vehicles can run out of fuel (and have to be refueled), or have a longer idle periods (where they have to be sent to a parking location). We presented a set partitioning model for problem, which serves as a general framework that can include most vehicle-specific activities and consider their application-oriented constraints. We gave a column generation-based solution method for this model, and showed its efficiency on randomly generated test instances. To showcase the model, refueling was considered for these instances as their vehicle-specific activity, and multiple fuel-types were also studied for the vehicles, which is also rarely examined in other papers. We introduced the integrated vehicle scheduling and assignment problem in [16].

III In Chapter 5, we studied two main concepts in the area of vehicle rescheduling.

III/1 In Section 5.2, we proposed a multi-depot network model for the VRSP. We also designed two heuristic solution methods for the problem, which are both able to provide multiple good quality solutions with a short running time. One is a recursive method that traverses the search tree of the problem, and distributes trips of a single disruption scenario to the available blocks either by simple insertion, or by deleting overlapping trips from these blocks. To avoid the exploration of the large solution space, the depth of the tree is limited using a simple practical observation. The other method is a tabu search algorithm, which starts with an infeasible vehicle block in its schedule, and uses its neighborhood transformations to find a good quality feasible solution. This method is incentivized to remove trips from this infeasible schedule first. Because of their ability to produce multiple good quality solutions in a short time, both algorithms seem suitable for a decision support system that helps the operators of a transportation company by providing them with possible suggestions for the solution of arising disruptions. Our model and heuristics for the VRSP were published in [35].

III/2 In Section 5.3, we introduced the dynamic vehicle rescheduling problem. While the classical method of resolving disruptions with the VRSP focuses on solving a single disruption to optimality, the DVRSP aims for a good quality solution at the end of the day after managing a series of disruptions. Because the problem itself is dynamic, and the input (the list of disruptions) arrives in an online manner, we also presented the concept of the quasi-static DVRSP, which provides an 'offline' version of the problem where all the disruptions are known in advance. The quality of a solution for the DVRSP can be measured using this model. We applied both heuristic methods presented for the VRSP to solve a series of disruptions for pre-planned daily schedules, and showed that they give good quality solutions in a short time for the problem. The DVRSP was proposed in [37].

IV In Chapter 6, we introduced the schedule assignment problem, which assigns the pre-planned daily vehicle blocks of a planning horizon to buses in the fleet of a transportation company. As the problem considers a long-term plan for several days or weeks, activities connected to the vehicles such as daily parking and regular preventive maintenance have to be taken into account. We gave a state-expanded multi-commodity flow network for this problem, which was then solved by a MIP solver. The efficiency of the model was presented on both real-life and randomly generated instances. We presented a simple assignment model for the problem with parking in [33], and proposed the state expanded model that also considers maintenance in [38].

V In Chapter 7, we studied three different concepts that can be integrated into the optimization system of a transportation company. These can be used for real-time control and for evaluating certain modules of the system.

V/1 In Section 7.2, We proposed a decision support framework for vehicle rescheduling. This framework can apply multiple solution methods in parallel and present several suggestions to the operators of a company in a short time. The system also has a high number of different parameters that can be modified depending on the type of disruption, or the structure of the desired solution. We proposed this system in [12].

V/2 In Section 7.3, we gave a novel modeling approach for the schedule assignment problem using timed automata. We chose this modeling technique for the problem because its solution process is easy to follow, and its structure represents every important aspect of the input scenario. The resulting system can be efficiently applied to visualize the structure of the problem, and validate certain queries about the input. This idea was introduced in [39].

V/3 Finally, in Section 7.4, we presented a random instance generation method that can be used to create input with a structure similar to real-life problems. We developed this method as an alternative to the existing state-of-the-art approach by Carpaneto et al. [26], as we felt that the instances generated by their algorithm were not similar enough to the real-life instances of the city of Szeged. Our approach provides instances with multiple depots and vehicle types, and can also include data for refueling activities. This method was given in [34].

9.2 Magyar nyelvű összefoglaló

A disszertációban több olyan optimalizálási probléma is bemutatásra került, ami egy közlekedési társaság járműveihez kapcsolódik. Típusukat tekintve ezek számos témakört felölelnek: napi ütemezések kialakítása, hosszú távú tervezés, váratlan események valós idejű kezelése, valamint példafeladatok generálása. A témaköröket nem csak elméleti oldalról, hanem a valós alkalmazhatóság szempontjából is vizsgáljuk. A disszertáció téziseit öt különböző problémacsoportba soroltuk:

I A 3. fejezetben két gyakorlatorientált heurisztikus módszert mutatunk be az MDVSP-re. Ezek gyakorlatorientáltsága abból ered, hogy nem csak elméleti szempontból hatékonyak az általuk szolgáltatott megoldások, hanem felépítésük és gyors megoldási idejük miatt gyakorlatban is jól használhatóak.

I/1 A 3.2. Szekcióban három megközelítést adtunk a változófixálásos heurisztikára. Ezek olyan járatsorozatok azonosításával próbálja meg csökkenteni a probléma méretét, amik a végső megoldásban is nagy valószínűséggel egymáshoz fognak tartozni, majd az ilyen járatokat egyetlen, hosszú járatlánccá ragasztják össze. Az így kapott kisebb méretű probléma hatékonyan megoldható klasszikus IP-alapú módszerekkel, de eredményeiket a szokásosnál jelentősen rövidebb futásidővel kapjuk meg. Valós és véletlenszerűen generált bemeneteken végzett alapos teszteléssel megmutattuk, hogy — bár elméleti szempontból az általánosan jól működő megközelítések vonzóbbak lehetnek — a feladat valós jellegzetességeit kihasználó módszerek jól teljesítenek gyakorlati alkalmazhatóság szempontjából. A módszert a következő publikációinkban vizsgáltuk: [32, 34].

I/2 A 3.3. Szekcióban bevezettünk egy "vezetőbarát" iteratív módszert, ami olyan ütemezéseket készít, melyek kielégítik a legfontosabb vezetőkhöz kapcsolódó szabályokat is. A módszer egy hagyományos VSP megoldásával kezdődik, aminek eredményét különböző lépesek alakítják át: először a túl hosszú blokkok kerülnek módosításra egy párosítással összekötött vág-és-ragaszt módszer segítségével úgy, hogy megfeleljenek a maximális műszakhossznak, majd a kötelező szünetek is beszúrásra kerülnek mindegyikbe. Az iteráció addig ismétlődik, amíg minden járat kiosztásra került valamely járműblokkba. Az eredmények kiértékeléséhez egy olyan módszert adtunk, mely egy menetrend járatai alapján megbecsüli a kiszolgálásukhoz szükséges munkaidő alsó korlátját. A korlát segítségével megmutatjuk, hogy valós példákon kapott megoldásaink költségei jobbak a közlekedési társaság eredeti ütemezéseihez képest. Mivel a "vezetőbarát" módszer bármilyen megközelítést használhat a VSP megoldására, így ezek az eredményeink pár perc alatt megkaphatóak. A "vezetőbarát" módszert [9]-ben vezettük be.

II Az 4. Fejezetben bevezettük az integrált járműütemezési és hozzárendelési problémát, ami az ütemezések kialakításakor figyelembe veszi az azokat kiszolgáló járművek igényeit is. Egy jármű például kifogyhat az üzemanyagból (el kell küldeni tankolni), vagy hosszabb időszakon keresztül tétlenül várhat (ilyenkor parkolóhelyre kell irányítani). A feladatra egy halmazparticionálási modellt adtunk, amelybe a legtöbb járműspecifikus tevékenység beilleszthető, és rugalmas keretet nyújt a gyakorlatorientált feltételek beépítésre is. A modellre oszlopgeneráláson alapuló megoldási módszert ismertettünk, melynek hatékonyságát véletlenszerűen generált bemeneteken mutattuk meg. Ezekben a járművek tankolását vesszük jármű-specifikus tevékenységnek, egyszerre több üzemanyagtípust is vizsgálva, ami szintén egy keveset kutatott terület. A fenti problémát [16]-ban vezettük be.

III Az 5. Fejezetben két fő területet vizsgáltunk a jármű-újraütemezés témájában.

III/1 Az 5.2. Szekcióban többdepós jármű-újraütemezési modellt adtunk a VRSP-re, valamint két olyan heurisztikus módszert terveztünk, amik rövid idő alatt több jó minőségű megoldást is képesek szolgáltatni. Az egyik egy rekurzív algoritmus, ami bejárja a probléma által meghatározott keresési fát, és szétosztja a zavar következtében nem kiszolgálható járatokat az ütemezés többi blokkja között (vagy egyszerű beszúrással, vagy a blokkon lévő átfedő járatok törlésével). Mivel egy ilyen keresési fa mérete túl nagy a hatékony bejáráshoz, egy gyakorlati szempontból fontos megfigyelés segítségével csökkentjük annak mélységét. A másik módszerünk egy tabu kereső algoritmus, ami kezdetben egy nem használható járműblokkot épít a zavar miatt nem kiszolgálható járatokból. A módszer különböző szomszédsági transzformációk segítségével keres lehetséges megoldásokat. Annak elkerülése érdekében, hogy ne a problémamentes blokkokat próbálja meg újraütemezni a módszer, jutalmazzuk az olyan lépéseket, amik ezt a kezdetben nem használható blokkot javítják, vagy onnan mozgatnak el járatokat. Mivel mindkét módszer képes arra, hogy rövid idő alatt több megoldási javaslatot is szolgáltasson, ezért ideális jelöltek egy olyan döntéstámogató rendszer számára, ami a társaságok operatív forgalomirányításán dolgozó alkalmazottaknak segít meghozni a döntéseiket a különböző valós időben felmerülő problémák esetén. A VRSP-re adott modellt és heurisztikus módszereket [35]-ben publikáltuk.

III/2 A 5.3. Szekcióban bevezettük a dinamikus jármű-újraütemezési problémát, ami zavarok sorozatának kezelése után vizsgálja a módszerek hatékonyságát a nap végéig végrehajtott tényleges ütemezés alapján. Míg a VRSP célja, hogy egyetlen zavart kezeljen a lehető legnagyobb hatékonysággal, addig a DVRSP a nap végéig ténylegesen végrehajtott ütemezésre vizsgálja azt, hogy az mennyiben tér el a nap kezdeti tervtől a nap során végrehajtott újraütemezések hatására. A probléma dinamikus mivolta miatt, illetve mert a bemenet (a zavarok listája) online módon érkezik, felírjuk a feladat 'kvázi-statikus' változatát is, ami a DVRSP olyan 'offline' megfelelője, ahol minden zavar előre ismert már a nap elején. Ezzel a modellel mérhetőek a DVRSP-re adott módszerek hatékonyságai. A DVRSP problémák megoldását mindkét fent bemutatott heurisztikával megvizsgáltuk, és ezek ebben az esetben is jó eredményeket szolgáltatnak. A DVRSP koncepcióját [37]-ben vezettük be.

IV A 6. Fejezetben bevezettük az időszakra történő jármű-hozzárendelési problémát, ami egy társaság hosszabb időszakra kialakított napi ütemezéseihez rendel járműveket. Ennek a beosztásnak a járművekhez kapcsolódó olyan igényeket is figyelembe kell vennie, mint a parkolás minden nap végén, vagy a rendszeres időközönként elvégzett műszaki karbantartás. A feladatot egy állapotkiterjesztett többtermékes hálózati folyamproblémaként írjuk fel, majd MIP megoldó segítségével oldjuk meg. A modell hatékonyságát valós és véletlenszerű bemeneteken is bemutattuk. A modell egy kezdeti verzióját [33]-ben, annak állapotkiterjesztését [38]-ben adtuk meg, és itt vezettük be a karbantartáshoz kapcsolódó feltételeket is.

V A 7. Fejezetben három különböző témakört vizsgáltunk, amik beilleszthetőek egy társaság közlekedésoptimalizáló rendszerébe, és segítik az operatív forgalomirányítást, valamint a rendszer egyes moduljainak kiértékelését.

V/1 A 7.2. Szekcióban egy döntéstámogató keretrendszert adtunk a jármű-újraütemezéshez. A rendszer egyszerre több megoldási módszert képes kezelni, és több javaslatot ajánl fel a vállalat operátorai számára. Egy ilyen rendszer nagyszámú változtatható paraméterrel kell, hogy rendelkezzen, melyek módosításával mindig az aktuális helyzethez legjobban megfelelő megoldások készíthetőek. Ennek a rendszernek a felépítését [12]-ben ismertettük.

V/2 A 7.3. Szekcióban időzített automaták használatával egy újfajta modellezési módszert javasoltunk a jármű-hozzárendelési problémára. A választott modellezési módszer miatt a megoldás lépései könnyen követhetőek, és a modell felépítése a bemenet minden fontos strukturális tulajdonságát visszaadja. A kapott rendszer hatékonyan alkalmazható a probléma felépítésének vizualizációjára, valamint a bemenettel kapcsolatos különböző kérdések validációjára. A fenti ötletet [39]-ben vezettük be.

V/3 Végezetül a 7.4. Szekcióban egy véletlenszerű bemeneteket generáló módszert ismertettünk, ami olyan példafeladatok készítésére alkalmas, melyek felépítése több szempontból is hasonlít a valós életből vett problémákéra. Módszerünket alternatívaként kínáljuk a szakirodalom leggyakrabban használt megközelítéséhez [26], mivel úgy érezzük, hogy az általa generált feladatok szerkezete több fontos pontban is különbözött Szeged városának valós példáitól. A módszer képes több depót és több járműtípust is kezelni, valamint tankolási események generálására is használható. Ezt a módszer [34]-ben vezettük be.

Bibliography

[1] Jonathan D Adler. *Routing and scheduling of electric and alternative-fuel vehicles.* PhD thesis, Arizona State University, 2014.

[2] Jonathan D Adler and Pitu B Mirchandani. The vehicle scheduling problem for fleets with alternative-fuel vehicles. *Transportation Science*, 51(2):441–456, 2016.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Inc., 1993. ISBN 0-13-617549-X.

[4] Rajeev Alur. Timed Automata. *Theoretical Computer Science*, 126:183–235, 1999.

[5] Rajeev Alur and D.L. Dill. A theory of timed automata. *Theoretical computer science*, 126 (2):183–235, 1994. ISSN 03043975. doi: 10.1.1.51.1093. URL `http://www.sciencedirect.com/science/article/pii/0304397594900108`.

[6] Boris Amberg, Bastian Amberg, and Natalia Kliewer. Approaches for increasing the similarity of resource schedules in public transport. *Procedia - Social and Behavioral Sciences*, 20:836–845, 2011. ISSN 1877-0428. doi: https://doi.org/10.1016/j.sbspro.2011.08.092. URL `http://www.sciencedirect.com/science/article/pii/S1877042811014728`.

[7] Viktor Árgilán, János Balogh, József Békési, Balázs Dávid, Miklós Krész, and Attila Tóth. A flexible system for optimizing public transportation. In *Proceedings of the 8th International Conference on Applied Informatics*, volume 2, pages 181–190, 2010. ISBN 978-963-9894-72-3.

[8] Viktor Árgilán, Csaba Kemény, Gábor Pongrácz, Attila Tóth, and Balázs Dávid. Greedy heuristics for driver scheduling and rostering. In *Poceedings of the 2010 Mini-Conference on Applied Theoretical Computer Science: MATCOS-10*, pages 101–108, 2011. ISBN 978-961-6832-10-6.

[9] Viktor Árgilán, János Balogh, József Békési, Balázs Dávid, Miklós Krész, and Attila Tóth. Driver scheduling based on "driver-friendly" vehicle schedules. In *Operations Research Proceedings 2011*, pages 323–328. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29210-1.

[10] Viktor Árgilán, János Balogh, József Békési, Balázs Dávid, Gábor Galambos, Miklós Krész, and Attila Tóth. Scheduling problems in the operative planning of public transportation. *ALKALMAZOTT MATEMATIKAI LAPOK*, pages 1–40, 2014. ISSN 0133-3399. in Hungarian.

[11] Viktor S. Árgilán, János Balogh, and Attila Tóth. The basic problem of vehicle scheduling can be solved by maximum bipartite matching. In *Proceedings of the 9th International Conference on Applied Informatics*, volume 2, pages 209–218, 2014. doi: doi:10.14794/ICAI.9.2014.2.209.

[12] János Balogh and Balázs Dávid. An algorithmic framework for real-time rescheduling in public bus transportation. In *Proceedings of the 2013 Mini-Conference on Applied Theoretical Computer Science (MATCOS)*, pages 29–33, 2016. ISBN 978-961-6984-21-8.

[13] G Behrmann, A Fehnker, T Hune, K G Larsen, P Pettersson, and J Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01). Genova, Italy, April 2 to 6, 2001. LNCS 2031*, pages 174–188. Uppsala University, Department of Information Technology, 2001.

[14] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Optimal scheduling using priced timed automata. performance evaluation review. *ACM Sigmetric*, 32(4):34–40, 2005.

[15] J. Békési, A. Brodnik, M. Krész, and D. Pas. An integrated framework for bus logistics management: Case studies. *Logistik Management*, 5(1):389–411, 2009.

[16] József Békési, Balázs Dávid, and Miklós Krész. Integrated vehicle scheduling and vehicle assignment. *Acta Cybernetica*, 2017. 18 pages, submitted.

[17] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a tool suite for automatic verification of real–time systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, 1995.

[18] Alan A Bertossi, Paolo Carraresi, and Giorgio Gallo. On some matching problems arising in vehicle scheduling models. *Networks*, 17(1):271–281, 1987.

[19] Lawerence Bodin, Bruce Golden, Arjang Assad, and Mark Ball. Routing and scheduling of vehicles and crews: The state of the art. *Computers and Operations Research*, 10(1):63–212, 1983.

[20] Ralf Borndörfer, Andreas Löbel, and Steffen Weider. A bundle method for integrated multi-depot vehicle and duty scheduling in public transit. *Lecture notes in economics and mathematical systems*, 600:3, 2008.

[21] Ralf Borndörfer, Markus Reuther, Thomas Schlechte, and Steffen Weider. A Hypergraph Model for Railway Vehicle Rotation Planning. In *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, pages 146–155. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.

[22] Ralf Borndörfer, Markus Reuther, Thomas Schlechte, and Steffen Weider. Vehicle rotation planning for intercity railways. In *Proc. Conference on Advanced Systems for Public Transport*, volume 2012, pages 12–11, 2012.

[23] Ralf Borndörfer, Markus Reuther, Thomas Schlechte, Kerstin Waas, and Steffen Weider. Integrated optimization of rolling stock rotations for intercity railways. *Transportation Science*, 50(3):863–877, 2015.

[24] Ralph Borndörfer. Discrete optimization in public transportation. Technical report, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2008.

[25] Stefan Bunte and Natalia Kliewer. An overview on vehicle scheduling models. *Journal of Public Transport*, 1(4):299–317, 2009.

[26] G. Carpaneto, M. Dell'amico, M. Fischetti, and P. Toth. A branch and bound algorithm for the multiple depot vehicle scheduling problem. *Networks*, 19(5):531–548, 1989. ISSN 1097-0037. doi: 10.1002/net.3230190505.

[27] Avishai Avi Ceder. Optimal multi-vehicle type transit timetabling and vehicle scheduling. *Procedia-Social and Behavioral Sciences*, 20:19–30, 2011.

[28] Jens Clausen, Allan Larsen, Jesper Larsen, and Natalia J. Rezanova. Disruption management in the airline industry - concepts, models and methods. Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2005.

[29] Jens Clausen, Allan Larsen, Jesper Larsen, and Natalia J. Rezanova. Disruption management in the airline industry-concepts, models and methods. *Computers & Operations Research*, 37 (5):809–821, 2010. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2009.03.027.

[30] William Cook. TSP test data. University of Waterloo, Canada, `http://www.math.uwaterloo.ca/tsp/data/`, 2009. Accessed: 2018-03-31.

[31] Joachim R. Daduna and José M. Pinto Paixão. Vehicle scheduling for public mass transit — an overview. In *Computer-Aided Transit Scheduling*, pages 76–90. Springer Berlin Heidelberg, 1995. ISBN 978-3-642-57762-8.

[32] Balázs Dávid. Heuristics for the multiple-depot vehicle scheduling problem. In *Poceedings of the 2010 Mini-Conference on Applied Theoretical Computer Science (MATCOS)*, pages 23–28, 2011. ISBN 978-961-6832-10-6.

[33] Balázs Dávid. Schedule assignment for vehicles in inter-city bus transportation over a planning period. In *Middle-European Conference on Applied Theoretical Computer Science (MATCOS 2016): Proceedings of the 19th International Multiconference INFORMATION SOCIETY - IS 2016*, pages 9–12, 2016. ISBN 978-961-264-104-7.

[34] Balázs Dávid and Miklós Krész. Application oriented variable fixing methods for the multiple depot vehicle scheduling problem. *Acta Cybernetica*, 21(1):53–73, 2013. ISSN 0324-721X. doi: 10.14232/actacyb.21.1.2013.5.

[35] Balázs Dávid and Miklós Krész. A model and fast heuristics for the multiple depot bus rescheduling problem. In *PATAT 2014: Proceedings of the 10th International Conference of the Practice and Theory of Automated Timetabling*, pages 128–141, 2014. ISBN 978-0-9929984-0-0.

[36] Balázs Dávid and Miklós Krész. Vehicle scheduling based on the similarity of planning units. In *Conference on Computational Management Science: Pricing, Risk and Optimization in Management Science., Bergamo, Italy*, page 38, 2017.

[37] Balázs Dávid and Miklós Krész. The dynamic vehicle rescheduling problem. *Central European Journal of Operations Research*, 25(4):809–830, 2017. ISSN 1613-9178. doi: 10.1007/s10100-017-0478-7.

[38] Balázs Dávid and Miklós Krész. Multi-depot bus schedule assignment with parking and maintenance constraints for intercity transportation over a planning period. *Transportation Letters*, 2017. 16 pages, submitted.

[39] Balázs Dávid, Máté Hegyháti, and Miklós Krész. Linearly priced timed automata for the bus schedule assignment problem. In *Proceedings of the 4th International Conference on Logistic Operations Management - GOL'2018*, 2018. 7 pages, accepted.

[40] Renato de Matta and Emmanuel Peters. Developing work schedules for an inter-city transit system with multiple driver types and fleet types. *European Journal of Operational Research*, 192(3):852–865, 2009.

[41] Guy Desaulniers and Mark D Hickman. Public transit. *Handbooks in operations research and management science*, 14:69–127, 2007.

[42] Guy Desaulniers, Jacques Desrosiers, and Marius M Solomon. *Column generation*, volume 5. Springer Science & Business Media, 2006.

[43] M. Desrochers, J.K. Lenstra, M.W.P. Savelsbergh, and F. Soumis. Vehicle routing with time windows: Optimization and approximation. *Vehicle routing: Methods and studies*, 16:65–84, 1988.

[44] Jacques Desrosiers, Yvan Dumas, Marius M Solomon, and François Soumis. Time constrained routing and scheduling. *Handbooks in operations research and management science*, 8:35–139, 1995.

[45] D Dill. Timing assumptions and verification of finite-state concurrent systems. In J Sifakis, editor, *Proc. Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1990.

[46] Andreas T Ernst, Houyuan Jiang, Mohan Krishnamoorthy, and David Sier. Staff scheduling and rostering: A review of applications, methods and models. *European journal of operational research*, 153(1):3–27, 2004.

[47] EURO Special Interest Group on Cutting and Packing (ESICUP). Data set repository for bin packing, 2007. URL `https://paginas.fe.up.pt/~esicup/datasets`. Accessed: 2018-03-31.

[48] Richard Freling and José M. Pinto Paixão. *Vehicle Scheduling with Time Constraint*, pages 130–144. Springer Berlin Heidelberg, 1995. ISBN 978-3-642-57762-8. doi: 10.1007/978-3-642-57762-8_10. URL `https://doi.org/10.1007/978-3-642-57762-8_10`.

[49] Giovanni Luca Giacco, Donato Carillo, Andrea D'Ariano, Dario Pacciarelli, and Ángel G Marín. Short-term rail rolling stock rostering and maintenance scheduling. *Transportation Research Procedia*, 3:651–659, 2014.

[50] Giovanni Luca Giacco, Andrea D'Ariano, and Dario Pacciarelli. Rolling stock rostering optimization under maintenance constraints. *Journal of Intelligent Transportation Systems*, 18 (1):95–105, 2014.

[51] Vitali Gintner, Natalia Kliewer, and Leena Suhl. Solving large multiple-depot multiple-vehicle-type bus scheduling problems in practice. *OR Spectrum*, 27(4):507–523, 2005. ISSN 1436-6304. doi: 10.1007/s00291-005-0207-9.

[52] Vitali Gintner, Natalia Kliewer, and Leena Suhl. *A Crew Scheduling Approach for Public Transit Enhanced with Aspects from Vehicle Scheduling*, pages 25–42. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-73312-6. doi: 10.1007/978-3-540-73312-6_2.

[53] Pablo C Guedes and Denis Borenstein. Column generation based heuristic framework for the multiple-depot vehicle type scheduling problem. *Computers & Industrial Engineering*, 90: 361–370, 2015.

[54] Pablo C. Guedes and Denis Borenstein. Real-time multi-depot vehicle type rescheduling problem. *Transportation Research Part B: Methodological*, 108:217–234, 2018. ISSN 0191-2615. doi: https://doi.org/10.1016/j.trb.2017.12.012.

[55] Pablo Cristini Guedes, William Prigol Lopes, Leonardo Rosa Rohde, and Denis Borenstein. Simple and efficient heuristic approach for the multiple-depot vehicle scheduling problem. *Optimization Letters*, 10(7):1449–1461, 2016. ISSN 1862-4480. doi: 10.1007/s11590-015-0944-x. URL `https://doi.org/10.1007/s11590-015-0944-x`.

[56] Ahmed Hadjar, Odile Marcotte, and Franois Soumis. A branch-and-cut algorithm for the multiple depot vehicle scheduling problem. *Operations Research*, 54(1):130–149, 2006. ISSN 0030-364X. doi: 10.1287/opre.1050.0240.

[57] Ali Haghani and Yousef Shafahi. Bus maintenance systems and maintenance scheduling: model formulations and solutions. *Transportation Research Part A: Policy and Practice*, 36(5):453–482, 2002.

[58] Christopher A. Hane, Cynthia Barnhart, Ellis L. Johnson, Roy E. Marsten, George L. Nemhauser, and Gabriele Sigismondi. The fleet assignment problem: Solving a large-scale integer program. *Mathematical Programming*, 70(1):211–232, 1995. ISSN 1436-4646. doi: 10.1007/BF01585938.

[59] Dennis Huisman. *Integrated and dynamic vehicle and crew scheduling*. PhD thesis, Erasmus University Rotterdam, 2004.

[60] Dennis Huisman. Data instances for multiple-depot vehicle scheduling, 2007. URL `https://personal.eur.nl/huisman/instances.htm`. Accessed: 2018-03-31.

[61] Dennis Huisman and Albert P.M. Wagelmans. A solution approach for dynamic vehicle and crew scheduling. *European Journal of Operational Research*, 172(2):453–471, 2006. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2004.10.009.

[62] Dennis Huisman, Richard Freling, and Albert P.M. Wagelmans. A robust solution approach to the dynamic vehicle scheduling problem. *Transportation Science*, 38(4):447–458, 2004. doi: 10.1287/trsc.1030.0069.

[63] O.J. Ibarra-Rojas, F. Delgado, R. Giesen, and J.C. Muñoz. Planning, operation, and control of bus transport systems: A literature review. *Transportation Research Part B: Methodological*, 77:38–75, 2015. ISSN 0191-2615. doi: https://doi.org/10.1016/j.trb.2015.03.002.

[64] Julie Jespersen-Groth, Daniel Potthoff, Jens Clausen, Dennis Huisman, Leo Kroon, Gábor Maróti, and Morten Nyhave Nielsen. *Disruption Management in Passenger Railway Transportation*, pages 399–421. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-05465-5. doi: 10.1007/978-3-642-05465-5_18.

[65] Natalia Kliewer, Taieb Mellouli, and Leena Suhl. A time-space network based exact optimization model for multi-depot bus scheduling. *European Journal of Operational Research*, 175(3):1616–1627, 2006. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2005.02.030.

[66] Michael Kuby and Seow Lim. The flow-refueling location problem for alternative-fuel vehicles. *Socio-Economic Planning Sciences*, 39(2):125–145, 2005.

[67] Yung-Cheng Lai, Dow-Chung Fan, and Kwei-Long Huang. Optimizing rolling stock assignment and maintenance plan for passenger railway operations. *Computers & Industrial Engineering*, 85:284–295, 2015.

[68] Benoît Laurent and Jin-Kao Hao. Iterated local search for the multiple depot vehicle scheduling problem. *Computers & Industrial Engineering*, 57(1):277–286, 2009.

[69] Ladislav Lettovsky. *Airline operations recovery: An optimization approach.* PhD thesis, Georgia Institute of Technology, 1997.

[70] Jing-Quan Li. Transit bus scheduling with limited energy. *Transportation Science*, 48(4): 521–539, 2013.

[71] Jing-Quan Li. Battery-electric transit bus developments and operations: A review. *International Journal of Sustainable Transportation*, 10(3):157–169, 2016.

[72] Jing-Quan Li, Denis Borenstein, and Pitu B. Mirchandani. A decision support system for the single-depot vehicle rescheduling problem. *Computers & Operations Research*, 34(4):1008–1032, 2007. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2005.05.022.

[73] Jing-Quan Li, Pitu B. Mirchandani, and Denis Borenstein. The vehicle rescheduling problem: Model and algorithms. *Networks*, 50(3):211–229, 2007. doi: 10.1002/net.20199.

[74] Jing-Quan Li, Pitu B. Mirchandani, and Denis Borenstein. A lagrangian heuristic for the real-time vehicle rescheduling problem. *Transportation Research Part E: Logistics and Transportation Review*, 45(3):419–433, 2009. ISSN 1366-5545. doi: https://doi.org/10.1016/j.tre.2008.09.002.

[75] Andreas Löbel. *Optimale Vehicle Scheduling in Public Transit.* PhD thesis, Technischen Unversität Berlin, 1997.

[76] Marco E Lübbecke and Jacques Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023, 2005.

[77] Michael Meilton. *Selecting and Implementing a Computer Aided Scheduling System for a Large Bus Company*, pages 203–214. Springer Berlin Heidelberg, 2001. ISBN 978-3-642-56423-9. doi: 10.1007/978-3-642-56423-9_12.

[78] Marta Mesquita, Ana Paias, and Ana Respício. Branching approaches for integrated vehicle and crew scheduling. *Public Transport*, 1(1):21–37, 2009.

[79] Marta Mesquita, Margarida Moz, Ana Paias, José Paixão, Margarida Pato, and Ana Respício. A new model for the integrated vehicle-crew-rostering problem and a computational study on rosters. *Journal of Scheduling*, 14(4):319–334, 2011.

[80] Juan Carlos Muñoz and Ricardo Giesen. *Optimization of Public Transportation Systems*. American Cancer Society, 2011. ISBN 9780470400531. doi: 10.1002/9780470400531.eorms0935. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0935`.

[81] Kimmo Nurmi, Jari Kyngäs, and Gerhard Post. Driver rostering for bus transit companies. *Engineering Letters*, 19(2):125–132, 2011.

[82] Tomoshi Otsuki and Kazuyuki Aihara. New variable depth local search for multiple depot vehicle scheduling problems. *Journal of Heuristics*, 22(4):567–585, 2016. ISSN 1572-9397. doi: 10.1007/s10732-014-9264-z. URL `https://doi.org/10.1007/s10732-014-9264-z`.

[83] Sebastian Panek, Sebastian Engell, Subanatarajan Subbiah, and Olaf Stursberg. Scheduling of multi-product batch plants based upon timed automata models. *Computers & Chemical Engineering*, 32(1-2):275–291, 2008. ISSN 0098-1354. URL `http://www.sciencedirect.com/science/article/B6TFT-4PB6VXG-1/2/5a93b3cf39a0bb24cb7e69f6cdc01b03`.

[84] Ann-Sophie Pepin, Guy Desaulniers, Alain Hertz, and Dennis Huisman. A comparison of five heuristics for the multiple depot vehicle scheduling problem. *Journal of Scheduling*, 12(1):17, 2008. ISSN 1099-1425. doi: 10.1007/s10951-008-0072-x.

[85] Ann-Sophie Pepin, Guy Desaulniers, Alain Hertz, and Dennis Huisman. A comparison of five heuristics for the multiple depot vehicle scheduling problem. *Journal of Scheduling*, 12(1): 17–30, 2009.

[86] Emmanuel Peters, Renato de Matta, and Warren Boe. Short-term work scheduling with job assignment flexibility for a multi-fleet transport system. *European journal of operational research*, 180(1):82–98, 2007.

[87] Victor Pillac, Michel Gendreau, Christelle Guéret, and Andrés L Medaglia. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11, 2013. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2012.08.015.

[88] Harilaos N. Psaraftis. Dynamic vehicle routing problems. *Vehicle Routing: Methods and Studies*, 16:223–248, 1988.

[89] Harilaos N. Psaraftis, Min Wen, and Christos A. Kontovas. Dynamic vehicle routing problems: Three decades and counting. *Networks*, 67(1):3–31, 2016. ISSN 0028-3045. doi: 10.1002/net.21628.

[90] Gerhard Reinelt. TSPLIB. University of Heidelberg, `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`, 2013. Accessed: 2018-03-31.

[91] J Reuer, N Kliewer, and L Wolbeck. The electric vehicle scheduling problem: A study on time-space network based and heuristic solution approaches. In *Proceedings of the 13th Conference on Advanced Systems in Public Transport (CASPT), Rotterdam*, 2015.

[92] Celso C. Ribeiro and François Soumis. A column generation approach to the multiple-depot vehicle scheduling problem. *Operations Research*, 42(1):41–52, 1994. doi: 10.1287/opre.42.1.41.

[93] Jean-Paul Rodrigue, Claude Comtois, and Brian Slack. *The geography of transport systems*. Routledge, 2009. ISBN 978-0415483247.

[94] J. L. Saha. An algorithm for bus scheduling problems. *Journal of the Operational Research Society*, 21(4):463–474, 1970. ISSN 1476-9360. doi: 10.1057/jors.1970.95.

[95] F. Zeynep Sargut, Caner Altuntaş, and Dilek Cetin Tulazoğlu. Multi-objective integrated acyclic crew rostering and vehicle assignment problem in public bus transportation. *OR Spectrum*, 39(4):1071–1096, 2017.

[96] Ingmar Steinzen, Vitali Gintner, Leena Suhl, and Natalia Kliewer. A time-space network approach for the integrated vehicle-and crew-scheduling problem with multiple depots. *Transportation Science*, 44(3):367–382, 2010.

[97] Subanatarajan Subbiah, Thomas Tometzki, Sebastian Panek, and Sebastian Engell. Multi-product batch scheduling with intermediate due dates using priced timed automata models. *Computers & Chemical Engineering*, 33(10):1661–1676, 2009. ISSN 0098-1354. doi: 10.1016/j.compchemeng.2009.05.007. URL `http://www.sciencedirect.com/science/article/pii/S0098135409001197`.

[98] Uwe H Suhl, Swantje Friedrich, and Veronika Waue. Progress in solving large scale multi-depot multi-vehicle-type bus scheduling problems with integer programming. *Wirtschaftinformatik Proceedings 2007*, 2:429–446, 2007.

[99] Dušan Teodorovic and Goran Stojkovic. Model to reduce airline schedule disturbances. *Journal of Transportation Engineering*, 121(4):324–331, 1995. doi: https://doi.org/10.1061/(ASCE)0733-947X(1995)121:4(324).

[100] Attila Tóth and Miklós Krész. A flexible framework for driver scheduling. In *Proceedings of the 11th International Symposium on Operational Research in Slovenia SOR'11*, pages 341–345, 2011. ISBN 978-961-6165-35-8.

[101] Ezgi Uçar, Ş. İlker Birbil, and İbrahim Muter. Managing disruptions in the multi-depot vehicle scheduling problem. *Transportation Research Part B: Methodological*, 105:249–269, 2017. ISSN 0191-2615. doi: https://doi.org/10.1016/j.trb.2017.09.002.

[102] M. E. van Kooten Niekerk, J. M. van den Akker, and J. A. Hoogeveen. Scheduling electric vehicles. *Public Transport*, 9(1):155–176, 2017. ISSN 1613-7159. doi: 10.1007/s12469-017-0164-0.